On Modeling and Tolerating Incorrect Software

Anish Arora
The Ohio State University
Columbus, OH 43214
anish@cis.ohio-state.edu

Marvin Theimer
Microsoft Research
Redmond, WA 98052
theimer@microsoft.com

ABSTRACT

Distributed systems have to deal with the following scenarios in practice: bugs in components; incorrect specifications of components and, therefore, incorrect use of components; unanticipated faults due to complex interactions or to not containing the effects of faults in lower-level components; and evolution of components. Extant fault tolerance models deal with such scenarios in only a limited manner. In particular, we point out that state corruption is inevitable in practice and that therefore one must accept it and seek to correct it.

The well-known concepts of detectors and correctors can be used to find and repair state corruption. However, these concepts have traditionally been employed to immediately detect and correct errors caused by misbehaving system components. Immediate detection and correction is often too expensive to perform and hence we consider the implications of running detectors and correctors only intermittently.

More specifically, we address issues that must be dealt with when state corruption may persist within a system for a period of time. We show how to both detect and correct state corruption caused by infrequently occurring "transient" errors despite the ability for it to actively spread to other parts of the system. We also show how to eventually detect all state corruption, even in cases where continually recurring errors are constantly introducing new state corruption. Finally, we discuss the minimum set of capabilities needed from a trusted base of software in order to guarantee the correctness of our algorithms.

This work was partially sponsored by Microsoft Research, and by a 2001 unrestricted gift award from Microsoft Research University Relations

1. Introduction

1.1 State Corruption as a Fact of Life

Incorrect software and the attendant possibility for system state corruption is a fact of life:

- Incorrectly implemented software methods are inevitable in a large system and hence normal operation of the system will corrupt system state unless additional measures are taken.
- Incorrect specifications for software methods are inevitable in a large system and hence even correctly implemented methods may end up violating the set of invariants that the system is supposed to maintain. This problem is exacerbated by the fact that, in practice, very few people – if anyone – understand all the state guarantees and behaviors that an entire system is supposed to maintain. Hence, unintended consequences and unanticipated interactions and inconsistencies among system components will arise due to misconceptions held by various system designers and developers.
- Evolution of functionality is a common occurrence for many large systems. This makes the likelihood of incorrect specifications (as well as incorrect implementations) even higher. Equally importantly, for long-running systems, it introduces the requirement to upgrade a system while it is in operation. Especially for distributed systems, performing an upgrade in a globally consistent manner is a difficult problem. Hence components may periodically be incorrect because they have not yet been upgraded or because system administrators misunderstood how to correctly update all components in a globally consistent manner.

Traditionally, tolerance to incorrect software has been designed by relying on multiple versions of the

software (in space or in time), at least some of which are known to execute correctly. Examples include the N-version fault-tolerance and the recovery blocks approaches. The problem with these approaches is that they are difficult to apply to entire, large systems and aren't perfect in any case:

- N-version programming applied to large components risks having sufficient numbers of bugs in each version that eventually versions drift apart from each other.
- N-version programming applied to small components makes it difficult to come up with N different versions of each component that do not have correlated error behaviors.
- Recovery-block programming only deals with known errors; it cannot deal with unanticipated errors that may be lurking in the code. Stated differently, recovery-block programming only reduces the number of errors in a component, it cannot guarantee that it will be able to detect and compensate for all the errors that might occur.

The net result is that, in practice, it is infeasible to get *all* the bugs out of a system. Furthermore, these approaches do not deal with system specifications that might themselves be incorrect (in the sense that they are inconsistent).

1.2 Summary of Contributions

In this paper, we study how to complement these approaches with an approach that assumes that all versions of a software component, and indeed its very specification, may be subject to incorrectness and hence that state corruption/inconsistency is a fact of life that must be tolerated. We introduce an incremental, timing-independent, invariant-based approach to detecting and correcting corrupted software state after the fact by means of special detector and corrector methods that are associated with each software component of a system. Our approach is intended to be supplementary to other means of achieving software correctness and fault tolerance.

We divide the problem of dealing with state corruption into two cases: a *transient* case wherein state corruption is newly introduced into a system only a finite number of times and an *incorrigible* case wherein new state corruption is introduced into a system on a continual basis. We show how to eventually correct all state corruption in the system for the transient case and how to infinitely often detect the presence of state corruption for the incorrigible case. We also show that it is impossible to always (or eventually always) accurately distinguish whether a system is in the transient or the incorrigible case.

We present results concerning what can and cannot be done with detectors and correctors for individual software components in isolation as well as for distributed aggregates of components that interact with each other. To deal with the latter efficiently, we introduce both a model of how corruption can spread among components—even by means of inherently correct methods—and a distributed *cleaning* algorithm. Our cleaning algorithm corrects all corruption in the transient errors efficiently, by arresting the spread of corruption and by preventing the corruption from re-infecting cleaned components.

Because a-posteriori detection and correction of state corruption is an indirect approach to dealing with incorrect software we employ the notion of suspicion and introduce suspector algorithms. These are necessary in order to improve the rate at which a system detects and corrects state corruption.

Finally, we discuss how to deal with incorrect detector and corrector software and define the minimum *trusted base* of software needed in order for our approach to be able to succeed. We also discuss how to implement this trusted base of software.

1.3 Correctness through Enforcement of Invariants

In a setting where state corruption cannot be prevented and system specifications themselves may be inconsistent the very notion of system correctness can only be defined approximately. One way to define correctness is to approximate it with a set of invariants that the system state should be made to obey. This approach has two advantages: (a) it defines correctness in terms of current system state rather than timing-dependent and possibly incorrectly specified update actions and (b) it makes inconsistent system specifications visible as irreconcilable invariant violations. The time-independent nature of invariant specifications means that one can reason

about them without having to deal with the nasty difficulties of temporal interleaving of software methods. The true-false nature of invariants implies that we will be unable to repair an inconsistently specified system to a state wherein all invariants return true. Thus, inconsistent specifications will result in explicit violations rather than silent divergence of system state from its intended values.

As mentioned, system correctness is only approximated by this approach since the set of invariants defined may not completely encode the intended specification of the system. This is both a strength and a weakness: System developers can start out by specifying only a few invariants that describe the most important aspects of a system's state and then improve the accuracy of the approach by incrementally adding additional invariants as time and desire permits. For the same reason, however, the approach should be used as a supplement to traditional methods of achieving system correctness rather than as a replacement for them.

Since we assume that state corruption is inevitable the only way to achieve correct system state over the long run is to introduce a notion of continually searching for, finding, and correcting corrupted system state. That is, we employ the well-known notion of *detectors* that check the invariants defining a system's correctness and the notion of *correctors* that correct system state once corruption has been discovered by a detector.

1.4 Corruption Containment and Suspicion

In the absence of resource and performance constraints, one could check for system state corruption after every client method invocation by running all detectors in order to check all system invariants. Any detected state corruption could be corrected immediately by running the relevant correctors. The result would be that state corruption would be corrected before it was ever noticeable by anyone.

However, this approach is generally too expensive: in practice one can only afford to run detectors and correctors intermittently. Consequently state corruption will persist in a system component for some period of time. During that time the corruption may spread to additional parts of the component's state as well as to other components within the system and to clients of the system.

This spreading of corruption can occur even via correct methods. Letting *m* be an incorrect method in a component *C*, we observe that:

- If an execution of *m* corrupts the state of *C*, then unless the invariants of *C* are restored immediately subsequent execution of even correct methods in *C* may yield incorrect results or further state corruption.
- Similarly, if an execution of *m* yields incorrect outputs to other methods (whether or not they are in *C*) then their subsequent execution may yield incorrect results or corruption of their own invariants.

This implies that, when trying to detect and correct an instance of state corruption we must take into account that it may simultaneously be spreading to other state as well as potentially returning to state that was previously corrupted and that we have already corrected. Thus, an important factor to consider in the design of our detection/correction algorithms is how to control the rate at which corruption in one component can *infect* other components and, equally importantly, how to prevent already-corrected system state from being *re-infected* through the internal activities of a component or its external interactions with other components.

We attack this problem by defining a simple, generic contamination model based on interaction among system components only via well-defined, remote method invocations. Thus components are independent of each other and, in particular, do not share state with each other. We assume the ability to atomically "clean" the state of a component, the ability to intercept remote method invocations before they are executed, and the ability to ask a component to clean itself before allowing a remote method invocation it has requested to continue. Given these assumptions we can impose a "transitive cleaning procedure" that will enable us to both contain, as well as partially control the rate at which contamination spreads among system components.

