Appears in Performance Evaluation Journal, vol. 56, nos. 1-3, March 2004

Improving Availability with Recursive Microreboots: A Soft-State System Case Study

George Candea, James Cutler, Armando Fox
Stanford University
{candea,jwc,fox}@cs.stanford.edu

Abstract

Even after decades of software engineering research, complex computer systems still fail. This paper makes the case for increasing research emphasis on dependability and, specifically, on improving availability by reducing time-to-recover.

All software fails at some point, so systems must be able to recover from failures. Recovery itself can fail too, so systems must know how to intelligently retry their recovery. We present here a recursive approach, in which a minimal subset of components is recovered first; if that does not work, progressively larger subsets are recovered. Our domain of interest is Internet services; these systems experience primarily transient or intermittent failures, that can typically be resolved by rebooting. Conceding that failure-free software will continue eluding us for years to come, we undertake a systematic investigation of fine grain component-level restarts—microreboots—as high availability medicine. Building and maintaining an accurate model of large Internet systems is nearly impossible, due to their scale and constantly evolving nature, so we take an application-generic approach, that relies on empirical observations to manage recovery.

We apply recursive microreboots to Mercury, a COTS-based satellite ground station that is based on an Internet service platform. Mercury has been in successful operation for over 3 years. From our experience with Mercury, we draw design guidelines and lessons for the application of recursive microreboots to other software systems. We also present a set of guidelines for building systems amenable to recursive reboots, known as "crash-only software systems."

1 Introduction

Complex computer systems fail embarassingly often. Numerous studies [3, 47, 75, 27, 74, 91, 4] report that buggy software is a main source of unplanned downtime in large-scale computing infrastructures. In spite of sophisticated development processes and tools, all production-quality software still has bugs; most of these are difficult to track down and resolve, or else development and testing would have fixed them [14]. When such bugs strike, they often result in prolonged outages [47, 74].

Fault-free software of reasonable complexity will continue eluding us in the foreseable future. Computer engineers and researchers have traditionally relied on fault avoidance techniques, system validation, and analytical models to try and build fault-free software. These decades of experience have led to numerous valuable techniques [67, 84, 93], which have improved the reliability of our systems. But these same efforts have also demonstrated that accurate modeling and/or verification of complex computer systems and their environment is impractical in most cases. Most other branches of engineering build and maintain systems that are subject to the laws of physics (buildings, integrated circuits, chemical processes, etc.); these laws can be used to model the systems. Software has only an abstract embodiment, and is thus governed solely by laws laid down, sometimes unwittingly, by its designers and implementors. The ability to make mistakes is, therefore, unbounded.

The True Cost of Performance

The focus of computer systems researchers and developers for the last few decades has been on increasing performance, and that single-minded effort has yielded four orders of magnitude improvement [52]. Not surprisingly, this single-minded focus on performance has neglected other important aspects of computing: availability, security, privacy, and total cost of ownership, to name a few. For example, total cost of ownership is widely reported to be 5 to 10 times the purchase price of hardware and software, and [42] suggests that a third to a half of it goes toward failure recovery and planning. Despite marketing campaigns promising 99.999% availability, well-managed servers today achieve 99.9% to 99%, or 8 to 80 hours of downtime per year; each such hour can cost from $200,000 for an Internet service like Amazon.com to $6,000,000 for a stock brokerage firm [61].

In software, more so than in hardware, higher performance requires tradeoffs that often make programs more fragile. For example, why is it not safe to shut down a workstation by just flipping its power switch? Often the reason is performance tradeoffs made in the filesystem and other parts of the software. To avoid synchronous disk writes, many operating systems cache metadata updates in memory and, when power is lost, those updates are lost as well, leaving the file system in an inconsistent state. This usually requires a lengthy `fsck` or `chkdsk` to repair, an inconvenience that could have been avoided by shutting down cleanly. The designers of such operating systems traded data safety and recovery performance for improved steady state performance. Not only do such performance tradeoffs impact robustness, but they also lead to complexity by introducing multiple ways to manipulate state, more code, and more APIs. The code becomes harder to maintain and offers the potential for more bugs—a fine tradeoff, if the goal is to build fast systems, but a bad idea if the goal is to build highly available systems.

If the cost of such performance enhancements is lower dependability and longer downtimes, then perhaps we should leave further performance improvement to Moore's Law, and reevaluate our design strategies.

Recovery-Oriented Computing

"If a problem has no solution, it may not be a problem, but a fact, not to be solved, but to be coped with over time" said Shimon Peres, 1994 Nobel Peace Prize laureate. This quote has become the mantra of the Recovery-Oriented Computing (ROC) project [82], in which we consider crashes, hangs, operator errors, and hardware malfunctions to be facts, and the way we cope with these inevitable failures is through fast recovery. The ROC hypothesis is that, in the 21st century, recovery performance would be a more fruitful pursuit and more important for society than traditional performance. In the absence of perfect software, we might hope that the number of software bugs will at least asymptotically approach zero. Unfortunately, as seen in recent studies [26], the rate at which the number of bugs per thousand lines of code (bugs/Kloc) is reduced through tools, language features, and programmer training appears to be far outpaced by the rate at which software size increases (Kloc/software product). The total number of bugs goes up, and there is no reason to believe this will change. Software-induced system failures therefore become inevitable; to be effective in coping with them, we must devise techniques for rapidly and effectively recovering from their failure.

This approach suggested ROC's prefered metric: mean time to recover. A widely accepted equation for availability is A=MTTF/(MTTF +MTTR), where MTTF is the mean time to fail of a system or subsystem (i.e., the reciprocal of reliability), MTTR is its mean time to recover, and A is a number between 0 and 1. The equation suggests that downtime, or unavailability, is U=MTTR/(MTTF +MTTR), which can be approximated by MTTR/MTTF when MTTF is much larger than MTTR [1]. Thus, to reduce downtime by a factor of 10, a tenfold decrease in MTTR is just as valuable as a tenfold increase in MTTF. We therefor make a case for adopting MTTR as the primary metric for reasoning about system availability and focusing designs on fast recovery.

In the case of hardware, today's component MTTF's are so high that directly measuring them requires many system-years of operation; most customers cannot afford this and must largely rely on vendor claims to assess the impact of MTTF on availability. Verification is further complicated because hardware manufacturers exclude operator error and environmental failures from their calculations, even though they account for 7-28% of all unplanned downtime in some cluster and mainframe installations [94] and more than half of unplanned downtime in a selection of contemporary Internet services [80]. On the other hand, MTTR can be directly measured, making MTTR claims independently verifiable. In the case of software, for example, MTTF's are on the order of days or months, while MTTR varies from minutes to hours.

For end-user interactive services, such as Web sites, lowering MTTR can directly affect the user experience of an outage.

[1]Today's Internet systems achieve availabilities between 0.99 and 0.999, meaning that service MTTF is 2 to 3 orders of magnitude greater than service MTTR

In particular, reducing MTTR can be shown to reduce the impact, and therefore the cost, of a specific outage, especially when redundancy and failover are used to mask the failure. In contrast, increasing MTTF may reduce the frequency of failures (and therefore the probability that a given user will experience a failure during her session), but does not capture the impact of a particular outage on the user experience or cost to the service provider. [106] found that, when end-user behavior is considered, sites with lower MTTR and lower MTTF are perceived by users as more available than sites having the same availability A but higher MTTR.

Progress in improving performance was rapid in part because there was a common yardstick—benchmarks—by which success could be measured. To make similar progress on recovery, we need similar incentives, that measure progress and reward the winners. If we embrace availability and MTTR, systems of the future may compete on recovery performance rather than just SPEC performance, and such a change may improve the feelings that end users have toward the computer infrastructures they depend on. By focusing this discussion on lowering MTTR, we are not advocating that less effort be spent on debugging or operations, nor do we claim that all hard failures can be masked through redundancy to achieve lower MTTR. Nonetheless, the above observations—measurability and relevance of MTTR as an availability metric, and the ability to exploit low MTTR to mitigate the effects of partial failures—suggest that, as a community, we should more aggressively pursue opportunities for improvement based on design for fast recovery: it has direct correlations to user satisfaction, and is benchmarkable to boot.

The rest of this paper describes a recursive approach to recovering large systems (section 2), followed by a case study in which we applied microreboots to a recursively recoverable satellite ground station (section 3). We present guidelines for building software systems amenable to recursive reboots, known as "crash-only systems," in section 4. Section 5 presents related work, and section 6 concludes.

2 Recursive Recovery and Microreboots

Most engineering disciplines are governed by the laws of physics, which make many processes irreversible and unrecoverable. The creations of software engineering, however, lack physical embodiment. This is a two-edged sword: while the lack of such laws makes software more chaotic, it also allows us to do things that are impossible in the physical world. For example, California civil engineers have no choice but to design buildings as durable as they can, to maximize the chance of surviving an earthquake. In the virtual world of software, however, it may be just as effective to let a building crumble and then replace it milliseconds later, if this happens fast enough that nobody notices. As software complexity increases, the cost of chasing and resolving elusive bugs goes up, and bugs multiply further, making it difficult to prevent failure.

It is common for bugs to cause a system to crash, deadlock, spin in an infinite loop, livelock, or to develop such severe state corruption (memory leaks, dangling pointers, damaged heap) that the only high-confidence way of continuing is to restart the process or reboot the system [12, 35, 86, 78]. In fact, [91] estimates that 60% of unplanned downtime in business environments is due to application failures and hardware faults, of which 80% are transient [27, 72], hence resolvable through reboot. From among the variety of recovery mechanisms available, we are most interested in reboots, because, for properly designed software, they: (a) unequivocally return the recovered system to its start state, which is the best understood and best tested state of all; (b) provide a high confidence way to reclaim resources that are stale or leaked, such as memory and file descriptors; and (c) are easy to understand and employ, which makes reboots easy to implement, debug, and automate.

Starting from this observation, we argue that in an appropriately designed system, we can improve overall system availability through a combination of reactively restarting failed components (revival) and prophylactically restarting functioning components (rejuvenation) to prevent state degradation that may lead to unscheduled downtime. We define a framework that uses recursive microreboots to recover a minimal subset of a system's components and, if that doesn't help, recursively recover progressively larger subsets. The microreboot is a low-cost form of reboot that is applied at the level of individual fine grained software components. Recursive microreboots provide a way to deal with some of the drawbacks of using inexpensive COTS software, particularly after deployment. Rebooting can be applied at various levels: Deadlock resolution in commercial database systems is typically implemented by killing and restarting a deadlocked thread in hopes of avoiding a repeat deadlock [45]. Major Internet portals routinely kill and restart their web server processes after waiting for them to quiesce, in order to deal with known memory leaks that build up quickly under heavy load. A major search engine periodically performs rolling reboots of all nodes in their search engine cluster [11].

For a system to be recursively recoverable (RR), it must consist of fine grain components that are independently recoverable, such that part of the system can be repaired without touching the rest. This requires components to be loosely coupled and be prepared to be denied service from other components that may be in the process of microrebooting. We are encouraged by the increasing popularity of technologies that enable loosely coupled, componentized systems, such as Sun J2EE [97] and

Microsoft .NET [71]. A recursively rebootable system gracefully tolerates successive restarts at multiple levels, in the sense that it does not lose or corrupt data, does not cause other systems to crash, etc. Due to its fine restart granularity, an RR system enables bounded microreboots that recover a failed system faster than a full reboot.

2.1 No A Priori Models

Diverging from traditional reliability research, we regard the computer system as an entity that cannot be understood in its entirety, and rely on empirical observations to keep it running. Similar approaches have been used in the study of the Internet [36, 83, 89], the most complex computer system built to date. Based on observations of the target system's behavior, we infer its internal fault domains [17], as well as some of the unintended ways in which faults propagate across the perimeters of these domains. Modeling systems is particularly unsuccessful in large scale Internet services, that are built from many heterogenous, individually packaged components, and whose workloads can vary drastically. Such systems are often subject to rapid and perpetual evolution, which makes it impossible to build a model that stays consistent over time. Due to large scale and high availability requirements, fixing software bugs once the software has been deployed is difficult and expensive. One major advantage of Internet-like services, though, is that workloads consist of large numbers of relatively short tasks rather than long-running operations; this makes the unit of lost work very small, thus reducing the overall impact of transient failures.

Most software can be made recursively rebootable at the cost of performance. We do recognize that in some cases the price of recursive reboots may be unacceptably high, so our framework accomodates non-reboot-based recovery methods as well. Such methods can be defined in a component-specific way and can therefore accommodate heterogeneous recursive recovery.

2.2 Recursively Recoverable Systems

Systems in our chosen application domain often have the structure shown in Figure 1. The execution infrastructure, be it an operating system, an application server, or a Java virtual machine, provides a set of services (shown as `SVC` bubbles) to applications. The applications themselves are built from components (shown as `COM` bubbles) which interact with each other and with the infrastructure services. The components can be any of a variety of types: threads, processes, Java beans, .NET services, etc. To enable recursive recovery, we impose restrictions on the components (described in section 4) and augment the execution infrastructure with a recovery manager, monitoring agents, and recovery agents that effect the recovery per se.

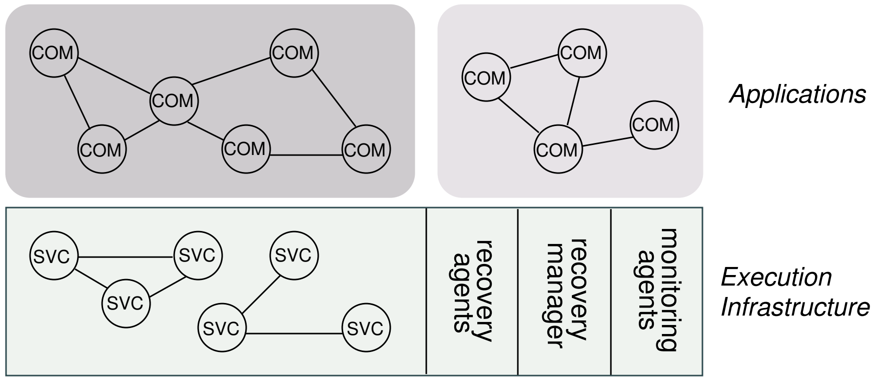


Figure 1. A typical recursively recoverable architecture

The smallest unit that can be recovered in the system (`COM` and `SVC` bubbles) is called an r-unit, regardless of whether it is a component in the application or the execution infrastructure. The monitoring agents are in charge of supervising the health of the system and reporting interesting changes to the recovery manager. Based on the inferred knowledge of the system and its history, the manager decides which of the r-units need to be recovered. These decisions are handed to the recovery agents for execution, who are the only ones who know how specific r-units can be recovered. In order for the recovery system to scale, we must localize the scope of recovery, in addition to having good fault isolation and rapid failure detection.

In the rest of this section we provide more detail on the various aspects of recursive recovery. When a failure is detected in the system (section 2.2.1), recovery for the faulty component(s) is triggered. Should the recovery not cure the observed failure, we assume it is because the fault has contaminated other parts of the system, and we progressively enlarge the recovery perimeter to the larger, containing subsystem. The recovery attempts continue with increasingly larger scope, chasing the fault recursively through successively broader fault domains until the failure stops manifesting or until human intervention is deemed necessary. The recovery agents needs a map to be able to navigate the system's fault boundaries during recovery; section 2.2.2 describes how we obtain this map. The procedure for recovering a component and its dependent peers is described in section 2.2.3.

2.2.1 Monitoring System Health

In designing our monitoring framework, we employ multiple lines of defense. Any one detection technique cannot find all faults, so having multiple layers reduces the chances of a fault going undetected. Moreover, fault detection software itself will have bugs, and a combination of alternate techniques can help compensate for failures in fault detection. We monitor the system at three layers: platform, application, and end-to-end:

- Platform-level monitoring uses generic knowledge of component behavior. An operating system kernel can detect segmentation violations or watch a process' memory and I/O activity; a Java VM can inspect the application's use of synchronization primitives. Placing timeouts on all communication enables the detection of remote failures, and using ICMP pings can reveal network partitions. Monitoring application activity in terms of I/O and communication can also help infer progress [19].

- Application-level monitoring uses a combination of application progress counters and behavior monitors. The counters are component-implemented methods that translate application-specific progress into a universal number (e.g., a distributed transaction manager engaged in a 2-phase-commit could indicate how many participants are in the "prepared" phase as a way to measure the commit progress); a component that is not making any progress for a long time may indicate a failure. Behavior monitoring uses deviations from pre-agreed behaviors to infer failure (e.g., if nodes in a cluster agree to issue periodic heartbeats and the monitor does not hear from one of them for a long time, the node must have either failed or fallen behind a network partition).

- End-to-end monitoring exploits the application's end-user-visible interface to verify its liveness. For example, performing an SQL query on a database and verifying the result provides reasonable confidence that the database system as a whole is OK. Similarly, doing an HTTP GET operation on a known URL can confirm that a web front-end is up and running.

Typically, there are orders of magnitude differences in the amount of code exercised by each layer of monitoring and failure detection, so each type of monitoring has a different level of invasiveness and performance impact on the monitored system. Consequently, these techniques are performed with different frequencies: platform-level checks are the least expensive and are performed more often than application-level checks, which in turn are performed more often than end-to-end checks.

The monitoring agents convey noteworthy changes in system health to the recovery manager. Such changes include not only failure information, but also the appearance or disappearance of components, such as when a new application or service is deployed or undeployed. Some of the monitors may be faulty or provide incomplete information, so the recovery manager must be able to corroborate failure information and form an "opinion" about the state of the system. Since the monitors, recovery manager, and recovery agents represent single points of failure, we have simple mutual supervision arrangements that allow any of the three modules to detect the others' failure and recover them. A specific case will be described in section 3.3.

2.2.2 Fault Propagation and Recovery Maps

When the recovery manager receives a failure notification from the monitoring agents, it is responsible for making recovery decisions. To aid in these decisions, the manager maintains a dynamic view of the system that captures the currently known paths along which faults can propagate. This view can be restructured for optimal recovery performance, as will be described in section 3.5.

The system view is captured in an f-map—a graph that has components as nodes and direct fault-propagation paths as edges. Given that the recovery manager has no a priori knowledge of the layout of the application or system it is supposed to manage, nor of what components form the system or how they interact, we have devised a technique for automatic failure-path inference (AFPI) [17]. AFPI consists of two phases: in the (invasive) staging phase, the recovery manager actively performs both single-point and correlated fault injections, observes the system's reaction to the faults, and builds the "first draft" of the f-map; in the (non-invasive) production phase, the system passively observes fault propagation when such faults occur during normal operations, and uses this information to refine and evolve the f-map on an ongoing basis. In both phases, the monitors report to the recovery manager the path taken by faults through the system, and the manager adds the corresponding edges to the f-map. If components are added or removed from the system for upgrade or reconfiguration reasons, the recovery manager is notified and automatically removes/adds the corresponding nodes to the f-map. The passive observation phase works fine even without the initial active phase, but can take much longer to converge onto a correct representation of the failure dependencies.

Based on the f-map, the manager constructs a recovery map that describes the direction in which recovery perimeters should be enlarged during successive recursive recovery attempts. Unfortunately, the f-map can contain cycles reflecting mutual fault-dependence between sets of nodes; such cycles must be treated as single units of recovery. The recovery manager computes all connected components and collapses these subgraphs into single nodes; the resulting graph is acyclic and serves as the recovery map (r-map).

2.2.3 The Recovery Process

We define the recovery group of a given r-unit as the set of nodes in the r-map that are reachable from that r-unit, essentially representing all the nodes that could be contaminated by a fault in the r-unit. When the recovery manager decides to recover a given r-unit, it actually recovers that r-unit's entire group. If we think of an r-unit as an object that has `pre-recovery()` and `post-recovery()` methods, then Figure 2 describes the recursive recovery of an r-unit r. Prior to recovering any of the downstream[2] r-units, the `pre-recovery()` method prepares r for recovery; the downstream r-units are recovered recursively, and then the `post-recovery()` method finalizes recovery.

```
recover(r)
    invoke r.pre-recovery()
    for each r-unit r_i immediately downstream from r
        invoke recover(r_i)
    invoke r.post-recovery()
```

Figure 2. Generic recovery of an r-unit.

If a number of monitoring agents indicate the failure is persisting even after `recover(r)` has completed, the recovery manager concludes that the failure must have propagated to r from one of its upstream neighbors, across a fault boundary. Recovery is thus propagated in the reverse direction of failure propagation, by invoking `recover()` on all the r-units immediately upstream from r. This recursive process is repeated until the failure has been eliminated, or until a failure is found that requires human intervention (e.g., we just rebooted the entire system and the problem did not go away).

The `pre-recovery()` and `post-recovery()` methods provide a general framework for defining per-component recovery procedures. RR can simultaneously support different types of recovery in the same system: microreboots for crash-only components (described in section 4), in which case the pre-recovery and post-recovery procedures are empty, checkpoint restoration for stateful components, log-based rollback for transactional components, etc. Moreover, hybrid recovery strategies are possible as well: one might choose to roll back prior to recovering downstream r-units, and then restart the component once the downstream recovery has completed.

Based on the r-map and on the monitoring information, the recovery manager has the ability to not only make reactive recovery decisions but proactive preventive maintenance decisions as well. Software rejuvenation [56, 39] has been shown to be a useful technique for staving off failure in systems that are prone to aging; for instance, rebooting several times a day Apache web servers that leak memory is an effective way to prevent them from failing [13]. The recovery manager in a recursively recoverable system tracks components' failure histories and infers for how long a component can be expected to run without failing due to age; restarting it before that time runs out will avert aging-related failure. The observation of fail-stutter behavior [8] can also trigger rejuvenation. A number of sophisticated models have been developed for the software aging process [41, 40], but experience with deployed large scale Internet services seems to indicate that simple observation-based strategies work best [13].

[2]In a directed acyclic graph, if there exists a directed path from vertex A to vertex B, then we say A is upstream from B, and B is downstream from A. If the graph contains the directed edge (A, B), then vertex B is *immediately* downstream from A, and A is immediately upstream from B.

3 Case Study: The Mercury Ground Station

To illustrate recursive recovery, we present in this section a case study of applying recursive microreboots to a Java-based software system. A number of this system's properties make it particularly amenable to a restart-based failure management regimen. The emphasis in this presentation will be on transformations that can be applied to the system's recovery map to minimize MTTR of the overall system.

3.1 Overview

The Recovery-Oriented Computing group (ROC) at Stanford and the Space Systems Development Lab (SSDL) are collaborating on the design and deployment of space communications infrastructure to make collection of satellite-gathered science data less expensive and more reliable. One necessary element of satellite operations is a ground station, a fixed installation that includes tracking antennas, radio communication equipment, orbit prediction calculators, and other control software. When a satellite appears in the patch of sky whose angle is subtended by the antenna, the ground station collects telemetry and data from the satellite. In keeping with the strong movement in the aerospace research community to design ground stations around COTS (commercial off-the-shelf) technology [70], part of the collaboration between SSDL and ROC includes the design and deployment of Mercury, a prototype ground station that integrates COTS components.

A current goal in the design and deployment of Mercury is to improve ground station availability, as it was not originally designed with high availability in mind. Our first step in improving the availability of Mercury was to apply recursive reboots [18] to "cure" transient failures by restarting suitably chosen subsystems, such that overall mean-time-to-recover (MTTR) is minimized.

We had two main goals in applying RR to Mercury. The first was to partially remove the human from the loop in ground station control by automating recovery from common transient failures we had observed and knew to be curable through full or microreboots. In particular, although all such failures are curable through a brute force reboot of the entire system, we sought a strategy with lower MTTR. The second goal was to identify design guidelines and lessons for the systematic future application of RR to other systems. For example, we found that, if one adopts a transient-recovery strategy based on partial restarts, redrawing the boundaries of software components based on their MTTF and MTTR can minimize overall system MTTR by enabling the tuning of which components are rebooted together. In contrast, most current system and software engineering approaches establish software component boundaries based solely on considerations such as amount and performance overhead of communication between components or amount and granularity of state sharing.

3.2 Ground Station Architecture

The Mercury ground station communicates with low earth orbit satellites at data speeds up to 38.4 kbps. For the past three years, the Mercury system has been used in 10-20 satellite passes per week as a primary communication station for Stanford's satellites Opal [31] and Sapphire [98].

Mercury's design has drawn heavily on the lessons of Internet technologies, resulting in an architecture that is novel in the space systems community [30]. The station, composed primarily of COTS hardware and software written mostly in Java, is controlled both remotely and locally via a high-level, XML-based command language. Software components are independently operating processes with autonomous loci of control and interoperate through passing of messages composed in our XML command language. Messages are exchanged over a TCP/IP-based software messaging bus.

The general software architecture is shown in Figure 3: fedrcom is a bidirectional proxy between XML command messages and low-level radio commands; ses (satellite estimator) calculates satellite position, radio frequencies, and antenna pointing angles; str (satellite tracker) points antennas to track a satellite during a pass; rtu (radio tuner) tunes the radios during a satellite pass; mbus passes XML-based high-level command messages between software components[3]. REC and MON will be described in the next section.