Controlling the rate at which system corruption can spread is important not just for reducing the amount of time that a system is in a corrupted state; it is also important for correlating detected state corruption to

the incorrect software methods that were its original cause. Since we cannot afford to run detectors after every software method invocation, the most we can hope to do is to lay *suspicion* on particular software methods as being incorrect. This is important because, as discussed later on, our approach can only promise to repeatedly flag, but not repair, state corruption that occurs due to continually recurring software errors. Explicitly flagging that repeated state corruptions are occurring is useful for alerting the administrators and users of a system that it is in an incorrect state, but that usefulness is significantly increased to the extent that it also allows system developers/administrators to debug which system components are incorrect.

Deciding whether or not one or more software methods of a given system component are suspicious will, in turn, also allow us to refine our decisions about when and where to run detectors. By running detectors more frequently when a system component is suspected of behaving incorrectly, or of having suspicious neighbor components with which it interacts, we can try to spend the limited resources available in a system for detection in a more effective manner.

1.5 Transient versus Persistent Errors

In addition to worrying about the spread of existing state corruption we must also worry about how often incorrect software causes new introductions of system state corruption. It is useful to divide this problem into two cases: systems in which new introductions of state corruption occur infrequently and systems in which these introductions occur on a more regular basis.

In the former case we can try to deal with a single instance of introduced state corruption – and any subsequent spread of that corruption – in isolation. That is, we can assume that subsequent invocations of system methods – including our detectors and correctors – will involve only correct software and that corruption can actively spread only through use of corrupted state. For this case we are able to devise an algorithm that will enable a system's detectors and correctors to eventually find and correct all the state corruption that might occur in the system. A key subcomponent is the corruption containment scheme introduced in Section 1.3.

In the latter case we allow state corruption to be introduced into any part of the system arbitrarily often. In general, this makes it impossible to say anything specific about how "correct" the state of a system can be made. However, we *can* ensure that continually repeated introductions of new state corruption by a particular method will eventually and continually be detected.

1.6 Trusted Software Base

An important question to ask about our design is what assumptions it requires regarding the integrity of various underlying software components. If *any* software component in the system can be corrupted then it becomes impossible to guarantee that system state corruption can always be detected, let alone corrected.

Since the work presented in this paper depends fundamentally on the use of detector/corrector software, the key to its correctness lies with ensuring that detectors and correctors do not themselves misbehave. We make two kinds of assumptions in order to achieve this: We assume that system invariants are specified in an overlapping manner and in sufficient numbers that checking them all will effectively show up any errors in detector/correctors as irreconcilable invariant violations. We also assume the availability of a small trusted base of software that enables detectors and correctors to correctly interact with the rest of the system in order to get their jobs done.

We furthermore show that the trusted base of software need only ensure the following things:

- That detectors and correctors are able to correctly read and write system state.
- That they are scheduled in a manner that cannot be predicted by the rest of the system's components.
- That they are able to reliably store private state that cannot be corrupted by other components in the system.

We also discuss how to implement the capabilities of this trusted base by relying on either separate address spaces within each machine or on separate machines.

The rest of this paper is organized as follows. In Section 2 we present a model of incorrect method faults in systems, and develop the assumptions on which our framework is based. We then present our model of and method for tolerance to incorrect methods: in Section 3, we consider the case where there is a single component, and in Section 4, we consider the case of general, distributed systems. We develop the trusted base framework in Section 5. Finally, we present related work in Section 6, and discuss key issues in our work and conclusions in Section 7.

2. System and Fault Model

2.1 System Model

A *system* consists of a set of components. Each *component* has *state* and a set of *methods*. The state of a component is given by a value in some predefined domain for each state *variable* in the component. Each method has *input* and *output* arguments, that also range over corresponding predefined domains, and its meaning is specified by a relation from inputs and (pre) state values to outputs and (post) state values. We refer to this relation as the *specification* of the method.

The semantics of a system is given by a set of *correct computations*, where each computation is a potentially infinite alternating sequence s.0, e.1, s.1, e.2, s.2, … where each s.j is a system state, comprising of the state of each component, and e.j is an *event* consisting of a method name, input values, and output values. By correct, we mean that for each event e.j the input values and s.j are related to the output values and s.(j+1) according to the specification of the method. Note that method executions are "atomic" with respect to computations; atomicity means that each method execution is effectively uninterrupted and logically instantaneous.

We assume:

Components do not share state variables.

Moreover, we assume:

Components are independent.

That is, the correctness of a component —i.e., the requirement that each of its methods satisfy their specification upon execution— is independent of other components, even those whose methods it may invoke and those who may invoke its methods. Thus, in a correct system computation, every corresponding component computation (i.e., the projection of the system computation to the invocation of methods on that component) is correct. Vice versa, the composition of correct component computations yields a correct system computation.

A predicate *p* on the state of a component *C* is an *invariant* of *C* iff every correct computation of *C* starts at a state where *p* is satisfied, and *p* remains satisfied at every state in the computation. Note that the conjunction of invariant state predicates of *C* is also an invariant of *C* (as is the disjunction).

We assume:

Associated with every correct component is a set of invariants.

Intuitively, the set of invariants is intended to suffice to prove that in every correct computation of the component every step is correct in the sense of satisfying the corresponding method specification. Maintaining all of these invariants suffices to maintain the correctness of the component. In other words, if after any correct computation prefix of *C*, *C*'s state is corrupted so its invariant does not hold, reestablishing the invariant preserves computation correctness.

It should be emphasized that our use of invariants is stylized: the predicate *true* is indeed satisfied by every state of component *C* and hence holds "invariantly'' in every computation of *C*, yet it is inadequate as an invariant predicate (it may be insufficient for proving that every computation of *C* satisfies its problem specification). In other words, the invariants which we have assumed are not just predicates that remain satisfied in computations of *C*, but are predicates that are also sufficient for proving the correctness of *C*. As one example, the set of reachable states of *C* sometimes yields an adequate invariant for our purposes. As another example, sometimes the set of reachable states of *C* coupled with auxiliary variables –maintaining history and prophecy information– yield adequate invariants.

Finally, we assume:

Each component C has a detector & corrector.

The detector witnesses whether or not any invariants of C are violated, and the corrector ensures that C is placed in a state where all its invariants hold. They can be run atomically at any time, including before any remote method invocation on it, and before any remote method invocation it requested continues.

Note that the model does not specify whether a component maintains its state in a persistent or in a volatile manner. Nor does the model distinguish whether components are "clients", "servers", or "peers" of other components. Furthermore, it is closed in the sense that (i) component state is not affected by "external interactions", i.e. interactions with anything other than the system components, and (ii) the output of a component method does not affect anything other than other system components. Said another way, components external to the system are treated as though they are specially marked components in the system scope.

2.2 Fault Model

Informally, our fault model aims to capture "incorrect" behavior of methods and components, in that they exhibit any of the problems described in the Introduction, including incorrect or incomplete specification and implementation, as well as side effects due to partially deployed new versions of software (i.e. evolution).

Formally, we say a method is

- *incorrect* iff for some component (pre) state and inputs where the method is defined, execution of the method yields a (post) state and/or outputs that do not satisfy the specification of the method.

We say a component is

- *correct* iff it has no incorrect method; else, the component is *incorrect*.

Upon execution of an incorrect method given a component (pre) state and inputs where the method is defined, faulty behavior of the component may manifests itself in at most two ways:

(1) the component state is corrupted, and/or

(2) an incorrect output is returned to the caller.

In turn, these sorts of faulty behavior may lead to yet another sort of faulty behavior during subsequent invocation of the component or the caller:

(3) the method may be undefined for the given pre-state and input.

We say that a method execution is

- *faulty* iff either the method relation is not defined for the inputs & (pre) state, or the outputs & (post) state do not satisfy the method specification.

A computation is

- *faulty* iff it has a faulty method execution (else, it is correct by definition).

Note that all computations may be divided into three groups where incorrect methods respectively execute: correctly in all steps; incorrectly in a finite number of steps; or, infinitely often incorrectly.

A computation is

- *transiently faulty* iff it has a finite number of faulty method executions.
- *transiently perturbed* iff it has a finite number of faulty method execution of type (1) or (2); that is, where the outputs & (post) state do not satisfy the method specification even though the method relation is defined for the inputs & (pre) state.
- *incorrigibly faulty* iff it has an infinite number of faulty method executions.

Note that a transiently faulty computation is always transiently perturbed; a transiently perturbed one may be either transiently or incorrigibly faulty; but a transiently faulty one is not incorrigibly faulty and vice versa.

Thus, the semantics of a system with some incorrect components is given by its correct computations and its faulty computations. In each faulty computation, some method executions do not conform to the specifications of their methods (recall that all events in correct computations conform to their corresponding specification.) In particular, the first faulty execution exhibits faulty behavior of the sort of (1) or (2), but not (3). Subsequent faulty executions, if any, may exhibit (3) as well.