The ground station components are safe to reboot, since they do not maintain persistent state; they use only the state explicitly encapsulated by received messages from mbus. Hard state exists in Mercury, but is read-only during a satellite pass and is modified off-line by ground station users. In addition, the set of Mercury failures that can be successfully cured by reboot is large, and in fact this is how human operators recovered from most Mercury failures before we implemented automated recovery.

[3]There are a few other components in Mercury, but, for the sake of simplifying the presentation, we do not describe them in this paper.

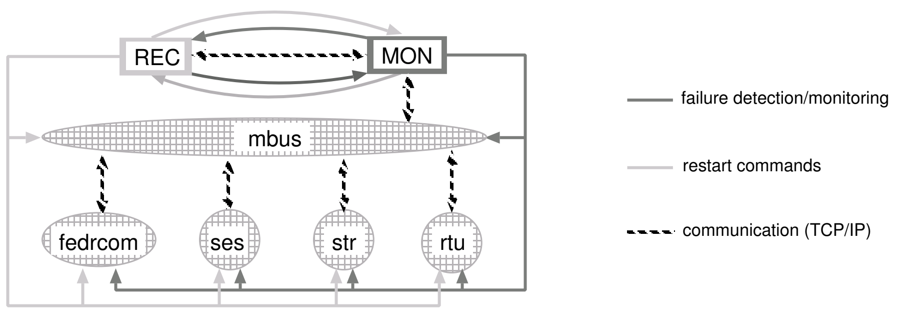


Figure 3. Mercury software architecture

Mercury is a soft-state system, in that any writeable state is constantly refreshed by messages, and state which is not refreshed eventually expires [85]. Soft state and announce/listen protocols have been extensively used at the network level [107, 34] as well as the application level [37]. Announce/listen makes the default assumption that a component is unavailable unless it says otherwise; soft state can provide information that will carry a system through a transient failure of the authoritative data source for that state. The use of announce/listen with soft state allows restarts and "cold starts" to be treated as one and the same, using the same code path. Moreover, complex recovery code is no longer required, thus reducing the potential for latent bugs and speeding up recovery. In soft-state systems, reboots are guaranteed to bring the system back to its start state; by definition, no data corruption is possible.

Unfortunately, sometimes soft state systems cannot always react quickly enough to deliver service within their specified time frame. Use of soft state implies tolerance of some state inconsistency, and sometimes the state may never stabilize. This type of problem can generally be addressed by increasing refresh frequency, albeit with additional bandwidth and processing overhead. The benefit of having loosely coupled components, however, makes the tradeoff worthwhile.

Two salient properties of Mercury distinguish it form larger scale Internet applications. First, this is a static system that does not need to evolve online; it can be upgraded and reconfigured inbetween satellite passes. Second, there are no circular functional dependencies between components, and in particular, its fault propagation and recovery maps are very simple and have a tree structure. We will call Mercury's tree-like recovery map a "reboot tree" from here on.

3.3 Adding Failure Monitoring to Mercury

Adding failure monitoring and detection to this architecture was motivated by the need to automate detection of several common failure modes; we understood these modes from extensive past experience with Mercury. All the components we focused on were fail-silent: when they failed, they simply stopped responding to messages, such as when the JVM containing a component crashes. Moreover, all failures were curable through restart of either a single software component or a group of such components.

Given the fail-silent property, we chose application-level liveness pings (i.e., "are you alive?" messages) sent to a component via the software message bus, mbus. The pings are encoded in and replied to in a high-level XML command language, so a successful response indicates the component's liveness with higher confidence than a network-level ICMP ping. Application-level liveness pings are simple and low-cost, and effectively detect all fail-silent failures that humans were detecting before in the ground station, thus satisfying the immediate goal of automated failure detection.

Figure 3 illustrates Mercury's simple failure detection architecture, based on the addition of two new independent processes: a failure monitor (MON) and a recovery module (REC), which logically colocates the recovery manager and recovery agents. This colocation is made possible by the static nature of this system; otherwise, monitoring agents would have to be easily interchangeable. MON continuously performs liveness pings on Mercury components, with a period of 1 second, determined from operational experience to minimize detection time without overloading mbus. When MON detects a failure, it tells REC which component(s) appear to have failed, and continues its failure detection. For improved isolation, MON and REC communicate over a separate dedicated TCP connection, not over mbus; mbus itself is monitored as well. REC uses a reboot tree data structure and a simple policy to choose which module(s) to restart upon being notified of a failure. The policy also keeps track of past microreboots to prevent infinite reboots due to "hard" failures. Once REC reboots the chosen modules, future application-level pings issued from MON should indicate the failed components are alive and functioning again. If the microreboot does not cure the failure, MON will redetect it and notify REC, which may choose to reboot a different module this time, and so on.

Given the above strategy, two situations can arise, which we handle with special case code. First, MON may fail, so we wrote REC to issue liveness pings to MON and detect its failure, after which it can initiate MON recovery. Second, REC may go down, in which case MON detects the failure and initiates REC's recovery, although the generalized procedural knowledge for how to choose the modules to microreboot and initiate recovery is only in REC.

Splitting MON and REC requires the above two cases to be handled separately, but it results in a separation of concerns between the modules and eliminates a potential single point of failure. Our enhanced ground station can tolerate any single and most multiple software failures, with the exception of MON and REC failing together.

It is important to note that, in our system, microreboots are a recovery mechanism based on detecting black box failures, not low level faults. The response to a failure (i.e., microrebooting) is independent of the fault that caused the failure. This is in keeping with our desire to manage the system with minimal information about its internals. microreboots can be used in addition to other recovery strategies, not necessarily in place of them, so we do not believe that anything we have done precludes the use of more sophisticated failure detection or high availability mechanisms in the future.

3.4 The Reboot Tree

To explain the meaning of a reboot tree, we show in Figure 4 a simple transformed recovery map with 3 r-units, A, B, and C. There are two recovery agents, R_{BC} and R_{ABC}, which have a conceptual "button" that can be "pushed" to cause the restart of the entire subtree rooted at that node. The reboot tree in Figure 4 captures the fact that B and C must be microrebooted together, and that, if A is rebooted, so must B and C (pressing R_{ABC}'s reset button will automatically press R_{BC}'s button as well). This representation of the reboot policy is only possible because we are dealing with a non-evolving system.

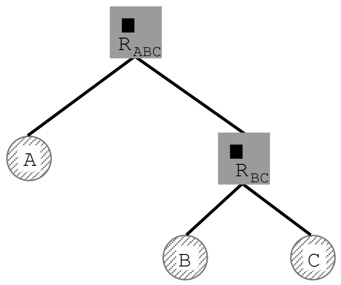


Figure 4. A simple reboot tree.

The techniques we will describe in this section for constructing and evolving reboot trees are based on the assumption that MTTF and MTTR represent the means of distributions with small coefficients of variation. We have confirmed through experiment that this is the case with our system, and for compactness we will henceforth use the notations MTTF_S and MTTR_S to refer to the MTTF and MTTR, respectively, of subsystem S. In particular, we assert that the MTTF for a restart group G containing components $c_0, c_1, \ldots, c_n$ is $\text{MTTF}_G \leq \min(\text{MTTF}_{c_i})$, and that the corresponding MTTR is $\text{MTTR}_G \geq \max(\text{MTTR}_{c_i})$.

Based on information about which component has failed, the recovery manager decides which recovery agent(s)' reset buttons to press. If a reboot at that point fixes the problem, then the system continues operation normally. However, if the failure still manifests (or another failure appears) even after the restart completes, the recovery manager moves up the tree and requests the restart of the previously-reset agent's parent. This process can be repeated up to the very top, when the entire system is rebooted.

We say a failure is n-curable if it is cured by a restart at node n or any of its ancestors in the reboot tree. A *minimally n-curable* failure is a failure that is n-curable and n is the *lowest* node in the tree for which a restart will cure the failure. Admitting that mean-time-to-repair is non-decreasing as we move up the tree, a minimal cure implies the failure is resolved with minimal downtime. For a given failure, it is possible for n to not be unique (e.g., if restarting the parent of n is no more expensive than restarting n itself). A perfect recovery manager is expected to embody the *minimal restart policy*, i.e., for every minimally n-curable failure, it recommends a microreboot of node n. In section 3.5.4 we illustrate what happens when the manager is imperfect.

3.5 Evolving Mercury's Reboot Tree

We show on the left side of Figure 5 a simple reboot tree for Mercury (tree I), consisting of a single recovery group. The only possible policy with this tree is to reboot all of Mercury when something goes wrong. The system MTTF is at least as bad as the lowest MTTF of any component, and its MTTR at least as bad as the highest MTTR of any component. Table 1 shows rough estimates of component failure rates, made by the administrators who have operated the ground station for the past three years. The components that interact with hardware are particularly prone to failure, because they do not fully handle the wide variety of corner cases.

Component	mbus	fedrcom	ses	str	rtu
MTTF	1 month	10 min	5 hr	5 hr	5 hr

Table 1. Observed per-component MTTF's.

In Mercury, each software component is failure-isolated in its own Java virtual machine (JVM) process, a failure in any component does not necessarily result in failures in others, and a reboot of one component does not necessarily entail the reboot of others. This suggests the opportunity to exploit microreboots to lower MTTR. Set against this opportunity is the reality that (a) some failures do propagate across JVM boundaries, and (b) microrebooting some components can cause other components to need a microreboot as well; both result in observed correlated failures. In the former case, a state dependency leads to a restart dependency; in the latter case, a functional dependency leads to a restart dependency. In the rest of this section we describe how to modify the trivial reboot tree to reduce the MTTR of the overall system, illustrating which tree modifications are most effective under specific conditions of correlated failures.

We describe three techniques: *depth augmentation*, that results in the addition of new nodes to the tree, and *group consolidation* and *node promotion*, that result in the removal of nodes from the tree. Since the focus of the present work is to investigate a recovery strategy designed for transient failures, we make the following simplifying assumption, that does hold for our system:

$\mathcal{A}_{\text{cure}}$: *All failures that occur are detectable by* MON *and curable through restart.*

This assumption is consistent with the fail-silent and reboot properties of our system's components.

Another assumption, $\mathcal{A}_{\text{entire}}$, arises when there is no functional redundancy in the system; it would not necessarily apply, for example, to a cluster-based Internet server with hot standby nodes or similar functional redundancy:

$\mathcal{A}_{\text{entire}}$: *A failure in any component will result in temporary unavailability of the entire system.*

3.5.1 Simple Depth Augmentation

A failure in any component of tree I will result in a maximum-duration recovery. For example, rtu takes less than 6 seconds to restart, whereas fedrcom takes over 21 seconds. Whenever rtu fails, we would need to restart the entire system and wait for all components, including fedrcom, to come back up, hence incurring four times longer downtime than necessary. In this argument we implicitly assume that components can restart concurrently, without significantly affecting each other's time-to-recover.

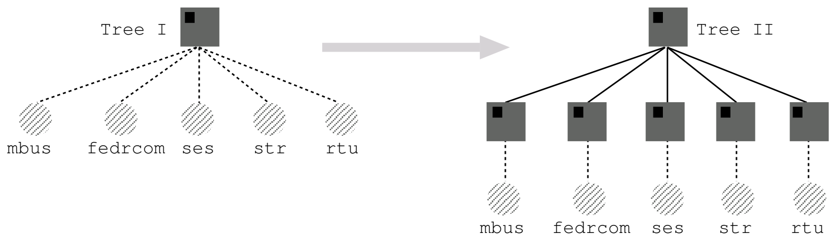


Figure 5. Simple depth augmentation gives tree II.

The "total reboot" shortcoming can be fixed by modifying the tree to allow for microreboots, which can cure subsystems containing one or more components without bringing the entire system down. Figure 5 illustrates this transformation.

To measure the effect this transformation has on system recovery time, we force the failure of each component, using a `SIGKILL` signal, and measure how long the system takes to recover. We log the time when the signal is sent; once the component determines it is functionally ready, it logs a timestamped message. The difference between these two times is what we consider to be the recovery time. Table 2 shows the results of 100 experiments for each failed component.

In the new reboot tree `II`, each recovery group, except the root, contains exactly one component. Because of $\mathcal{A}_{\text{entire}}$, the system's MTTF has not changed under the new tree, but its MTTR is lower, because a failure in a component can potentially be cured by microrebooting a subset of the components, possibly only the failed component.

Specifically, for a restart group G,

$$\text{MTTR}_G^{\text{II}} \leq \sum f_{c_i} \text{MTTR}_{c_i}$$

where c_i is G's i-th child, and f_{c_i} represents the probability that a manifested failure in G is minimally c_i-curable. As mentioned earlier, all observed failures in our ground station prototype were restart-curable, so the sum of f_{c_i} in any G is 1. As long as our system contains some component c_k such that $f_{c_k} > 0$ and $\text{MTTR}_{c_k} \neq \max(\text{MTTR}_{c_i})$, the result will be that $\text{MTTR}^{\text{II}} < \text{MTTR}^{\text{I}}$, since $\text{MTTR}^{\text{I}} = \max(\text{MTTR}_{c_i})$. Note in Table 2 that $\max(\text{MTTR}_{c_i})$ is different in the two trees. A whole system restart causes contention for resources that is not present when restarting just one component; this contention slows all components down.

Failed node	mbus	ses	str	rtu	fedrcom
MTTR^{I}	24.75	24.75	24.75	24.75	24.75
MTTR^{II}	5.73	9.50	9.76	5.59	20.93

Table 2. Tree II recovery: time to detect failed component plus time to recover system (in seconds).

Given that reboot tree `II` now has more than one recovery group, we must assume that the recovery manager is perfect, i.e., it is an omniscient oracle (in section 3.5.4 we will relax $\mathcal{A}_{\text{oracle}}$):

$\mathcal{A}_{\text{oracle}}$: *The system's recovery manager always recommends the minimal microreboot policy.*

Another assumption we have made in this transformation is that the recovery groups are independently recoverable:

$\mathcal{A}_{\text{independent}}$: *microrebooting a group will not induce failure(s) in any component of another recovery group.*

This assumption is important for recursive microreboots, as it captures the requirement of strong fault-isolation boundaries around groups. In section 3.5.3 we describe how to transform the reboot tree so that it preserves this property even when the design of our components impose the relaxation of $\mathcal{A}_{\text{independent}}$.

Note that, in defining the f_{c_i} measures, we are not attempting to build a model of the Mercury system's behavior, because we believe it is not possible to do so accurately. Instead, we use these measures simply to reason about the transformations. f_{c_i}'s are normally derived from historical observations made by the recovery manager.

3.5.2 Subtree Depth Augmentation

An interesting observation is that components may be decomposable into sub-components that have highly disparate MTTR and MTTF. In our system, the fedrcom component connects to the serial port at startup and negotiates communication parameters with the radio device; thereafter, it translates commands received from the other components to radio commands. Due to the hardware negotiation, it takes a long time to restart fedrcom; due to instability in the command translator, it crashes often. Hence fedrcom has high MTTR and low MTTF—a bad combination.

Luckily, fedrcom itself consists of two components: pbcom, which maps a serial port to a TCP socket, and fedr, the front end driver-radio that connects to pbcom over TCP. Separating the two requires a configuration change, but no code changes and no understanding of how the two components are written. pbcom is simple and very stable, but takes a long time to recover (over 21 seconds); fedr is buggy and unstable, but recovers very quickly (under 6 seconds). After restructuring the components and augmenting the reboot tree (Figure 6), it becomes possible to reboot the two components independently. We show the intermediate tree `II'`, which is identical to tree `II`, except the fedrcom component is split.

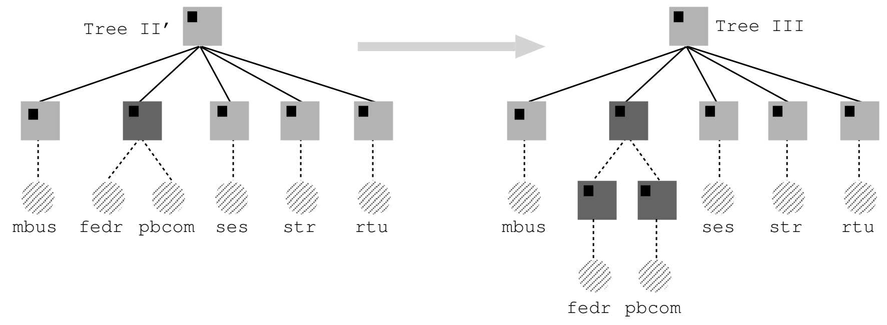


Figure 6. Subtree depth augmentation: tree `III`.

The new tree `III` has no effect on the system's MTTF, as the split did not affect the failure characteristics of what used to be fedrcom.

All failures that were previously minimally curable by a reboot of fedrcom are now minimally curable by a microreboot of pbcom, a microreboot of fedr, or a reboot of both together. Since $\text{MTTR}_{\text{fedr}} \ll \text{MTTR}_{\text{pbcom}}$ and $\text{MTTF}_{\text{fedr}} \ll \text{MTTF}_{\text{pbcom}}$, most of the failures will be cured by quick fedr microreboots, and a few of the failures will result in slow pbcom microreboots, whereas previously they would have all required slow fedrcom reboots. Therefore, the overall MTTR is improved.

Our measurements confirm this expected improvement: while before it took the system 20.93 seconds to recover from a fedrcom failure, it now takes 5.76 seconds to recover from a fedr failure and 21.24 seconds to recover from the seldom occurring pbcom failure. The increased value of pbcom's recovery time is due to communication overhead.

Some failures that manifest in either of the two new components may only be curable by rebooting both, i.e., we have not succeeded in separating fedrcom into completely independent pieces. We observed that multiple fedr failures eventually lead to a pbcom failure. We suspect this is due to the fact that, when fedr fails, its connection to pbcom is severed; due to bugs, pbcom ages every time it loses the connection and, at some point, the aging leads to its total failure. The presence of such correlated failures after splitting a component into pieces is in accord with software engineering reality.

The depth augmentation resulting from insertion of a joint node [fedr, pbcom], as opposed to having fedr and pbcom be top-level nodes under the root, is called for because correlated failures between fedr and pbcom exist: $f_{\text{fedr,pbcom}} > 0$ (i.e., there is a non-zero probability that a failure in fedr or pbcom is minimally curable by rebooting both fedr and pbcom). Subtree depth augmentation enables us to cure such correlated failures by microrebooting both components in parallel without rebooting the entire tree. If the two components could be made completely independent, then in theory we would have no correlated failures between fedr and pbcom ($f_{\text{fedr,pbcom}} = 0$), and there would be no benefit to the joint node.

We should note that the lower MTTR is achieved only if the recovery manager makes no mistakes when indicating which node to microreboot, i.e., $\mathcal{A}_{\text{oracle}}$ holds. section 3.5.4 will show why this assumption is necessary to realize the lower MTTR, and will examine the effect of relaxing $\mathcal{A}_{\text{oracle}}$.

From this example we may conclude the following: suppose we have a subsystem containing modules A and B, that any failure in the subsystem is guaranteed to be curable through a microreboot, and that $f_A, f_B, f_{A,B}$ correspond to the probabilities that a failure in the subsystem can be minimally cured by a microreboot of A only, B only, or [A,B] only, respectively. Then, if $f_{A,B} > 0$, in the engineering sense of being statistically significant, depth augmentation should be used to enable all three kinds of reboots. The same argument holds for the case when $f_A + f_B > 0$.

3.5.3 Consolidating Dependent Nodes

In the above example, the newly-created fedr and pbcom components, which started out as one, exhibited occasional correlated failures due to bugs in both components. In other cases, components such as ses and str exhibit correlated failures due to functional dependencies. Although ses and str were built independently, they synchronize with each other at startup and, when either is rebooted, the other will inevitably have to be rebooted as well. When restarted, both ses and str block waiting for the peer component to resynchronize. Such artifacts are not uncommon, especially when using COTS software. In fact, our experience with these components indicated that $f_{\text{ses}} \approx f_{\text{str}} \approx 0$, whereas $f_{\text{ses,str}} \approx 1$. That is, we observed that a failure/reboot in one of these components substantially always leads to a subsequent failure/reboot in the other.

However, the recovery manager (REC) does not know this ahead of time: under tree III, REC will restart ses/str when the component fails, then be told there is another failure, that was induced by the curing action, because of failure to resynchronize with str/ses. It will then reboot the peer component. Note that this does not violate $\mathcal{A}_{\text{oracle}}$: if the recovery manager made a mistake in its reboot choice, the original failure would persist. Here, the curing of the failure generates a new, related failure. This violates $\mathcal{A}_{\text{independent}}$.

To fix this, we encode the correlated-failure knowledge into the structure of a new reboot tree, as shown in the transformation of Figure 7. It is also possible for the recovery manager to learn these dependencies over time, by analyzing its history of reboots, but this is not yet implemented in Mercury. With the new reboot tree, whenever a failure occurs in either ses or str, it will force a reboot of both, yielding a recovery time proportional to $\max(\text{MTTR}_{\text{ses}}, \text{MTTR}_{\text{str}})$, instead of $\text{MTTR}_{\text{ses}} + \text{MTTR}_{\text{str}}$. This intuition is confirmed by experiment: with tree III it took on average 9.50 and 9.76 seconds to recover from a ses and str failure, respectively; with tree IV the system recovers in 6.25 and 6.11 seconds, respectively.

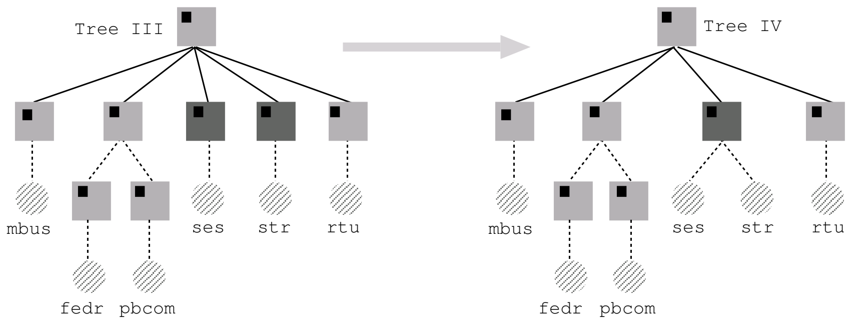


Figure 7. Group consolidation leads to tree IV.

Group consolidation and depth augmentation are duals of each other. We have seen that in a subsystem containing modules A and B, with the probabilities f_A, f_B, $f_{A,B}$, if the ability to microreboot each component is useful, then the group's depth should be augmented. Similarly, this section has shown that, if the ability to microreboot each component is not useful (i.e., $f_A + f_B \ll f_{A,B}$), then the restart group should be consolidated.

3.5.4 Promoting High-MTTR Nodes

There are two kinds of mistakes an imperfect recovery manager can make, which we call "guess-too-low" and "guess-too-high". In guess-too-low, the recovery manager suggests a microreboot at node n, when in fact a microreboot at one of n's ancestors is the minimum needed to fix the problem. In this case, the time spent rebooting only n will be wasted, because n's ancestor, and hence n as well, will eventually also need to be rebooted. In guess-too-high, the recovery manager suggests a reboot at a level higher than minimally necessary to cure the failure. The recovery time is therefore potentially greater than it had to be, since the failure could have been cured by microrebooting a smaller subsystem, with lower MTTR.

Guessing wrong is particularly bad when the MTTRs of components differ greatly, as is the case for fedr (5.76 sec) and pbcom (21.24 sec). However, we can structure the reboot tree to minimize the potential cost incurred from recovery manager failures: keep low-MTTR components low in the tree, and promote high-MTTR components toward the top, as illustrated with pbcom in Figure 8. As mentioned earlier, there exist failures that manifest in pbcom but can only be cured by a joint reboot of fedr and pbcom. We ran an experiment with a perfect recovery manager, that always correctly guessed when to do a joint restart, as well as with a faulty recovery manager that guessed wrong 30% of the time (we chose this percentage arbitrarily). The faulty recovery manager restarts pbcom, then realizes the failure is persisting, and moves up the tree to restart both fedr and pbcom, which eventually cures the failure. Our measurements confirm the impact of node promotion on system recovery time: in tree IV, Mercury took 29.19 seconds to recover from a pbcom failure in the presence of the faulty manager, in tree V it only takes on average 21.63 seconds to recover with the same faulty recovery manager.

Intuitively, the reason this structure reduces the cost of recovery manager mistakes is because mistakenly guessing that a pbcom-only reboot was required ultimately leads to pbcom being rebooted twice: once on its own, and then together with fedr. Tree V forces the two components to be rebooted together on all pbcom failures. Because tree IV is strictly more flexible than tree V, there is nothing that a perfect recovery manager could do in tree V but not in tree IV. Therefore, tree V can be better only when the manager is faulty.

Node promotion can be viewed as a special case of one-sided group consolidation, induced by asymmetrically correlated failure behavior. If the correlated behaviors were reasonably symmetric, as was the case for ses and str, then full consolidation would be recommended.