A few points are worth emphasizing. A faulty execution of sort (1) implies that in case the component has many instances, then the state of the particular instance where the fault occurred is corrupted. In other words, the states of other instances of the component are not directly corrupted by the fault occurrence.

A faulty execution of sort (2) implies that the state of an invoking component may become indirectly corrupted upon using the outputs yielded by an incorrect method, even if the invoking component itself has only correct methods. In turn, this corrupted state of the invoking component may yield incorrect outputs when its methods are invoked, thereby further propagating the state corruption. In other words, a fault occurrence in a component may yield a spreading of state corruption to other components that (directly or indirectly) depend on the former, even if the latter are not subject to faults themselves.

Detector and corrector methods may be faulty, not only for the same reasons as component methods are incorrect, but also because the task of determining component invariants is typically incremental and the set of invariants discovered at any point may not be complete even if they are sound (i.e., they may hold at each state in correct component computations, but they may also hold at states in incorrect component computations), or may not even be sound (i.e., if some of the invariants are inconsistent, they may not hold at some states in correct component computations).

2.3 Tolerance Framework Assumptions

Towards designing a tolerance framework that deals with the fault model described above, we assume the following three properties:

- **State Accessibility** *Component methods, faulty or otherwise, cannot prevent component state and invocations from being observed and controlled.*
- **Angelic Schedulability** *Component methods, faulty or otherwise, cannot anticipate events, i.e. which methods are invoked when, by whom, and with which inputs.*
- **Event Atomicity** *All events, faulty or otherwise, are atomic, i.e., they are terminating, and effectively occur instantaneously and without interference from others.*

State Accessibility implies that it is feasible to devise a mechanism to read/write the state of each component and to log/initiate invocations. Angelic Schedulability implies that it is feasible to devise a mechanism to randomly or at any time schedule distinguished events.

Event atomicity implies that we eschew the possibility of a crashing, nonterminating or divergent execution of an incorrect method. Note, however, that an incorrect fault method can corrupt the state of its component so severely so that all methods of that component methods are essentially “disabled”, mimicking the effect of component crash/failstops.

That said, our assumption precludes crash/failstop fault models per se. It focuses the framework on dealing with incorrect methods (and incorrect specifications) that introduce initially undetectable state corruption and wrong results, whereas crash/failstop faults focus attention on dealing with catastrophic events that are immediately detectable and assumed not to cause subsequently visible corruption. Examples of such events include machine failures as well as component failstops initiated in response to the detection of incorrect system behavior. Thus, our tolerance approach is primarily about detection and correction of errors after the fact whereas crash/failstop schemes are primarily about prevention of errors in the first place. Another distinction is that our tolerance approach relies on the use of specification (in the form of derivative invariants) to add robustness whereas crash/failstop schemes do not.

Towards separation of concerns, we have chosen not to address crash/failstop model in the rest of the paper, although we claim that since crash/failstop are orthogonal to incorrect methods, their consideration can be later incorporated in the tolerance framework that we propose in this paper. We also claim that dealing with incorrect methods via the tolerance framework that we propose may indirectly reduce the number of crash/failstop failures that a system is subject to.

3. Tolerance to Self Faults in Components: Model & Approach

In this section, we present a model for an individual system component to tolerate its own incorrect method faults, and an approach to implement this tolerance model that is based on detectors and correctors of state corruption. As a related matter, we consider how to handle incorrect method faults in detectors and correctors themselves. We relegate to the next section the question of component tolerance to the incorrect method faults of other components.

3.1 A Model of Self Fault Tolerance

It follows from the fault model that the net effect of incorrect method executions of a component on itself is that its state is corrupted and its invariants may no longer hold. Invariant violations will occur only finitely in transient faulty computations of the component and possibly infinitely in incorrigibly faulty computations.

An appropriate model of tolerance to self faults is therefore:

- In transiently faulty computations of the component, eventually the invariants of the component should hold always. In other words, once incorrect method executions stop occurring, the invariants should from some subsequent state onwards hold forever. To be of practical use, of course, the invariants should be restored as soon as possible.
- In incorrigibly faulty computations of the component, the component should infinitely often be suspected, i.e. "marked" as being incorrect. In other words, the component and others should detect that its method executions are often incorrect. Again, practically speaking, suspicion should be accomplished as soon as possible and the longer it remains the better its use. The suspicion should not however be infinitely often in transiently faulty computations; i.e., the suspicion should eventually quiesce.

Of course, a more desirable tolerance to self fault would be to eventually suspect forever in incorrigible faulty computations (and to eventually unsuspect forever in transient faulty computations). That tolerance is however impossible to achieve in the general case, as we show later in this section.

3.2 Detectors and Correctors

Detectors and correctors provide the building blocks for implementing the tolerance model described above. Intuitively, detectors witness invariant violations and —by establishing that the violation count exceeds some threshold— implement marking in incorrigible faulty component computations. In contrast, correctors enable restoration of invariants and thus implement eventually always correct execution in transiently faulty component computations.

3.2.1 Detectors & incorrigibly faulty computations

Consider a method that atomically accesses the state of a component *C* and checks whether that state satisfies an invariant *p* of *C*. Implementing such a *detector* of *p* requires the following property:

> **Trusted Read** The state of a component can be *read* accurately and atomically.

Given State Accessibility and Event Atomicity, we know that this property is satisfiable, regardless of how corrupted the component state is. In other words, even if the component is deadlocked, this property enables a check on the component state.

Assume for now that the detector method is, by itself, correct. In this case, State Accessibility and Angelic Schedulability imply that faults can neither prevent the detector from checking the state of *C* nor always schedule corruptions and subsequent invocations that themselves self-correct the state so that violations of *p* are hidden from the detector. Thus, if the detector is executed after every event of *C*, every violation of *p* will be observed.

For efficiency, however, we must consider the case where the detector cannot execute after every event of *C*. In this case we will want to eventually detect corrupted state so that it can eventually be corrected. To do this, we must ensure that an eventually executed detector will never "miss" violations of *p* forever. In particular, we must ensure that the implementation for such a detector ensures that its scheduling does not "miss" violations of *p* forever.

Hence, given Angelic Schedulability, we may additionally require the following property:

> **Trusted Schedule** Random Scheduling is possible.

This property implies that the scheduler can produce every possible timing sequence for detector execution, and for the detector to be executed at those times. Henceforth in this paper, we assume that detectors and correctors are scheduled randomly. It follows that

Theorem 1 (detectability of invariant violation)

> *In any computation of C where a detector of an invariant p of C is executed infinitely often:*
>
> - *if p is violated infinitely often, violations of p will be detected infinitely often.*

Proof of Theorem 1 follows from—in addition to State Accessibility, Angelic Schedulability, Trusted Read, and Trusted Schedule— the observation that detectors simply evaluate the invariant predicate in the current state of the component. Evaluation of the invariant predicate may be reasonably instrumented to not rely on invoking methods of other components, which may be incorrect and whose incorrect outputs may corrupt the state of the detector. Based on the independence of the detector from other system components, it follows from Angelic Schedulability and Trusted Schedule that in any computation where *p* is violated infinitely and the detector is executed infinitely often, then with probability 1 the detector will eventually be scheduled to execute immediately after some violation (or between some violation and a subsequent method invocation that by itself corrects the component state). In this eventuality, from State Accessibility and Trusted Read, the violation of *p* will be detected.

Theorem 1 provides a basis for implementing a suspector of component incorrectness. The simplest of schemes for marking incorrectness follows from:

Fact 1 (threshold-based suspicion)

> *In any computation of C where a detector of an invariant p of C is executed infinitely often, if the number of violations of p is infinite then the number of detected violations will exceed any fixed integer threshold (that is used for suspecting C as incorrect).*

3.2.2 Correctors & transiently faulty computations

Consider a method that atomically corrects the state of a component *C* to reestablish an invariant *p* of *C* in case *p* is violated in the component state. Implementing such a state *corrector* of *p* requires the following property:

> **Trusted Write** The state of any component can be accurately and atomically *overwritten*.

Note that this property implies that component state may be written regardless of how corrupted it is. In other words, even if the component is deadlocked, this property enables its state to be overwritten. The satisfiability of this property follows from State Accessibility.

Assume for now that the corrector method is, by itself, correct. In this case, State Accessibility and Angelic Schedulability imply that faults can neither prevent the corrector from reestablishing the invariant nor always "time" the state corruptions so that the violation of *p* is hidden from the corrector. Following the reasoning for detectors, i.e. based on Trusted Read, Schedule, and Write and assuming (again, for now) that correctors work independently of any other components in the system, we have

Theorem 2 (correctability of invariant violation)

> *In any computation of C where a corrector of an invariant p of C is executed infinitely often, if p is ever violated, eventually that violation of p will be corrected.*

Proof of Theorem 2 is similar to that of Theorem 1.