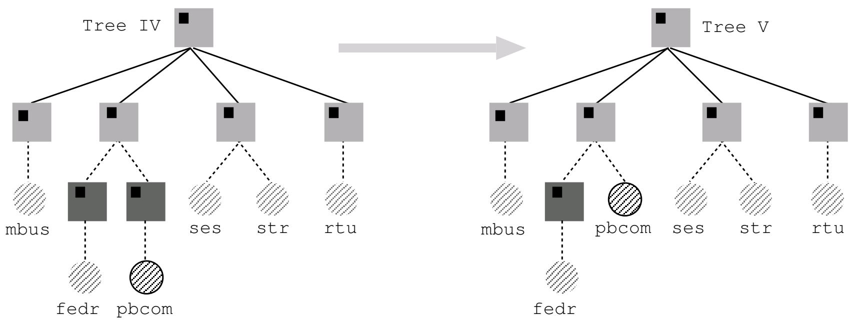


Figure 8. Node promotion yields tree V.

An interesting observation that we have not yet fully explored is the fact that a "free" fedr restart not only accounts for the possibility that the recovery manager guessed wrong, but also constitutes a prophylactic microreboot that rejuvenates [56] the fedr component, hence improving its MTTF. Remember that $\text{MTTF}_G^{\text{V}} \leq \min(\text{MTTF}_{c_i})$. Rejuvenation of fedr will likely increase $\text{MTTF}_{\text{fedr}}$, so in the cases in which the next component destined to fail would be fedr, with tree V this would happen later than with tree IV. Therefore, $\text{MTTF}^{\text{V}} \geq \text{MTTF}^{\text{IV}}$.

Tree	Recovery Manager	mbus	ses	str	rtu	fedr	pbcom	fedrcom
I	perfect	24.75	24.75	24.75	24.75	—	—	24.75
II	perfect	5.73	9.50	9.76	5.59	—	—	20.93
III	perfect	5.73	9.50	9.76	5.59	5.76	21.24	—
IV	perfect	5.73	6.25	6.11	5.59	5.76	21.24	—
IV	faulty	5.73	6.25	6.11	5.59	5.76	29.19	—
V	faulty	5.73	6.25	6.11	5.59	5.76	21.63	—

Table 3. Overal MTTR's (seconds). Rows show tree versions, columns represent component failures.

In this section we have seen how the reboot tree was first augmented: we added an entire new level of nodes across the tree, then we added an extra level in one of the subtrees. Then we started reducing the tree, by consolidating nodes within a recovery group and by promoting a high-MTTR component up the reboot tree. Table 3 centralizes our measurements, and Table 4 summarizes the tree transformations and reasoning behind them.

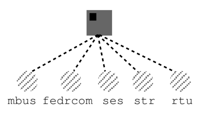
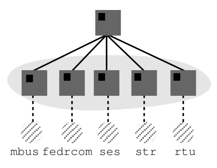
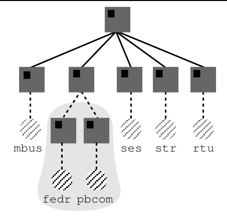
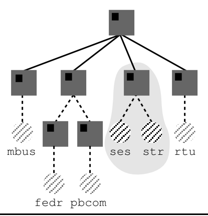
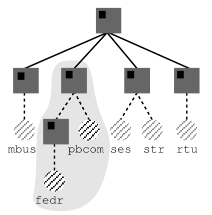
Original Tree	Augmentations		Reductions	
				
Original reboot tree. Any component failure triggers a reboot of the entire system.	Allows components to be independently rebooted, without affecting others.	Saves the high cost of rebooting pbcom whenever fedr fails (fedr fails often).	Reduces the delay in rebooting component pairs with correlated failures (ses and str).	Encodes information that prevents the recovery manager from making guess-too-low mistakes.
Embodies $\mathcal{A}_{\text{cure}}$, $\mathcal{A}_{\text{entire}}$	Embodies $\mathcal{A}_{\text{independent}}$, $\mathcal{A}_{\text{oracle}}$, $\mathcal{A}_{\text{cure}}$, $\mathcal{A}_{\text{entire}}$	Embodies $\mathcal{A}_{\text{independent}}$, $\mathcal{A}_{\text{oracle}}$, $\mathcal{A}_{\text{cure}}$, $\mathcal{A}_{\text{entire}}$	Embodies $\mathcal{A}_{\text{oracle}}$, $\mathcal{A}_{\text{cure}}$, $\mathcal{A}_{\text{entire}}$	Embodies $\mathcal{A}_{\text{cure}}$, $\mathcal{A}_{\text{entire}}$
Useful only if all component MTTRs are roughly equal.	Useful when $f_{A,B} > 0$ or $f_A + f_B > 0$	Useful when $f_{A,B} > 0$ or $f_A + f_B > 0$	Useful when $f_A + f_B \ll f_{A,B}$	Useful when the recovery manager is faulty i.e., it can guess wrong.

Table 4. Summary of reboot tree transformations

3.6 Lessons

The Mercury ground station is by design loosely coupled, its components are mostly stateless, and failure detection is based on application-level heartbeats. These are important RR-enabling properties and Mercury provides a good example of a simple RR system. The RR techniques described here are applicable to a wider set of applications. For example, we have found that many cluster-based Internet services [12] as well as distributed systems in general are particularly well suited to RR; in fact, many of the RR ideas originated in the Internet world. In this section, we extract from the Mercury experience some general principles we believe are fundamentally useful in thinking about applying RR to other systems.

3.6.1 Moving Boundaries

The most interesting principle we found was the benefit of drawing component boundaries based on MTTR and MTTF, rather than based solely on "traditional" modularity considerations such as state sharing. In transforming tree `III` to tree `IV`, we placed two independent components, `ses` and `str`, into the same recovery group. Presumably these two components are independent, yet we are partially "collapsing" the fault-isolation boundary between them by imposing the new constraint that, when either is rebooted, the other one is rebooted as well.

A dual of the above example is the splitting of `fedrcom` into the two separate components `fedr` and `pbcom`. As described in the text, these two components are intimately coupled from a functionality perspective; it is not an exaggeration to say that either is useless without the other. That is why in the original implementation `fedrcom` was a single process, i.e., communication between the components that became `fedr` and `pbcom` took place by sharing variables in the same address space. Post-splitting, the two components must explicitly communicate via IPC. This clearly adds communication overhead and complexity, but allows the two components to occupy different positions in the reboot tree, which in turn lowers MTTR. We conclude that if a component exhibits asymmetric MTTR/MTTF characteristics among its logical sub-components, rearchitecting along the MTTR/MTTF separation lines may often turn out to be the optimal engineering choice. Balancing MTTR/MTTF characteristics in every component is a step toward building a more robust and highly available system.

As explained in [18], RR attempts to exploit strong existing fault isolation boundaries, such as virtual memory, physical node separation, or kernel process control, leading to higher confidence that a sequence of restarts will effectively cure transients. To preserve this property, recovery-group boundaries should not subvert the mechanisms that create the existing boundaries in the first place.

3.6.2 Not All Downtime Is the Same

Unplanned downtime is generally more expensive than planned downtime, and downtime under a heavy or critical workload is more expensive than downtime under a light or non-critical workload. In our system, downtime during satellite passes (typically about 4 per day per satellite, lasting about 15 minutes each) is very expensive because we may lose some science data and telemetry from the satellite. Additionally, if the failure involves the tracking subsystem and the recovery time is too long, the communication link will break and the entire session will be lost. A large MTTF does not guarantee a failure-free pass, but a short MTTR can provide high assurance that we will not lose the whole pass as a result of a failure. As described in section 1, from among systems with the same availability A, those that have lower MTTR are often preferable.

Loosely coupled architectures often exhibit emergent properties that can lead to instability (e.g., noticed in Internet routing [36]) and investigating them is important for RR. There is also a natural tension between the cost of restructuring a system for RR and the cost (in downtime) of restarting it. Fine module granularity improves the system's ability to tolerate microreboots, but requires the implementation of a larger number of internal, asynchronous interfaces. The paradigm shift required of system developers could make RR too expensive in practice and, when affordable, may lead to buggier software. In some cases RR is simply not feasible, such as for systems with inherent tight coupling (e.g., real-time closed-loop feedback control systems).

Recursively rebootable systems rely on a generic execution infrastructure (EI) which is charged with instantiating the reboot tree mentioned in section 2, monitoring each individual component and/or subsystem, and prompting restarts when necessary. In existing restartable systems, the EI homologue is usually application-specific and built into the system itself. Some existing applications, most notably Internet services, are already incorporating a subset of these techniques (usually in an ad hoc fashion) and are primary candidates for systematic RR. Similarly, many geographically dispersed systems can benefit if they tolerate weakened consistency, due to the potential lack of reliability in their communication medium. We suspect the spectrum of applications that are amenable to RR is much wider, but still needs to be explored. We have applied RR to a Java 2 Enterprise Edition (J2EE) application server as well [20], and found significant self-management benefits.

Applying RR requires that components either be stateless or utilize soft state [18]. While recursive recovery can accomodate a wider range of recovery semantics for the cases where systems have hard state, we believe that using a uniform recovery strategy is very appealing both from an implementation and administration point of view. We have therefore developed the notion of *crash-only software* [19], which we summarize in the following section.

4 Crash-Only Software

Crash-only programs crash safely and recover quickly. There is only one way to stop such software—by crashing it—and only one way to bring it up—by initiating recovery. Crash-only systems are built from crash-only components, and the use of transparent component-level retries hides intra-system component crashes from end users. In high level terms, a crash-only system is defined by the equations *stop=crash* and *start=recover*. In this section, we describe the benefits of the crash-only design approach by analogy to physics, describe the internal properties of components in a crash-only system, the architectural properties governing the interaction of components, and a restart/retry architecture that exploits crash-only design, including our work to date on a prototype using J2EE.

We believe there should be only one way to stop or recover a system: by crashing it. Unfortunately, most non-embedded software systems have at least two ways to stop, one of which is a so-called "clean shutdown"—often an excuse for writing code that recovers both poorly and slowly after crashes. The recursive microreboots framework encourages development of crash-only software that needs no warning prior to shutting down, the same way a car, a TiVo, or a printer can be safely shut off by just pressing the on/off button. A great difficulty in developing recovery code is that it runs rarely and, whenever it does run, it must work perfectly; the crashing approach forces recovery code to be exercised regularly as part of the normal startup procedure. The reason clean shutdowns exist is most often for performance reasons, such as in-memory caching of buffers in file systems, and industry has shown that much better recoverability can be obtained by giving up some of this performance (e.g., Oracle's DBMS can checkpoint more often in order to reduce recovery time [62]).

We have not explored the extent to which crash-only design is applicable to non-Internet systems. Thus, this discussion encompasses only the class of systems distinguished by large scale, stringent high availability requirements, built from many heterogenous components, accessed over standard request-reply protocols such as HTTP, serving workloads that consist of large numbers of relatively short tasks that frame state updates, and subjected to rapid and perpetual evolution. We restrict our attention to single installations that reside inside one data center, that don't span administrative domains and don't communicate over the WAN.

4.1 Why Crash-Only Design ?

Mature engineering disciplines rely on macroscopic *descriptive* physical laws to build and understand the behavior of physical systems. These sets of laws, such as Newtonian mechanics, capture in simple form an observed physical invariant. Software, however, is an abstraction with no physical embodiment, so it obeys no physical laws. Computer scientists have tried to use *prescriptive* rules, such as formal models and invariant checks, to reason about software. These rules, however, are often formulated relative to an abstract model of the software that does not completely describe the behavior of the running system, which includes hardware, the OS, and runtime libraries. As a result, the prescriptive models do not provide a complete description of how the implementation will behave in practice, because many physically possible states of the complete system do not correspond to any state in the abstract model.

With the crash-only property, we are trying to impose, from the outside, macroscopic behavior that coerces systems into a simpler, more predictable universe with fewer states and simpler invariants. Each crash-only component has a single "power-off switch" and a single "power-on switch"; the switches for larger systems are built by wiring together their subsystems' switches in ways described by section 4.2. A component's power-off switch implementation is entirely external to the component, thus not invoking any of the component's code and not relying on correct internal behavior of the component. Examples of such switches include `kill -9` sent to a UNIX process, or turning off the virtual, or physical, machine that is running some software inside it. Keeping the power-off switch mechanism external to components makes it a high confidence "component crasher." Consequently, every component in the system must be prepared to suddenly be deactivated. Power-off and power-on switches must provide a small repertoire of high-confidence, simple behaviors, leading to a small state space. Of course, the "virtual shutdown" of a virtual machine, even if invoked with `kill -9`, has a much larger state space than the physical power switch on the workstation, but it is still vastly simpler than the state space of a typical program hosted in the VM, and it does not vary for different hosted programs. Indeed, the fact that virtual machines are relatively small and "simple" compared to the applications they host has been successfully invoked as an argument for using VMs for inter-application isolation [103].