If the invariant *p* is both sound and complete, as discussed earlier, then subsequent computation of *C* will be correct. In this case, we have

Corollary 2 (invariant stabilization)

> *In any computation of C where a corrector of an invariant p of C is executed infinitely often, if the computation is transiently perturbed, eventually p will hold at every state and the computation will not be incorrigible.*

Unlike for detectors, the assumption that correctors do not depend on other system components is a severe one. Sometimes, correctors do rely on other

system components. A standard example is a corrector that maintains a log of all events that have occurred on C in order to correct the state of C to a "recent" uncorrupted state upon detecting violation of p. In this example, the corrector depends on methods that provide outputs about states and events of C that have occurred, and since these outputs are potentially corrupted, it is possible that the state of the corrector itself gets corrupted. This possibility is avoided by ensuring the following property:

> **Trusted Store** Data values associated with a key can be *stored* accurately and atomically, guaranteed to remain uncorrupted regardless of all system states and events, and retrieved accurately and atomically given the key.

It is now possible to implement correctors that reestablish the invariant (ideally to an "interesting" state of C in the sense of backward recovery, as in the standard example discussed above, or forward recovery with respect to states in the current computation of C) because the trusted store enables the correctors to use data values without themselves being corrupted. Trusted Store enables the Correctability of Invariant Violation Theorem to hold even if we no longer assume that correctors are independent of other components in the system. (Note that in this discussion a corrector not only needs to detect that the "current" state of a component is uncorrupted but also needs to detect that a candidate "interesting" state of the component for purposes of reestablishing the invariant is also uncorrupted.)

3.3 Stabilizing Suspicion in Transiently & Incorrigibly Faulty Computations

One aspect of the desired model of tolerance to self-faults remains to be discussed: The suspicion of a component should occur infinitely often in incorrigibly faulty computations but at most finitely in transiently faulty computations. The threshold-based suspicion scheme mentioned previously satisfies the former requirement but not the latter. One scheme to satisfy the latter requirement—that is stabilizing the suspicion in transiently faulty computation so that eventually there are no false positive suspicions—is to augment the threshold-based scheme with "unsuspicion" actions. These actions undo false positives in suspicion that may have resulted from transient state corruption, but should the state corruption be incorrigible, suspicion would resume (infinitely often). In the simplest of settings, unsuspicion may occur periodically after suspicion occurs. This scheme is a witness for

Theorem 3 (suspicion stabilization)

> *There exist suspicion programs that in any computation of C, where the detector of C is scheduled infinitely often,*
> - *suspect C infinitely often if p is violated infinitely often, and*
> - *unsuspect C eventually always if p is violated only finitely.*

Note that in the witness scheme described above the unsuspicion actions may repeatedly yield false negative suspicions in incorrigible computations; i.e. they may unsuspect *C* infinitely often even when *C* keeps producing incorrect computation.

The problem is not limited to our witness scheme. It is impossible in general to suspect intelligently in the sense of "suspecting C eventually always if p is violated infinitely often". Intelligent suspicion must be able to predict unbounded futures where corruptions will occur. But, without giving the suspicion program some additional semantic knowledge about *C*, there is always some probability that in a computation where *p* is violated infinitely often, *C* is not be suspected infinitely often. That is, there may be false negative suspicions infinitely often.

Theorem 4 (impossibility of stabilizing to no false negative suspicions)

> *There exists no suspicion program that in any computation of C, where the detector of C is scheduled infinitely often, eventually always*
> - *suspect C if p is violated infinitely often, and*
> - *unsuspect C if p is violated only finitely.*

Proof of impossibility follows from the observation that to meet the second requirement, the suspicion program (in particular its unsuspicion actions) must be scheduled infinitely often in any computation of C upon starting from any state of C. In computations of

C where p holds infinitely often and p is violated infinitely often, since the suspicion program is assumed to have no semantic knowledge about *C*, with probability 1 unsuspicion actions will eventually be scheduled at states where p is not violated, and hence eventually unsuspicion will occur. Since this argument applies upon starting from any state, there are computations where unsuspicion occurs infinitely often.

3.4 Tolerating Faulty Detectors & Correctors

As discussed above, one could in principle consider executing detectors that check each invariant after every event. This would result in the immediate detection of any corrupted component state. There are at least two problems with this approach: We have already observed that this would be too expensive to do. Another problem is that the detector methods might themselves be incorrect: They may have bugs, be designed with incorrect specification in mind, be inconsistent with respect to other invariants, or be incomplete in the sense of not detecting the full set of predicates needed to establish correct component computation. Alternatively, the corruption of the state of the component may affect them as well, and hence affect their correctness. Consequently immediate detection of corrupted system state cannot be assured.

Our approach deals with incorrect detectors and correctors, in several ways. One is to introduce sufficient redundancy in invariants. Thus, a few incorrect detectors and correctors may be compensated for, or at least they may be "marked" should discrepancies be found regularly. This is analogous to the concept of double-entry bookkeeping in accounting.

A second way is to avoid the possibility that state corruption from the component affects the correctness of detectors and correctors, and vice versa. To this end, replicas of detectors and correctors may be executed in address spaces that are separate from that of the component, and may even reside on separate machines. We will address the implications of this distribution of execution in Section 5, which discusses our Trusted Base framework.

A third way is to task the rest of the system with detecting the incorrectness of the detectors and correctors. This will be discussed in the next section, where we consider the broader question of component tolerance to the incorrect method faults of other components.

4. Local System Tolerance to Regional Faults: Model & Approach

Thus far, we've discussed detection in incorrigible computations where invariant predicates are violated infinitely often, but not in incorrigible computations where incorrect outputs occur infinitely often—which may in turn infinitely often violate invariant predicates and/or outputs of neighboring components. We now turn to how the affected neighbors may carry out such detection, in the context of system level tolerance.

More specifically, in this section, we begin with a model of local tolerance in the system to incorrect components, and then discuss how detectors and correctors may be used to achieve the desired tolerance. As a related matter, we also consider how tolerance is achieved despite incorrect components and/or components whose detectors/correctors are themselves incorrect.

4.1 A Model of Local Tolerance

It follows from the fault model that the net effect of an incorrect component on the rest of the system is that the incorrect results produced by the component may spread corruption to the state of other components that invoke its method or whose methods are invoked by it. E.g., execution of an incorrect method of *C* may not only corrupt the (post) state of *C* (and thereby violate the invariants of *C*), it may also corrupt the state of a component *D* that interacts with *C*. In turn, execution of the methods of *D* may themselves yield incorrect outputs and (post) states —even if they themselves are correct— thus possibly corrupting the state of a component *E* that interacts with *D*.

This non-local spreading of corruption complicates the task of maintaining the states of each component so that their invariants hold. In other words, tolerance to incorrect components not only involves maintaining the state of the corresponding components but also those of other components. And,

intuitively, the speed of the maintenance is inversely related to the spreading of the data corruption in a system.

An appropriate model of tolerance to system tolerance is therefore one where the two following requirements hold:

- In transiently faulty computations of the system, eventually the invariants of *all* components should hold always. In other words, once incorrect method executions stop occurring, the system should be restored to a legitimate state. To be of practical use, of course, the invariants should be restored as soon as possible and as few components should be affected as possible.
- In incorrigibly faulty computations of the system, *each* incorrect component that produces incorrigible behavior should be suspected infinitely often by itself and/or its correct neighboring components that are subject to its incorrigible behavior. Again, practically speaking, suspicion should be accomplished as soon as possible and maintained for all long as possible. The suspicion should not however be infinitely often in transiently faulty computations; i.e., the suspicion should eventually quiesce.

4.2 Stabilizing Transient Faulty Computations

The difficulty of achieving the first tolerance requirement is that even if all components are correct and their correctors are executed infinitely often, transient state corruption may "cycle" forever. A simple example of corruption persistence is a computation involving two components that are alternately corrupted as a result of receiving a corrupted input from the other and then later their invariant is reestablished as a result of executing their corrector (but not before they have communicated corrupted output to the other).

This difficulty is overcome by allowing the possibility that all components execute their respective correctors at essentially the same time.

Theorem 5 (correctability of invariant violation)

In any transiently perturbed system computation, if each component corrector is executed infinitely often then all component invariants will hold eventually.

Proof of Theorem 5 builds on the proof of Theorem 2. Starting from any system state, in any system computation, with probability 1 eventually there is a "round" where each component executes its corrector and there is no communication between component in the round. At the end of such a round, all component invariants hold.

This proof does not however bound the rate at which invariant stabilization occurs. In fact, by scheduling checks at each component independently of others, it may take a while to execute the desired "round" where implicitly all correctors are synchronized and components do not communications. We therefore discuss a distributed algorithm for expediting the state of invariant stabilization, in Section 4.5.

If we assume that all component invariants are correct in the sense of being sound and complete, we then have

Corollary 5 (stabilization of system invariants)

In any transiently perturbed system computation, if each component corrector is executed infinitely often then eventually all component invariants will hold forever and the computation will not be incorrigible.

In case some component invariants are unsound or incomplete, subsequent computation of those components need not be correct. Nonetheless, should the resulting incorrect behavior have the potential to affect the correctness of rest of the system in an on-going manner, those components will be suspected by their neighbors, as discussed next.

4.3 Suspicion in Incorrigibly Faulty Computations

We cannot rely on components to locally suspect themselves for various reasons:

- An incorrect component's local suspicion does not detect that its outputs are ever incorrect, only that its state invariants are being violated infinitely often.
- An incorrect component's local suspicion is only infinitely often correct and a neighbor may not learn of these suspicions until an indeterminate

later time. Thus, there may be false negative suspicions at the neighbors infinitely often.

- An incorrect component may corrupt the state of its detector/corrector and its local suspicion, which cannot therefore be trusted by its neighbors.
- A correct component's detector/corrector methods may be incorrect, and hence its local suspicion cannot always be trusted.

The tolerance requirement therefore requires each component to additionally perform its own "neighbor suspicion", for each of the neighbors it receives communications from. To this end, we describe a regional suspicion program schema, next.

The regional program schema maintains two types of state variables at each component j.

- m.j.k, whose domain is {0, 1, 2} and denotes component j has marked (i.e. suspected) neighboring component k iff its value is 2. In this case, 2 is an arbitrarily chosen threshold for suspicion, any other strictly positive integer could be chosen if need be. 0 and 1 are respectively the thresholds for no suspicion and partial suspicion.
- s.j, whose domain is {0,1} and denotes the invariants of component j are satisfied iff its value is 0.

The program schema consists of two types of actions, one for detecting local invariant violations and the other for detecting neighbors output violations (via detecting whether those outputs corrupt the local invariants).

Upon detecting corruption in its local invariant, the component's first action executes the local corrector, and increments the self-marking of each component j. Detection of invariant violation implies s.j is set to 1, correction of invariant violation implies s.j is reset to 0, and we define an increment of an m variable whose value is 2 to be 2 itself; hence, the first action at component *j* is:

s.j=1 → *execute_corrector*; s.j := 0 ; m.j.j ++

The second action is superposed upon every underlying action where a communication is received by component j from component k. After the communication is processed by the underlying action, the second action chooses infinitely often to check the invariants of j; if they have been violated it executes the local corrector and increments the marking with respect to k. The choice of when to perform the check respects Angelic Schedulability and may variously depend on the current value of m.j.j (the self-suspicion of j), m.j.k (j's suspicion of k), and m.k.k (the self-suspicion of k as told to j). Let *random_chk* be a Boolean function that takes as input m values and accordingly decides whether or not to schedule a check; hence, the second action at component *j* is:

j≠k →
underlying_action ;
if (*random_chk*(m.j.j, m.j.k, m.k.k, m.k.j))
then {*execute_detector* ;
if s.j=1 then *execute_corrector*, s.j:=0, m.j.k++}

Theorem 6 (suspicion)

In any incorrigible faulty system computation, if the detector of C and neighbors' check of communications from C are scheduled infinitely often, then C's incorrigibility is detected infinitely often by itself and/or its correct neighbors.

Proof of Theorem 6 is essentially the same as Theorem 1. Detection of incorrigibility of *C* by its correct neighbors rests on their ability to possibly schedule an invariant check after receiving a communication from *C* that violates that invariant but before their own subsequent computation inadvertently restores the invariant.

The meaning of the suspicion variable m deserves elaboration. Assume that after every event in a system computation, the relevant suspicion actions are executed correctly. Under this assumption:

- m.j.j marks component j iff j is itself responsible for corrupting its state. In other words, m.j.j marks j iff j is incorrect.
- m.j.k marks component k iff j corrupted its state upon receiving communications from k, in which case either incorrect output of k or incorrect execution at j were responsible. In other words, m.j.k marks k iff k is incorrect or j is incorrect.

Without additional checking, it is in general impossible to further distinguish which of the two components is incorrect.

Of course, in our program schema, suspicion actions are not necessarily executed after every event. Thus, it is in general impossible for the schema to avoid marking j even when j is correct and its neighbors are correct, but state corruption from distant component spreads to j. That said, the rate of executing suspicion actions influences the accuracy with which the m variables indicate (local or neighboring) incorrectness.

4.4 Stabilizing Suspicion of Incorrigible & Transient Faulty Computations

In the same vein as discussed in the previous section, the regional program schema may be extended to ensure that self-suspicion and neighbor suspicion occurs at most finitely in transiently faulty computations.

To this end, two types of actions in the schema suffice: the first lets each component j eventually unsuspect itself starting from any state where it suspects itself (assuming further invariant violations do not occur):

$$m.j.j \neq 0 \ \wedge \ s.j=0 \quad \rightarrow \quad m.j.j - -$$

while the second lets j likewise unsuspect each of its neighbors k:

$$j \neq k \ \wedge \ m.j.k \neq 0 \quad \rightarrow \quad m.j.k - -$$

Our program schema witnesses

Theorem 7 (suspicion stabilization)

There is a correct stabilizing suspicion program that in any system computation, where the detector of each component C and its neighbors' check of communications from C are scheduled infinitely often,

- *unsuspects C eventually always in transiently faulty computation, and*
- *suspects C infinitely often in incorrigibly faulty computations, if C produces incorrigible behavior.*

Note that the rates of executing suspicion and unsuspicion actions influence the rate of false negative suspicions.

Analogous to the argument in the previous section, it is impossible to strengthen the second requirement to suspect *C* eventually always if *C* produces incorrigible behavior.

Theorem 8 (impossibility of stabilizing to no false negative suspicions)

There exists no suspicion program that in any system computation, where the detector of each component C and its neighbors' check of communications from C are scheduled infinitely often, eventually always

- *unsuspects C in transiently faulty computation, and*
- *suspects C in incorrigibly faulty computations, if C produces incorrigible behavior.*

4.5 Quality of System Stabilization

Theorems 5 and 7 characterize the ability of our detector-corrector program schema to stabilize the system state and suspicion, but do not characterize the quality of stabilization. Particular metrics of interest are:

1. Rate of state stabilization;
2. Rate of propagation of state corruption;
3. Frequency of false negative suspicions;
4. Rate of unsuspicion; and
5. Accuracy of marking local incorrectness (versus propagated nonlocal corruption) via suspicion variable m.

Of course, these metrics are interdependent. For instance, the first metric is likely to increase as the second metric decreases. In turn, the second metric is likely to decrease as the third metric decreases; etc.

Quality of stabilization in the program schema is controlled by suitably selecting the execution rates of the four types of stabilization actions in the schema, namely, self checking and invariant correction; communication checking and invariant correction; self unsuspicion; and neighbor unsuspicion.

The execution rate of the schema actions is constrained by the available resources in the system. Our approach is to use all available idle resources for

executing the schema actions. In fact, in some scenarios, execution of actions should even consume resources that would otherwise be used for component/system execution. This is because there is a trade-off between spending resources on detecting and correcting system corruption and spending resources on processing client work requests. The former reduces the number of incorrect outputs from the system, but at the expense of decreasing overall system throughput and potentially the overall rate of correct system throughput.

Given resource constraints, judicious scheduling of the schema actions may be based on available knowledge about the components, assuming there is any. Alternatively, they may be based on estimates of the rate of corruption injection/spreading. In particular, if corruption rates respect some probability distribution or flow model, then estimation techniques (for instance, sampling theory techniques) may be used to determine suspicion and unsuspicion rates.

Scheduling of schema actions in each component may be independent of other components (even though, as mentioned before, each component may choose to schedule based on the m values it receives from its neighbors in communications) or may be dependent.

4.5.1 Fast State Stabilization via Distributed Cleaning

Towards increasing the first metric, we describe a distributed *cleaning* algorithm which imposes some coordination in the execution of schema actions. This algorithm is especially useful in systems where incorrect method executions inject state corruption infrequently.

Recall that the difficulty of achieving state stabilization is that state corruption may "cycle" forever. Although Theorem 5 implies that independent scheduling of schema actions suffices to eventually stabilize the state, its proof places no bound on when the "round" of implicitly synchronized corrections will occur in components whose state is corrupted. It is therefore desirable to explicitly coordinate schema actions to ensure fast state stabilization.

Fast state stabilization is achieved by a two-fold approach:

(1) Avoid the possibility that a component that has checked (and if need be, corrected itself) is re-infected by spreading of corruption, that proceeded in advance of its correction.