4.1.1 Crash-Only and Fault Model Enforcement

A crash-only system makes it affordable to coerce every detected failure into component-level crash(es); this leads to a simple fault model in that components only need to know how to recover from one type of failure. Fault model enforcement [76] uses such an approach to turn unknown faults into crashes, effectively coercing reality into a well-understood, simple fault model. By performing recovery based on this fault model, [76] managed to improve availability in a cluster system. Much existing literature assumes unrealistic fault models (e.g., that failures are uncorrelated and occur according to well-behaved tractable distributions) for analysis of system behavior; fault model enforcement can increase the impact of such work.

Similarly, a system built from components that tolerate crashes at low cost makes it affordable to use software rejuvenation [56] to prevent failure. Rejuvenation can be triggered by fail-stutter behavior [8], a workload trough, or based on mathematical models of software aging [40].

4.2 Properties of Crash-Only Software

To make components crash-only, we require that all persistent state be kept in dedicated state stores, that state stores provide applications with the right abstractions, and that state stores be crash-only. To make a system of interconnected components crash-only, it must be designed so that components can tolerate their peers' crashes. This means we require strong modularity with relatively impermeable component boundaries, timeout-based communication and lease-based resource allocation, and self-describing requests that carry a time-to-live and information on whether they are idempotent. Many Internet systems today have some subset of these properties, but we do not know of any that combines all properties into a true crash-only system.

In section 4.3 we will show how crash-only components can be glued together into a robust Internet system based on a restart/retry architecture; in the rest of this section we describe in more detail the six properties of crash-only systems. The first three relate to intra-component state management, while the last three relate to inter-component interactions. While we recognize that many of these sacrifice performance, we reiterate out belief that the time has come for robustness to reclaim its status as a first-class citizen. We are currently implementing the crash-only properties in an open-source Java 2 Enterprise Edition (J2EE) application server; details can be found in [20]. We are also finalizing a new version of the Mercury software, and in the following sections we will illustrate how some of these properties are provided in the redesigned ground station.

4.2.1 Intra-Component Properties

Persistent state is managed by dedicated state stores, leaving applications with just program logic. Specialized state stores (e.g., relational and object-oriented databases, file system appliances, distributed data structures [49], middle-tier persistence layers [64, 59]) are much better suited to manage persistence and consistency than code written by developers with minimal training in systems programming. Applications become soft-state clients of the state stores, which allows them to have simpler and faster recovery routines. A popular example of such separation can be found in three-tier Internet architectures, where the middle tier is largely stateless and relies on backend databases to store data. The new Mercury software, unlike its previous version, maintains persistent state, and we store all such state in a MySQL database.

State stores are crash-only, otherwise the problem has just moved down one level. Many commercial off-the-shelf state stores available today are crash-safe, such as databases and the various network-attached storage devices, but most are not crash-only, because they recover slowly. A large group of products, however, offer tuning knobs that permit the administrator to trade performance for improved recovery time, such as making checkpoints more often in the Oracle DBMS [62]. An example of a pure crash-only state store is the Postgres database system [96], which avoids write-ahead logging and maintains all data in one append-only log. Recovery is practically instantaneous, because it only needs to mark the transactions that failed (i.e., uncommitted at the time of the crash). The latest version of Mercury does not use Postgres, but its MySQL database is sufficiently small that it recovers quickly from any crash. The ACID semantics ensure that persistent state does not become corrupt.

Abstractions and guarantees provided by the state store match the requirements of the application. This property makes it easier to build crash-only state stores that offer both good performance and fast recovery. The abstraction provided by the state store to its clients must enable the application to operate at its own semantic level. For example, it is preferable for an application that maintains customer data to use the record-based abstraction offered by a relational database, rather than the array-of-bytes abstraction offered by a file system. By ensuring a close match between the offered and the required abstraction, the state store can exploit application semantics to build simpler and better state stores. For example, Berkeley DB [79] is a data store supporting B+tree, hash, queue, and record abstractions. It can be accessed through four different interfaces, ranging from no concurrency control, no transactions, no disaster recovery to a multi-user, transactional data store API with logging, fine-grained locking, and support for data replication. Applications can use the abstraction that is right for their purposes and the underlying state store optimizes its operation to fit those requirements. Workload characteristics can also be leveraged by state stores; for instance, expecting a read-mostly workload allows a state store to utilize write-through caching, which can significantly improve recovery time and performance.

It appears that Internet systems have already started standardizing on a small set of state store types that they commonly use. Most such systems would likely be satisfied if they had transactional ACID stores (e.g., databases for customer data), simple read-only stores (e.g., NetApp filers for static HTML and GIFs), non-durable single-access stores (e.g., replicated in-memory store holding user session state), and a soft state store (e.g., web cache).

4.2.2 Extra-Component Properties

For a crash-only system to gracefully tolerate subsystem crashes, which temporarily make them unavailable to serve requests, components and their relationships must follow these guidelines:

Components are modules with externally enforced boundaries that provide strong fault containment. The desired isolation can be achieved using virtual machines such as VMware, isolation kernels [103], task-based intra-JVM isolation [32], Luna-style extensions [51], OS processes, etc. For example, web service hosting providers often use multiple virtual machines on one physical machine to offer their clients individual web servers they can administer at will, without affecting other customers. The boundaries between components delineate distinct, individually recoverable stages in the processing of requests. In both versions 1 and 2 of Mercury, isolation was provided by the fact that each component ran in its own Java virtual machine.

All components use timeout-based communication and all resources are leased, rather than permanently allocated. Whenever a request is issued from component A to component B, whether in the form of a message or an RPC, A starts a timer and expects an answer before the time runs out. If no response is received, A assumes B has failed and reports it to the recovery manager, which can microreboot B if appropriate. Timeouts provide an orthogonal mechanism for turning all non-Byzantine failures, both at the component level and at the network level, into fail-stop events (i.e., the failed entity either provides results or is stopped), even though the components are not necessarily fail-stop. Such behavior is easier to accomodate, making it more likely for faults to be well contained.

All resources allocated to a request are based on leases [44]. If a request dies due to components failing and/or running out of time, all resources associated with it will eventually be reclaimed. Infinite timeouts or leases are not acceptable; the maximum-allowed timeout and lease are specified in an application-global policy. This way, the probability that the system becomes blocked or hung is very small.

All requests carry a time-to-live and an indication of whether they are idempotent. This information will typically be set by the web front ends: timeouts as a function of load or service level agreements, and idempotency flags based, for instance, on URL substrings that determine the type of request. Many interesting operations in an Internet service are idempotent, or can easily be made idempotent by keeping track of sequence numbers or by wrapping requests in transactions; some large Internet services have already found it practical to do so [81]. Over the course of its lifetime, a request will split into multiple sub-requests, which may rejoin, similarly to nested transactions. Recovering from a failed idempotent operation entails simply reissuing it; for non-idempotent operations, the system can either roll them back, apply compensating operations, or tolerate the inconsistency resulting from a retry. Such transparent recovery of the request stream can hide intra-system component failures from the end user.

4.3 A Restart/Retry Architecture

A component infers failure of a peer component either based on a raised exception or a timeout. When a component is reported failed, a recovery agent crash-restarts it. Components waiting for an answer from the restarted component receive a *RetryAfter(n)* exception, indicating that the in-flight requests can be re-submitted after n msec (the estimated time to recover). If the request is idempotent and its time-to-live allows it to be resubmitted, then the requesting component does so. Otherwise, a failure exception is propagated up the request chain until either a previous component decides to resubmit, or the client web browser needs to be notified of the failure. The web front-end issues an HTTP/1.1 `Retry-After` directive to the web browser with an estimate of the time to recover, and retry-capable clients can resubmit the original HTTP request.

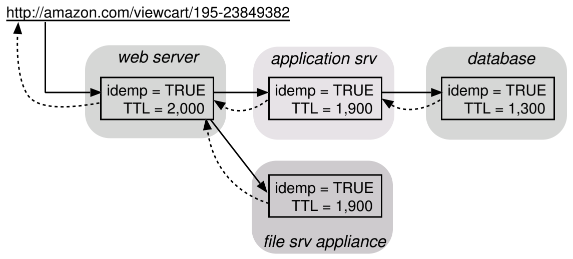


Figure 9. A simple restart/retry architecture

In Figure 9 we show a simple, coarse grain restart/retry example, in which a request to view a shopping cart splits into one subrequest to fetch the shopping cart content from the database and another subrequest for fetching from a file system the static content for generating the web page. Should the database become unavailable, the application server either receives a *RetryAfter* exception or times out, at which time the application server can decide whether to resubmit or not. Within each of the subsystems shown in Figure 9, we can imagine each subrequest further splitting into finer grain subrequests submitted to the respective subsystems' components.

Timeout-based failure detection is supplemented with traditional heartbeats and progress counters [92]. The counters—compact representations of a component's processing progress—are usually placed at state stores and in messaging facilities, where they can map state access and messaging activity into per-component progress. Components themselves can also implement progress counters that more accurately reflect application semantics, but they are less trustworthy, because they are inside the components. Primitive progress-based execution control is already available in PHP, a server-side scripting language used for writing dynamic web pages: it offers a `set_time_limit` function that limits the maximum execution time of a script; if this limit is reached, the script is killed and an error is returned.

Resubmitting requests to a component that is recovering can overload it and make it fail again; for this reason, the *RetryAfter* exceptions provide an estimated time-to-recover. Moreover, a maximum limit on the number of retries is specified in the application-global policy, along with the lease durations and communication timeouts. These numbers can be dynamically estimated or simply captured in a static description of each component, similar to deployment descriptors for Enterprise JavaBeans. In the absence of such hints, a simple load balancing algorithm or exponential backoff is used.

4.4 Discussion

We are focusing currently on applications whose workloads can be characterized as relatively short-running tasks that frame state updates. Substantially all Internet services, and many enterprise services, fit this description—in part because the evolution of the tools for building such systems forced designers into the "three-tier application" mold. We expect there are many applications outside this domain, such as interactive desktop applications, that could not easily be cast this way, and for which deriving a crash-only design would be impractical or infeasible. We also restricted the domain of Internet systems to those interacting based on HTTP, although Internet services might use additional protocols.

The restart/retry architecture has "execute at least once" semantics; in order to be highly available and correct, most requests it serves must be idempotent. This requirement might be inappropriate for some applications. Our architecture does not explicitly handle Byzantine failures or data errors, but such behavior can be turned into fail-stop behavior using well-known orthogonal mechanisms, such as triple modular redundancy [48] or clever state replication [21].

Rebooting is a correctness-preserving form of restart only to the extent that no "critical" state is lost and no inconsistency created. We chose in our current work to target three-tiered Internet applications because they are structured such that these problems are minimized. We can distinguish three kinds of state in such applications: (a) persistent state, the state that the application exists to manipulate, such as customer orders or a customer's shopping cart; (b) session state, which tracks where the customer is in the site's workflow—this is captured by cookies on the client side that identify session-state bundles on the server side; (c) transient state—the state created by the application as it runs, but which is incidental to the customer's experience: local variables, the heap, etc. Clearly persistent state for such applications is critical, but fortunately it is stored in a separate tier which has its own recovery mechanism (a relational database). A separate tier for session-state storage could take advantage of session state's very specific access characteristics to optimize a state store for fast recovery [64].

Rebooting cannot recover from a hard failure in a disk drive or other hardware component, such as Mercury's radio unit. Such failure is likely to happen eventually and more thorough failure detection can optimize the recovery process by notifying a human operator, or failing over to a redundant spare, before attempting reboot-based recovery. Comprehensive failure detection and logging are not the goals of this effort, though they are long-term goals for Mercury and are absolutely necessary for proper recovery.

We expect throughput might suffer in crash-only systems, but we consider this concern secondary to the high availability and robustness we expect in exchange. As described in section 1, Moore's Law has had a powerful effect on system performance, but has not resolved the dependability problem; if a software design paradigm can address the latter, then that is a good venue to focus our efforts.

In today's Internet systems, fast recovery is obtained by overprovisioning and using rapid failure detection to trigger failover. Such failover can sometimes mask hours-long recovery times. Crash-only software is complementary to this approach and can help alleviate some of the complex and expensive management requirements for highly redundant hardware, because faster recovering software means less redundancy is required.

5 Related Work

The rebooting technique embodied in recursive microreboots has been around as long as computers themselves, and our work draws heavily upon decades of system administration history. The RR model refines and systematizes a number of known techniques, in an attempt to turn the "high availability folklore" into a well-understood tool.

Many techniques have been advocated for improving software dependability, ranging from better software engineering [14] and object oriented programming languages [33] to formal methods that predict/verify properties based on a mathematical model of the system [90]. Language-based methods, such as static analysis [29], detect problems at the source-code level. Some programming languages prevent many programming errors by imposing restrictions, such as type safety [105] or a constrained flow of control [73], or by providing facilities like garbage collection [69]. This vast body of techniques have significantly improved software quality and, without them, it would be impossible to build today's software systems. Unfortunately, many bugs cannot be caught, and most software is not written by experts; hence the need for recovery-oriented techniques.

In the remainder of this section we will describe some of the techniques used to cope with bugs and other types of faults, and we will show how this prior work relates to our research. Techniques fall into three broad categories: fault detection, fault containment, and fault recovery.

5.1 Detection

Rapid detection is a critical ingredient of fast recovery and is therefore an integral part of any recovery-oriented approach to system dependability. The work presented in this paper concerns itself primarily with fail-fast software and does not address the detection of byzantine faults or data integrity violations; these aspects are orthogonal to our approach, as they can be implemented in failure monitors.

Programmer-inserted assertions and periodic consistency checks—a staple of defensive programming—are an excellent way to catch bugs. Database and telecommunications systems take this one step further and employ audit programs to maintain memory and data integrity. In the 4ESS telephone switch [104], for example, so-called mutilation detection programs constantly run in the background to verify in-memory data structures. When a corrupt structure is found, its repair is attempted by a correction module; if the repair fails, the data structure is reinitialized. Such applications integrate well with the recursive recovery (RR) framework, as they can notify the recovery manager when detecting an unrecoverable fault, and allow the manager to decide what higher level recovery action to take.

Active application-level fault detection can significantly reduce detection time, but it cannot be fully relied on, so the monitoring agents have to take an active role as well. The use of heartbeats [5] or watchdog timers [68] for software is a reliable complement, because it uses the absence of action to infer a failure, rather than waiting for a proactive report of the failure. Such timeout-based mechanisms lie at the very heart of the restart/retry architecture described in section 4.3.

Finally, some amount of knowledge about the application's semantics enables end-to-end failure detection. Infrastructure monitoring companies [77, 87, 28, 58, 53] actively supervise corporate databases, application servers, and web servers by monitoring specific aspects (e.g., the alert log contents of a DBMS, the throughput level of a web server). The infrastructure operator is notified when something has failed, is exhibiting fail-stutter behavior [8], or when resource utilization is approaching application-specific critical levels and may warrant rejuvenation. A similar approach has been taken in gray-box systems [7], where knowledge of the internal workings of an operating system is captured in information and control layers, which can then observe OS activity and infer facts about the OS state without using explicit interfaces. In JAGR [20] we use tools such as Pinpoint [24] to map end-to-end failures or performance degradation onto the specific components that are causing the failure and recover them with surgical precision.

5.2 Containment

Fault containment techniques aim to confine faults, so they affect as little of a system as possible and allow for localized recovery. Good fault containment reduces the number of recovery attempts required to resolve a failure, which results in faster recovery times. Drawing strong fault containment boundaries has long been considered good engineering and is found in many successful systems; for some, strong fault isolation is a fundamental principle [23]. Techniques used for containment range from physical isolation for cluster nodes to hardware-assisted virtual memory and sophisticated software-based techniques. For example, the *taintperl* package [100] employs dynamic dataflow analysis to quarantine data that may have beeen "contaminated" by user inputs and thus might contain malicious executable code—a serious security threat for web sites using cgi-bin scripts. Applications already employing such techniques are more amenable to our localized recovery, but it is difficult to retrofit such approaches without significant changes to the applications.

Another set of containment technologies however holds much more promise for RR. Virtual machine monitors provide a powerful way to draw strong fault isolation boundaries between subsystems without having to change the application software. A virtual machine monitor [43, 16] is a layer of software between the hardware and operating system, which virtualizes all the hardware resources and exports a conventional hardware interface to the software above; this hardware interface defines a virtual machine (VM). Multiple VMs can coexist on the same real machine, allowing for multiple copies of an operating system to run simultaneously. In our research group we use virtual machines to isolate each publically accessible network service (sshd, web servers, etc.) from all the other services running on the same host: in each VM we run a copy of the OS and one single service. This way, a vulnerability in a web server will not directly compromise any of the other services. Motivated by this type of VM uses, isolation kernels [103] provide a VM-like environment, but trade off completeness of the virtualized machine for significant gains in performance and scalability. While requiring slight modifications to the services, isolation kernels provide a light weight mechanism for building strong fault isolation boundaries. One isolation technique that we find particularly useful for RR, considering our current choice for using a J2EE platform, is the multi-tasking Java VM [32], which allows process-like isolation between applications running in the save JVM. Virtualization is a powerful method for making legacy software systems recursively recoverable.

The isolation of operating system services into separate components, for purposes that include containment, has been pioneered by microkernels [1]; more recent work has demonstrated that the performance overhead of achieving such isolation is negligible [50].

5.3 Recovery

Virtually all recovery techniques rely on some form of redundancy, in the form of either functional, data, or time redundancy. In the case of functional redundancy, good processors can take over the functionality of failed processors, as in the case of Tandem process pairs [10] or clusters [38]. Some forms of recovery use time redundancy and diversity of programming logic (e.g., recovery blocks [6], where the computation of an erroneous result triggers a retry using a different algorithm), but such techniques have had only limited appeal due to their cost of development and maintenance, as well as difficulty in ensuring true independence among the alternate program paths.

Failover to a standby node is a powerful high availability technique, but cannot be solely relied on. For instance, whenever a node fails in a cluster, the system as a whole enters a period of vulnerability in which further failures could cripple it, as was the case for CNN.com on 9/11/01. The CNN news site collapsed under the rapidly increasing load, because thrashing nodes could not recover quickly and good nodes could not reintegrate fast enough to take over from the thrashing ones [63]. Our project's emphasis on reducing recovery time complements redundancy-based failover by reducing the system's window vulnerability to additional failures. Most Internet services run on very large clusters of computers (as an extreme example, Google uses 20,000 CPUs in 5 geographically distributed sites to serve google.com [2]); at this scale, nodes going down is a frequent event, making rapid reintegration critical.

Finally, node redundancy does not scale indefinitely, because of the tension between number of nodes and diversity. Having a large number of diverse nodes increases the system's robustness to failure, but at the same time makes it very difficult to administer and maintain. On the other hand, having a large number of mostly identical nodes makes management easier, but drastically reduces system robustness. For example, when an obscure bug in a version of Akamai's software manifested simultaneously on all nodes running that release, a large part of the content distribution network went down. Rapid node recovery allows even widespread failure to be quickly eradicated.

The benefits of restarting quickly after failures have been recognized by many system designers, as they employed techniques ranging from the use of non-volatile memory (e.g., Sprite's recovery box [9], derivatives of the Rio system [25, 66]) to non-overwriting storage combined with clever metadata update techniques (e.g., the Postgres DBMS [96], Network Appliance's filers [54]). A common theme, which we have identified in section 4.3, is that of segregating and protecting state that needs to be persistent, while treating the rest as soft state. We see this approach reflected in recent work on soft-state/hard-state segregation in Internet services [38, 49] and we adopt it as a basic tenet for our restart/retry model.

Checkpointing [101, 22, 99] employs dynamic data redundancy to create a believed-good snapshot of a program's state and, in case of failure, return the program to that state. An important challenge in checkpoint-based recovery is ensuring that the checkpoint is taken before the state has been corrupted [102]. Another challenge is deciding whether to checkpoint transparently, in which case recovery rarely suceeds for generic applications [65], or non-transparently, in which case source code modifications are required. In spite of these problems, checkpointing is a useful technique for making applications restartable, and was successfuly utilized in [55], where application-specific checkpointing was combined with a watchdog daemon process to provide fault tolerance for long-running UNIX programs. ARMORs [60] provide a micro-checkpointing facility for application recovery, but applications must be (re)written to use it; limited protection is provided for legacy applications without their own checkpointing code. We believe that maintaining state in a suitable store (see section 4.3) obviates the need for checkpoints.

Log-based recovery techniques cost more than checkpoint-based recovery, but are considerably more powerful, because they allow the system to return to potentially any moment in time prior to the failure. Undo and redo logs [45, 88] allow the system to undergo a set of legal transformations that will take it from an inconsistent state, such as that induced by a bug or hardware failure, to a consistent one. Logs enable transactions [46], which are the fundamental unit of recovery for applications that require ACID [45] semantics. In a new twist on the undo approach, system-level undo [15] allows for an entire system's state to be rolled back, repaired, and then brough back to the present.

Some of the most reliable computers in the world are guided by the same principles we are following, and use dynamic recovery to mask failure from upper layers. For example, in IBM S/390 mainframes [95], computation is duplicated within each CPU and the results are compared before being committed to memory. A difference in results freezes execution, reverts the CPU to its state prior to the instruction, and the failed instruction is retried. If the results now compare OK, the error is assumed to have been transient and execution continues; if they are different, the error is considered to be permanent, the CPU is stopped and dynamically replaced with a spare CPU by reconfiguring data paths. Execution of the instruction stream resumes transparently to the operating system. In its memory system, the S/390 performs background scrubbing on main memory data to reduce the frequency of transient single-bit failures; faulty memory chips are dynamically replaced. While the S/390 is a reliable computer that hardly ever fails, in large Internet services most downtime is not caused by hardware. For this reason, RR performs fault detection at all levels in the system, thus being able to capture more failure scenarios than could be detected by the hardware alone. Moreover, the recursive recovery approach accepts that always choosing the right level in the system at which to recover is difficult, so it progressively tries higher and higher layers until the problem is eradicated.

All of these approaches have described ways to recover once failure is encountered. Software rejuvenation [56] on the other hand, terminates an application and restarts it at a clean internal state to prevent faults from accumulating and causing the application to fail. Although in this paper we have focused on reactive rather than proactive restarts, rejuvenation is an integral part of the RR strategy. Rejuvenation has also found its way into Internet server installations based on clusters of hundreds of workstation nodes; many such sites use rolling reboots to clean out stale state and return nodes to known clean states, Inktomi being one example [12]. IBM's xSeries servers also employ rejuvenation for improved availability [57].

6 Conclusion

In this paper we made the case for shifting some of the focus from traditional performance research to finding ways to improve recovery performance. Using MTTR as a metric for availability holds promise in shaping this direction, in much the same way SPEC benchmarks did for hardware performance. As an illustration of this belief, we have developed the recursive recovery (RR) framework, which reduces MTTR by recovering minimal subsets of a failed system's components. If localized, minimal recovery does not work, progressively larger subsets are recovered. When applying RR to Internet systems and Internet-like services, our recovery method of choice is the reboot; a fine grain "surgical" reboot is called a microreboot.

We applied recursive microreboots to Mercury, a COTS-based satellite ground station. We were able to improve recovery time of our Mercury software by up to a factor of four, without modifying any of the source code. Although we have not thoroughly measured the benefits resulting from automating the failure detection, we have observed them to be significant—in the past, relying on operators to notice failures was adding minutes or hours to the recovery time. There is an increasing trend toward complex, hard-to-manage software systems that integrate large numbers of COTS modules; we believe that recovery-oriented computing approaches hold a lot of promise as a dependability technique in such systems.

We discussed the notion of crash-only software and crash-only designs, through which we can obtain better reliability and higher availability in Internet systems. By using the *stop=crash*, *start=recover* approach, fault models that applications are required to handle can be simplified, thus encouraging simpler recovery routines which have higher chances of being correct. Crash-only components assemble into crash-only systems; once we surround such a system with a suitable infrastructure, we obtain a recursively rebootable system [19]. Transparent, application-agnostic recovery based on component-level restart enables restart/retry architectures to hide intra-system failure from the end users, thus improving the perceived reliability of the service. Current results [20] give us confidence that improvements similar to Mercury's can be obtained for larger systems, and that they that can be turned into autonomous, self-recovering systems providing higher availability than today's architectures.

7 Acknowledgements

We are indebted to the reviewers and our colleagues at Stanford University and U.C. Berkeley for their insights and helpful comments on our paper. This research benefitted from the financial support of the National Aeronautics and Space Administration (award NAG3-2579), the National Science Foundation (Career Award 133966), IBM Corporation through a Faculty Award, and the USENIX Society through a student scholarship.