(2) Arrest the spreading of corruption; that is, reduce the second metric.

In our cleaning algorithm, each time a "fresh" corruption is detected and corrected, a cleaning wave is spread across the system, piggybacked on component communications with each other. (1) is dealt with by "backpropagation" of the cleaning wave from the receiver of a communication to its sender, and (2) is dealt with by "propagation" of the cleaning wave from the sender of a communication to its receiver. When participating in a cleaning wave, each component *j* checks its invariant, if need be corrects its invariant, and increments its integer count *cw.j.j* of the number of cleaning waves it has participated in.

As can be expected, propagation and backpropagation of cleaning are recursive. Once *j* participates in a new cleaning wave, it subsequently propagates this wave to components it communicates with and backpropagates this wave to components it receives communications from. This process is made efficient by ensuring that each component cleans itself at most once in each cleaning wave as well as in multiple cleaning waves that spread concurrently (should corruptions at multiple components be detected at about the same time). To this end, each component *j* maintains an integer variable cw.j.k for each component *k*, which it receives communications from, that stores the cw.k.k value of the last communication from *k*, and in each communication that it sends to *k*, it sends the current values of cw.j.j and cw.j.k.

More specifically, upon receiving an invocation from *k*, *j* checks whether *k* or *j* have performed a fresh invocation since the last time they communicated. If neither is the case, then *j* simply performs the underlying action associated with the invocation. If only the former is true, i.e. when cw.j.k is less than the copy of cw.k.k that arrives in the invocation, *j* "cleans" itself, i.e. it checks its invariant, and if need be updates its marking, before performing the underlying action. If only the latter is true, i.e. when cw.j.j is greater than the copy of the cw.k.j that arrives in the invocation, *j* sends a backpropagation message to *k*. If both the former and the latter are true, then both *j* and *k* have respectively participated in waves that have not reached the other and there is no need to further propagate/backpropagate these

waves to the other; in this case, *j* simply completes the invocation. (Finally, if the cw values received from *k* are inconsistent with respect to those of *j*, i.e. when cw.j.j is less than the value of cw.k.j received or vice versa cw.j.k is greater than the value of cw.k.k received, then *j* behaves just as it did when it learns that only *k* has performed a fresh cleaning.) Hence, letting "args" denote the method name and argument in the invocation *k* communicates to *j*, this action at component *j* is:

```
j≠k ∧ receive invocation (v.k.k, v.k.j, args) from k →
  if   cw.j.j=v.k.j ∧ cw.j.k=v.k.k→ underlying_action
  elseif
              (cw.j.k<v.k.k ∧ cw.j.j=v.k.j) ∨
      cw.j.j<v.k.j ∨ cw.j.k>v.k.k →  clean;
                                     cw.j.k := v.k.k;
                                     underlying_action
  elseif
     cw.j.j>v.k.j  ∧ cw.j.k=v.k.k → send bkprop(cw.j.j,
                                     cw.j.k, args) to k
  elseif
     cw.j.j>v.k.j  ∧ cw.j.k<v.k.k →  cw.j.k := v.k.k;
                                     underlying_action
  fi
```

where

```
clean ≡
     { cw.j.j++ ;
       execute_detector ;
       if s.j=1 then execute_corrector, s.j:=0, m.j.k++}
```

Also, upon receiving a backpropagation message from *k*, *j* checks to see whether *k* has participated in a cleaning wave that *j* has not participated in but not vice versa. If so, then *j* cleans itself and if it has a pending invocation that it has communicated to *k*, it sends that invocation again (with possibly different arguments due to the cleaning) via *resend_pndng_inv*. If the cw values received from *k* are inconsistent with respect to those of *j*, *j* likewise increments its m.j.k suspicion marking with respect to *j*. In all other cases, it ignores the message. Hence, this action at component *j* is:

```
j≠k ∧ receive bkprop (v.k.k, v.k.j) from k →
  if    cw.j.j=v.k.j ∧ cw.j.k<v.k.k→ clean;
                                     cw.j.k := v.k.k;
                                     resend_pndg _inv
  elseif
      cw.j.j<v.k.j ∨ cw.j.k>v.k.k → m.j.k++;
                                     cw.j.k := v.k.k
  elseif
       true                          →  skip
  fi
```

Initially all the cw.k.j copies in neighbors of *j* are equal to cw.j.j. We assume that each time j executes its local corrector cw.j.j is incremented, whether this is due to the cleaning actions or the schema actions.

Proof of the cleaning algorithm rests on its flooding/gossip nature. A cleaning wave starts when the schema actions cause a correction to occur at *j*. In this case, the wave reaches all system components that receive communications directly or indirectly from *j* or that communicate directly or indirectly to any of these components transitively reached by communications from *j*. If en route, the wave meets another, needless further flooding of the wave is avoided. Backpropagation and re-invocation ensure that corruption does not spread from a component that has not yet received the wave to one that has due to backpropagation. And, provided state perturbations by incorrect methods are rare, corruption cannot spread from a component *k* in one wave to a component *l* in another wave, since both have cleaned their states in their waves and a fresh state corruption has not occurred at *k* or *l*. If state perturbations occur frequently, then the cleaning algorithm may not succeed in restoring the system state to satisfy all component invariant, but the incorrigibility in the computation will lead to appropriate suspicions by the program schema. Finally, transient corruption of the state of the cleaning algorithm is readily self-corrected, since cw.j.k is correctly set to the current value of cw.k.k in every communication the *j* receives from *k*.

5. Distributed Framework for Trusted Base

In this section, we discuss the implementation of a distributed framework that satisfies the **Trusted Base** properties (**Trusted Read**, **Trusted Schedule**, **Trusted Write**, and **Trusted Store**) required for supporting detectors and correctors that enable tolerance to incorrect method faults. Note that the Trusted Base properties are also required in part for

tolerance of the detectors and correctors themselves to incorrect method faults.

Let us first consider implementing a Trusted Base framework in the same machines where system components reside. (By machine, we mean a collection of components whose collective state is not shared with any system component not in that set. Thus, any invocation by a component not in the machine of a method that is in the machine is intuitively a "remote" invocation.) Modern operating systems provide the ability to isolate two activities from each other by means of multiple address spaces. Therefore, assuming that the operating system itself does not become compromised, one can implement the Trusted Base as follows:

- **Trusted Schedule** and **Trusted Store**, which are stateful, are implemented as a program running in its own separate address space.
- **Trusted Read**, **Trusted Write**, and the actual scheduling control aspect of **Trusted Schedule** are implemented as cross-address space OS functions. Note that these functions are essentially identical to those needed by a cross-address space debugger program and hence are commonly available on most operating systems.

Recall that the validity of our theorems rests on the ability to correctly execute at least some tolerance (i.e. detector, corrector, and suspicion) methods infinitely often, regardless of the state of the corresponding components. For the single component theorems we depend on the correct execution of sufficiently many detectors and correctors that incorrect implementations and inconsistencies among them can be overcome or at least reliably flagged. For the distributed case of multiple interacting components we treat arbitrary numbers of misbehaving tolerance methods in one component as just being another version of incorrigibility, however we still depend on mostly correct (and infinitely often) execution of tolerance methods within those components that are not incorrigible.

Since component state may be arbitrarily corrupted by incorrect methods in the component or by the spreading of corruption from other components, we cannot depend on the state of any tolerance methods whose implementation and state reside within a component itself. Thus, we require that the implementations of tolerance methods also reside in the Trusted Base.

This could be achieved by placing the implementations and state of all tolerance methods in the separate Trusted Base address space of each machine. However, the cost of running tolerance methods in this manner would be prohibitive: every read and write operation executed would be a cross-address space operating system call. To alleviate this overhead we can employ the following strategy:

- Run "cached" copies of tolerance methods within the address space of each component. These will run without the overhead of cross-address space overhead and hence can be run relatively frequently.
- Periodically (but still randomly) update the cached copies of tolerance methods from "master" copies that reside within the separate address spaces of the Trusted Base.
- Periodically (but still randomly) run the master copy of every tolerance method in cross-address space fashion.

This approach allows for efficient, but possibly corrupted execution of tolerance methods in the common case, while still providing a guarantee of correct execution eventually. That guarantee comes from the cross-address space execution of tolerance methods. The periodic rejuvenation of local cached copies is merely a means of regaining (possibly temporary) correct execution of the faster cached copies.

In some systems trust in the operating system may not be a reasonable assumption. For example, if some system components reside within the operating system kernel for performance reasons then the assumption of isolated address spaces being available for the implementation of a local Trusted Base component will be invalid. In this case Trusted Base components need to be kept on remote machines in order to achieve the same level of isolation that would otherwise have been provided by using a separate address space.

We must also assume the availability of incorruptible implementations of remote **Trusted Read**, remote **Trusted Write**, and a remote version of the actual scheduling control aspect of **Trusted Schedule**. That is, methods running in remote Trusted Base

components must be able to correctly read from, write to, and schedule the activities of components on a machine without anything on that machine being able to prevent it. This sort of functionality will most likely have to be provided within the (read-only) BIOS of each machine.

6. Related Work

Detection and correction are standard concepts for tolerance to communication errors. In this paper, we have focused on detection and correction of invariant state predicates in the context of software components. A formal definition of state predicate based detectors and correctors is given by Arora and Kulkarni [1]. Their work illustrates and delimits the role of this type of detection and correction in the design of various types of fault-tolerance [1, 2]. In particular, detectors of "safety predicates" are necessary and sufficient for fail-safe tolerance; correctors of "invariant predicates" are necessary and sufficient for nonmasking/self-stabilizing tolerance; and together these detectors and correctors are necessary and sufficient for masking fault-tolerance. Along similar lines, "auditor" programs that periodically check data invariants, and in turn lead to focused diagnosis, are well known to be useful in practice. AT&T, for instance, has long used this detector mechanism in their ESS switch systems.

As mentioned in the introduction, the invariant based approach to incremental correctness is in contrast to and is complementary to approaches based on N-version programming, originally due to Avizenis [3], and recovery blocks, originally due to Randell [14]. It is also in contrast to software rejuvenation due to Huang and Kintala [9], which assumes no knowledge of component invariants, and simply gracefully terminates and restarts a component at a clean state, as one way of proactive compensation for transients.

Regarding our work on suspicion of incorrect component faults and computation incorrigibility, we note that since Chandra and Toueg's work on failure detectors for crash node faults [5], several efforts have focused on designing suspicion programs particular to crash and, in some cases, to crash-repair faults. As was the case in our paper, these efforts are black-box in the sense that they do not rely on particular knowledge about the components. In contrast, Beauquier et al [4] show how to detect transient faults with the assumption that nodes in the system are aware of each other's component programs. Although we believe the ideas of invariant-based or black-box suspicion of incorrect components and computation incorrigibility occur in practice, we find published work in this regard lacking.

Regarding our modeling of incorrect components, we recall that early fault models emerged from work on hardware and communications, including stuck-at faults, message loss, message corruption, message reorder, and timing faults. Subsequent work on data critical and mission critical systems led to adoption of crash [12, 13], fail-stop [14] and Byzantine [10] fault models [9]. In parallel, the model of transient faults was adopted in early work on self-stabilization [7]. Gray [8] observed that most faults in complex computer systems are soft/transient/Heisenbugs, in that they will likely not be repeated if the component is immediately reinitialized and the method is retried. In contrast, a hard fault/Bohrbug is one that requires time to correct, but such faults are likely to be detected and eliminated during structured design, design reviews quality assurance, and testing stages.

7. Summary and Conclusions

7.1 Correctness through Enforcement of Invariants

The fundamental premise of this paper is that systems cannot be made perfectly correct and hence any approach to reliability must include a continually running repair phase. State corruption and bad results to client methods are a fact of life due to the inevitability of incorrectly implemented software and inconsistently specified requirements; problems that are exacerbated whenever systems evolve their functionality over time. Thus the best one can hope to do is to try to minimize incorrect behavior and then try to repair its consequences after the fact. The focus of this paper is on the latter, namely detecting and repairing system state corruption once it has occurred.

An important thing to understand is that we assume that detection and repair of corrupted system state are done in addition to – rather than instead of – any efforts to obtain correct system behavior by other means, such as verification and fault tolerance. This

assumption allows us to consider approximate solutions that need not be perfect. In particular, it allowed us to pursue two additional goals that we feel are highly important for making our approach cheap enough and simple enough to hopefully be of practical value. These goals are incrementality and timing independence.

An important factor in determining whether an approach will be adopted in practice is whether system developers can obtain incremental benefits by putting in incremental amounts of effort. Our approach allows system developers to start out by specifying only the most important aspects that a system's state must obey and then incrementally add additional invariant specifications as time and desire permits. Furthermore, invariants need be added only as appropriate for dealing with difficult parts of the system.

Our invariant-based approach also enables system developers to avoid having to reason about and deal with the complexities of temporal interleaving of software methods. Both evaluation and restoration of an invariant are independent of *how* the system arrived at its current state. Thus, we argue that specifying invariants and providing implementations for the associated detectors and correctors should be significantly simpler than proving the correctness of software methods, whose specification and implementation must take into account whatever concurrency model the system employs.

The simpler nature of timing-independent invariants reinforces the "incremental benefits for incremental effort" aspect of our approach. However, a second question must also be addressed, namely whether one can design and implement system invariants and detector/corrector methods that will yield *desirable* results in addition to simply "correct" results. Because invariants specify only an incomplete definition of what is required for a system to be truly correct, satisfaction of those invariants may not satisfy the users of the system, whose notion of system correctness may differ from that which the system developers were able to think of. Consider, for example, a banking system with an invariant that preserves the total amount of money in the system but says nothing about the history of debits and credits among accounts. The system can be repaired to a state that obeys the invariant but will not reflect the correct allocations of money among individual bank accounts. The bank administration might be happy but the individual customers might not.

A mitigating factor for this problem is that timing-independent invariants need not be oblivious to the execution history of a system. Since we assume the presence of trusted storage for our design, correctors have the option of implementing repair values that are derived from prior system states. Thus a system component could log information in an incorruptible manner that would be useful for restoring its state to useful values. This log of operations and/or data values, while produced in a timing-dependent manner, can still be used by correctors in a timing-independent manner.

Finally, because inconsistencies may exist in the specification of a system's correctness invariants and errors may exist in the implementation of the associated detectors and correctors, one must be prepared to accept the occurrence of irreconcilable invariant violations. When these occur the system will no longer be able to place itself in a correct state and will arguably no longer be able to make forward progress, at least in those components in which the irreconcilable invariant violations are occurring. We argue that this is actually a feature of our approach rather than a problem: Incorrect system components cannot silently continue to diverge from a correct state without bound. That is, running the correctors for a system component will result in its state either being correct or flagged as unusable.

7.2 Detecting and Correcting State Corruption

As mentioned, our approach is reactive rather than preventative in nature. This raises the question of whether corruption can always be detected and corrected. We divided this question into two cases: the case when a finite number of errors are introduced into the system and the case when errors are continually introduced into the system. For the former case we showed that state corruption can always eventually be both detected and corrected. For the latter case we showed that state corruption can always eventually be detected.

The key to always eventually detecting *all* state corruption, whether transiently or repeatedly introduced, is to ensure that it cannot "hide" by

dynamically moving around or being removed and then later reintroduced in suitable ways. To achieve this we required that the system continually run all its detectors and be able to randomly schedule them in a fashion that cannot be predicted by a misbehaving system component.

In order to continually detect the corruptions introduced by incorrigible system components while still being able to eventually declare a transient error case we introduced the suspicion algorithm described in sections 4.3 and 4.4. This algorithm translates repeated individual detections of corruption into suspicion of incorrigibility and prolonged periods of non-detection of state corruption into un-suspicion of incorrigibility; that is, into a presumption that previously seen corruption was due to purely transient errors.

Just as continually running randomly scheduled detectors ensures that state corruption cannot hide, so too will continually running randomly scheduled correctors result in the eventual correction of individual instances of introduced corruption, including any derivative corruption resulting from subsequent infection of other components. However, blindly random scheduling may result in corruption residing within the system for very long periods of time. We also showed how a fairly simple, distributed "cleaning" algorithm can substantially reduce the time required to expunge existing corruption from a system. The algorithm achieves this by creating ever-expanding sets of "cleaned" components, in which any existing corruption has already been detected and corrected. Corrupted components are prevented from re-infecting clean components by forcing them to clean themselves before they are allowed to interact with a cleaned component. Redundant invocations of detectors and correctors are prevented by maintaining a data structure in each component whose size is proportional to the number of other components a given component interacts with.

An important point to realize is that, in practice, infrequently occurring incorrigible errors can be modeled by the transient error case, and hence the state corruption they introduce will be both detectable and repairable. As long as new corruption is not introduced into a system until after existing corruption has been detected and corrected, the system achieves a correct state between each error introduction and is effectively "memory-less" with respect to past corruptions.

Conversely, when invariants do not completely specify the correctness criteria for a system we showed how it is possible for transient error sources to effectively become incorrigible error sources. Since a correct invariant state isn't necessarily a fully correct computation state, the execution of correct methods may still yield incorrect results. Thus, in general, the pragmatic difference between the transient error case and the incorrigible error case may be best thought of as one of degree rather than kind: Transient errors are ones that occur infrequently enough to be easily repairable whereas incorrigible errors are ones that occur frequently enough to prevent effective repair.

Because infrequently occurring incorrigible errors can be modeled as transient errors, we believe that systems that have been reasonably well debugged by conventional means will mostly look like systems that are experiencing transient errors. Thus the practical contribution of our work should be viewed as being an ability to detect and correct state corruption in the common case while being able to at least repeatedly flag state corruption in all cases.

7.3 Controlling the Rate at which Corruption is Searched For

Although our work focuses only on continual detection for the general case, one can still ask the question of what it means to correct corrupted system state in the face of continually introduced new corruption. Since corruption keeps getting reintroduced, all one can hope to achieve is to minimize the fraction of time that system state stays corrupted. For the transient corruption case there is a similar notion of trying to minimize the time that system state stays corrupted before being detected and corrected.

For the transient case we observed that there is a simple algorithm for deciding how often to run detectors and correctors in order to minimize the time that state stays corrupted: Use all available idle resources for randomly and periodically running detectors and run the distributed cleaning algorithm we described after corruption has actually been detected.

Unfortunately, for the continual corruption case when incorrigible system components are present, deciding when and how often to schedule detectors and correctors to run is, in general, quite difficult. Although Section 4.5 described the set of activities for which rates must be determined, one must first model the trade-off between spending resources on detecting and correcting system corruption and spending resources on processing client work requests. The former will reduce the number of incorrect results the system returns, but at the expense of decreasing overall system throughput and potentially the overall rate of correct system throughput.

Tackling this problem represents future work. We speculate that an empirical approach will be needed in which the system will need to define a framework that allows it to receive feedback on allocating resources to detectors and correctors. The feedback will need to include at the very least trends in the number of invariant violations occurring as well as in the overall system throughput. Note, however, that the transient case mentioned earlier, where invariant violations are occurring infrequently and where there are idle resources available, is arguably the common case and hence we believe that a practical version of our approach can be contemplated without having to first solve the general resource allocation problem just outlined.

7.4 Suspecting Components

A negative result presented in this paper is that one cannot expect to reliably distinguish between system components that have experienced transient errors and incorrigible system components that continually inject new errors into the system. That is, one can infinitely often *suspect* a component of being incorrigible, but one cannot declare a component to be incorrigible with certainty. To do so would effectively require that a suspector be able to predict the future.

A similar negative result is that a system component experiencing state corruption can only estimate whether it contains faulty methods or whether it is instead being subjected to bad values from neighboring components that contain faulty methods (or might themselves be instead subject to transitive exposure to faulty behavior from other neighboring components). The fundamental reason for this uncertainty is that our approach involves detecting incorrect system behavior *indirectly*, by means of detecting any state corruption that it produces, either directly or transitively via intermediaries. We do not try to directly detect incorrect output values of bad method invocations since that would effectively boil down to proving those methods to be correct.

Whereas suspicion of software components is the consolation prize we must accept in lieu of being able to actually detect which ones are incorrect, it does provide a tangible benefit beyond just being an input to system developers' debugging activities. As discussed in Section 4.5, suspicion can be used to differentially control the rate at which detectors are run in various system components. By using counting variables a component is able to manage the rate at which detectors are invoked, both within itself and in interacting neighbors, as a function of how suspicious the component is of its own state or the state of its neighbors. This should allow a system to utilize its detection resources more efficiently since detectors will be run more frequently in suspicious components and less frequently in components that have exhibited no recent signs of corruption.

An interesting area of future work is to see whether our suspicion algorithm can be extended to detect various patterns of corruption in order to obtain more precise estimates of which components are actually incorrect. We speculate that various simple patterns of incorrect behavior may be both common and feasible to detect, thereby allowing our suspicion algorithm to be both more efficient and more effective in preventing the spread of corruption, as well as becoming a potential debugging aid.

7.5 Depending on Trusted Software

Since detector/corrector software can itself be incorrect (and the invariants being enforced inconsistent with each other) we had to explain how to ensure system correctness despite this. The key was to require sufficient redundancy among invariants to ensure that inconsistencies could be detected among them when irreconcilable invariant violations occur. The basic idea is very similar to the concept of double-entry bookkeeping in accounting.

To achieve our guarantees we also required the existence of a trusted base of software that ensures

that detectors and correctors can do their job. This included reading and writing of system state, scheduling of detectors and correctors, and storing both internal detector/corrector state and possibly system logging information in a correct, incorruptible manner.

We also showed how this trusted base of software can be structured so that its mutable state can be kept out of the reach of the effects of errant system components while still allowing detectors, correctors, and suspectors to perform efficiently in the common case where their state is not corrupted. This was achieved by placing "master copies" of the relevant state either in different address spaces on the same machine as a system component is running on or on different machines altogether. Cached copies can then be run local to the address space of each system component to obtain good performance while relying on the distributed trusted base to ensure ultimately correct behavior.

7.6 Future Work

In this paper we have assumed that invariants can only describe state properties for individual system components. A natural extension is to explore how to handle invariants whose specification spans several components. This, in turn, will require exploring how to implement detectors and correctors that must be able to access and modify state in multiple, distributed components. For example, whereas in this paper we (mostly) assumed that detectors and correctors are able to run efficiently within the same address space as the system component they are associated with, that assumption does not necessarily hold in a distributed setting. Similarly, we have so far not concerned ourselves with any security implications of our approach. When invariants—and their associated detector and correct methods—only pertain to single system component it is reasonable to assume that they can run with the same privileges as the system component. This assumption again does not obviously hold in the distributed case.

In parallel to extending our work to distributed invariants we are also planning to implement our current design in the context of a real system in order to see how useful it is in practice. One of the authors is involved with a project whose goal is to design and implement a globally scalable event notification service [5]. We plan to augment the implementation of this service with our detector/corrector framework. We hope to achieve the following goals from that effort:

- Understand how easy it is to specify suitable invariants for the service and implement the associated detectors and correctors. "Suitability" will be measured in terms of both ease-of-application as well as ability to actually achieve better system reliability.
- Build the previously sketched framework for resource management and explore how well we can manage the trade-off between employing resources for system correction versus for doing actual service work.
- Explore whether we can do a better job of suspecting individual software components by trying to spot invariant violation patterns.

REFERENCES

[1] A. Arora and S. Kulkarni, "Component Based Design of Multitolerance", *IEEE Transactions on Software Engineering*, Vol. 24, No.1, 63-78, 1998.

[2] A. Arora and S. Kulkarni, "Detectors and correctors: A theory of fault-tolerance components", *International Conference on Distributed Computing Systems* ICDCS'98, Amsterdam, 1998, pp. 436-443.

[3] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, December 1985, pp. 290-300.

[4] J. Beauquier, S. Delaët, S. Dolev, and S.Tixeuil, "Transient Fault Detectors", *12th International Symposium on Distributed Computing* DISC '98, Andros, Greece, September 1998, pp. 62-74.

[5] L.F. Cabrera, M.B. Jones, and M. Theimer, "Herald: Achieving a Global Event Notification Service", *8th Workshop on Hot Topics in Operating Systems* HotOS'01, Elmau, Germany, 2001, pp. 87-94.

[6] T.D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems, *Journal of the ACM* , 43(2), pp 225—267.

[7] E.W. Dijkstra, "Self-stabilizing Systems in Spite of Distributed Control", *Communications of the ACM*, Vol, 17, No. 11, 1974.

[8] J. Gray, "Why Do Computers Stop and What Can We Do About It", *6th International Conference on Reliability and Distributed Databases*, June 1987, pp. 3-12.

[9] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software Rejuvenation : Analysis, Module and

Applications”, *IEEE Intl. Symposium on Fault Tolerant Computing*, FTCS 25, 1995, pp. 381-390.

[10] L. Lamport and N.A. Lynch, “Distributed Computing: Models and Methods”, *Handbook of Theoretical Computer Science*, Vol. 2, Chap. 18, Elsevier Science Publishers, 1990, pp. 1158—1199.

[11] L. Lamport, R. Shostak, and M. Pease, “The Byzantine Generals Problem”, *ACM Transactions on Programming Languages and Systems*, 1982.

[12] B.W. Lampson and H. Sturgis, “Crash Recovery in a Distributed Storage System”, *Xerox Park Technical Report*, Xerox Palo Alto Research Center, 1979.

[13] B.W. Lampson, "Atomic Transactions", *Distributed Systems -- Architecture and Implementation: An Advanced Course*. Chapter 11. Lecture Notes in Computer Science #105, Springer-Verlag, 1981.

[14] B. Randell, “System Structure for Software Fault-Tolerance”, *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, June 1975, pp. 220-232.

[15] R. Schlichting and F. Schneider, “Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems”, *ACM Transactions on Computers*, 1983, pp. 22-238.