References

[1] M. J. Accetta, R. V. Baron, W. J. Bolosky, D. B. Golub, R. F. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proc. USENIX Summer Conference*, Atlanta, GA, 1986.

[2] A. Acharya. Reliability on the cheap: How I learned to stop worrying and love cheap PCs. In *2nd Workshop on Evaluating and Architecting System Dependability*, San Jose, CA, 2002. Invited Talk.

[3] E. Adams. Optimizing preventative service of software products. *IBM Journal of Research and Development*, 28(1):2–14, 1984.

[4] T. Adams, R. Igou, R. Silliman, A. M. Neela, and E. Rocco. Sustainable infrastructures: How IT services can address the realities of unplanned downtime. Research Brief 97843a, Gartner Research, May 2001. Strategy, Trends & Tactics Series.

[5] M. K. Aguilera, W. Chen, and S. Toueg. Heartbeat: a timeout-free failure detector for quiescent reliable communication. In *Proc. 11th International Workshop on Distributed Algorithms*, Saarbrücken, Germany, 1997.

[6] T. Anderson and R. Kerr. Recovery blocks in action: A system supporting high reliability. In *Proc. 2nd International Conference on Software Engineering*, San Francisco, CA, 1976.

[7] A. Arpaci-Dusseau and R. Arpaci-Dusseau. Information and control in gray-box systems. In *Proc. 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, 2001.

[8] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-stutter fault tolerance. In *Proc. 8th Workshop on Hot Topics in Operating Systems*, Elmau/Oberbayern, Germany, 2001.

[9] M. Baker and M. Sullivan. The Recovery Box: Using fast recovery to provide high availability in the UNIX environment. In *Proc. Summer USENIX Technical Conference*, San Antonio, TX, 1992.

[10] J. F. Bartlett. A NonStop kernel. In *Proc. 8th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, 1981.

[11] E. Brewer. Inktomi insights. Personal Communication, 2000.

[12] E. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, July 2001.

[13] E. Brewer. Running Inktomi. Personal Communication, 2001.

[14] F. P. Brooks. *The Mythical Man-Month*. Addison-Wesley, Reading, MA, Anniversary edition, 1995.

[15] A. B. Brown and D. A. Patterson. Undo for operators: Building an undoable e-mail store. In *Proc. USENIX Annual Technical Conference*, San Antonio, TX, 2003.

[16] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. DISCO: running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

[17] G. Candea, M. Delgado, M. Chen, and A. Fox. Automatic failure-path inference: A generic introspection technique for software systems. In *Proc. 3rd IEEE Workshop on Internet Applications*, San Jose, CA, 2003.

[18] G. Candea and A. Fox. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proc. 8th Workshop on Hot Topics in Operating Systems*, Elmau/Oberbayern, Germany, 2001.

[19] G. Candea and A. Fox. Crash-only software. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, Lihue, Hawaii, 2003.

[20] G. Candea, P. Keyani, E. Kiciman, S. Zhang, and A. Fox. JAGR: An autonomous self-recovering application server. In *Proc. 5th International Workshop on Active Middleware Services*, Seattle, WA, June 2003.

[21] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. 3rd USENIX Symposium on Operating Systems Design and Implementation*, New Orleans, LA, 1999.

[22] K. M. Chandy and C. V. Ramamoorthy. Rollback and recovery strategies for computer programs. *IEEE Transactions on Computers*, 21(6):546–556, June 1972.

[23] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proc. 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, 1995.

[24] M. Chen, E. Kiciman, E. Fratkin, E. Brewer, and A. Fox. Pinpoint: Problem determination in large, dynamic, Internet services. In *Proc. International Conference on Dependable Systems and Networks*, Washington, DC, June 2002.

[25] P. M. Chen, W. T. Ng, G. Rajamani, and C. M. Aycock. The Rio file cache: Surviving operating system crashes. In *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, 1996.

[26] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proc. 18th ACM Symposium on Operating Systems Principles*, Lake Louise, Canada, 2001.

[27] T. C. Chou. Beyond fault tolerance. *IEEE Computer*, 30(4):31–36, 1997.

[28] Computer_Associates. Unicenter CA-SYSVIEW realtime performance management. `http://www.ca.com`, Oct. 2002.

[29] P. Cousot, editor. *Static Analysis*. Springer Verlag, 2001.

[30] J. W. Cutler, A. Fox, and K. Bhasin. Applying the lessons of Internet services to space systems. In *Proc. IEEE Aerospace Conference*, Big Sky, MT, 2001.

[31] J. W. Cutler and G. Hutchins. Opal: Smaller, simpler, luckier. In *Proc. AIAA Small Satellite Conference*, Logan, UT, 2000.

[32] G. Czajkowski and L. Daynés. Multitasking without compromise: A virtual machine evolution. In *Proc. Conference on Object Oriented Programming Systems Languages and Applications*, Tampa Bay, FL, 2001.

[33] O.-J. Dahl and K. Nygaard. Simula—an Algol-based simulation language. *Communications of the ACM*, 9(9):671–678, Sep 1966.

[34] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C. Liu, L. Wei, P. Sharma, and A. Helmy. Protocol independent multicast (PIM), sparse mode protocol: Specification, March 1996. Internet Draft.

[35] A. DiGiorgio. The smart ship is not enough. *Naval Institute Proceedings*, 124(6), June 1998.

[36] S. Floyd and V. Jacobson. The synchronization of periodic routing messages. *IEEE/ACM Transactions on Networking*, 2(2):122–136, Apr. 1994.

[37] S. Floyd, V. Jacobson, C. Liu, and S. McCanne. A reliable multicast framework for light-weight sessions and application level framing. In *Proc. ACM SIGCOMM Conference*, Boston, MA, 1995.

[38] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proc. 16th ACM Symposium on Operating Systems Principles*, Saint-Maló, France, 1997.

[39] S. Garg, Y. Huang, C. Kintala, and K. Trivedi. Minimizing completion time of a program by checkpointing and rejuvenation. In *Proc. ACM Conference on Measurement and Modeling of Computer Systems*, Philadelphia, PA, 1996.

[40] S. Garg, A. V. Moorsel, K. Vaidyanathan, and K. S. Trivedi. A methodology for detection and estimation of software aging. In *Proc. 9th International Symposium on Software Reliability Engineering*, Paderborn, Germany, 1998.

[41] S. Garg, A. Puliafito, M. Telek, and K. Trivedi. Analysis of software rejuvenation using Markov regenerative stochastic Petri nets. In *Proc. 6th International Symposium on Software Reliability Engineering*, Toulouse, France, 1995.

[42] A. Gillen, D. Kusnetzki, and S. McLaron. The role of Linux in reducing the cost of enterprise computing. IDC Whitepaper, Jan. 2002.

[43] R. P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):34–45, June 1974.

[44] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. 12th ACM Symposium on Operating Systems Principles*, Litchfield Park, AZ, 1989.

[45] J. Gray. Notes on data base operating systems. In R. Bayer, R. M. Graham, J. H. Saltzer, and G. Seegmüller, editors, *Operating Systems, An Advanced Course*, volume 60, pages 393–481. Springer, 1978.

[46] J. Gray. The transaction concept: Virtues and limitations. In *Proc. International Conference on Very Large Data Bases*, Cannes, France, 1981.

[47] J. Gray. Why do computers stop and what can be done about it? In *Proc. 5th Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles, CA, 1986.

[48] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann, San Francisco, CA, 1993.

[49] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for Internet service construction. In *Proc. 4th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, Oct. 2000.

[50] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proc. 16th ACM Symposium on Operating Systems Principles*, Saint-Maló, France.

[51] C. Hawblitzel and T. von Eicken. Luna: A flexible Java protection system. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, 2002.

[52] J. Hennessy and D. Patterson. *Computer Architecture: A Quatitative Approach*. Morgan Kaufmann, San Francisco, CA, 3rd edition, 2002.

[53] Hewlett_Packard. Integrated and correlated enterprise management with the open management interface specification. HP OpenView whitepaper, `http://www.openview.hp.com`, 2002.

[54] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proc. Winter USENIX Technical Conference*, San Francisco, CA, 1994.

[55] Y. Huang and C. M. R. Kintala. Software implemented fault tolerance: Technologies and experience. In *Proc. 23rd International Symposium on Fault-Tolerant Computing*, Toulouse, France, 1993.

[56] Y. Huang, C. M. R. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: Analysis, module and applications. In *Proc. 25th International Symposium on Fault-Tolerant Computing*, Pasadena, CA, 1995.

[57] International_Business_Machines. IBM director software rejuvenation. White Paper, Jan 2001.

[58] International_Business_Machines. Tivoli monitoring resource model reference. Document Number SH19-4570-01, `http://www.tivoli.com`, 2002.

[59] D. Jacobs. Distributed computing with BEA WebLogic server. In *Proc. Conference on Innovative Data Systems Research*, Asilomar, CA, 2003.

[60] Z. T. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant. Chameleon: a software infrastructure for adaptive fault tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 10:560–579, 1999.

[61] R. W. Kembel. *The Fibre Channel Consultant: A Comprehensive Introduction*. Northwest Learning Associates, 1998. page 8.

[62] T. Lahiri, A. Ganesh, R. Weiss, and A. Joshi. Fast-Start: Quick fault recovery in Oracle. In *Proc. ACM International Conference on Management of Data*, Santa Barbara, CA, 2001.

[63] W. LeFebvre. CNN.com—Facing a world crisis. In *15th USENIX Systems Administration Conference*, 2001. Invited Talk.

[64] B. Ling and A. Fox. The case for a session state storage layer. In *Proc. 9th Workshop on Hot Topics in Operating Systems*, Lihue, Hawaii, 2003.

[65] D. E. Lowell, S. Chandra, and P. M. Chen. Exploring failure transparency and the limits of generic recovery. In *Proc. 4th USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, 2000.

[66] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *Proc. 16th ACM Symposium on Operating Systems Principles*, Saint-Maló, France, 1997.

[67] M. R. Lyu, editor. *Software Fault Tolerance*. John Wiley & Sons, New York, NY, 1995.

[68] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors—a survey. *IEEE Transactions on Computers*, 37(2):160–174, Feb 1988.

[69] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. In J. McCarthy and M. L. Minsky, editors, *Artificial Intelligence. Quarterly Progress Report No. 53*. MIT Research Lab of Electronics, Cambridge, MA, April 1959.

[70] J.-J. Miau and R. Holdaway, editors. *Reducing the Cost of Spacecraft Ground Systems and Operations*, volume 3. Kluwer Academic Publishers, 2000.

[71] Microsoft. *The Microsoft .NET Framework*. Microsoft Press, Redmond, WA, 2001.

[72] D. Milojicic, A. Messer, J. Shau, G. Fu, and A. Munoz. Increasing relevance of memory hardware errors. a case for recoverable programming models. In *Proc. ACM SIGOPS European Workshop*, Kolding, Denmark, 2000.

[73] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proc. 11th ACM Symposium on Operating Systems Principles*, Austin, TX, 1987.

[74] B. Murphy and N. Davies. System reliability and availability drivers of Tru64 UNIX. In *Proc. 29th International Symposium on Fault-Tolerant Computing*, Madison, WI, 1999. Tutorial.

[75] B. Murphy and T. Gent. Measuring system and software reliability using an automated data collection process. *Quality and Reliability Engineering International*, 11:341–353, 1995.

[76] K. Nagaraja, R. Bianchini, R. P. Martin, and T. D. Nguyen. Using fault model enforcement to improve availability. In *Proc. 2nd Workshop on Evaluating and Architecting System Dependability*, San Jose, CA, 2002.

[77] NOCPulse. Command center overview. `http://nocpulse.com`, 2002.

[78] U. G. A. Office. Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia. Technical Report GAO/IMTEC-92-26, U.S. G.A.O., 1992.

[79] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proceedings of the 1999 Summer USENIX Technical Conference*, Monterey, CA, June 1999.

[80] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do internet services fail, and what can be done about it? In *Proc. 4th USENIX Symposium on Internet Technologies and Systems*, Seattle, WA, 2003.

[81] A. Pal. Personal communication. Yahoo!, Inc., 2002.

[82] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, and N. Treuhaft. Recovery oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB/CSD-02-1175, UC Berkeley, Berkeley, CA, March 2002.

[83] V. Paxson. End-to-end routing behavior in the Internet. In *ACM SIGCOMM Conference*, Stanford, CA, 1996.

[84] D. K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1995.

[85] S. Raman and S. McCanne. A model, analysis, and protocol framework for soft state-based communication. In *ACM SIGCOMM Conference*, Cambridge, MA, 1999.

[86] G. Reeves. What really happened on Mars? RISKS-19.49, Jan. 1998.

[87] Resonate. Application performance management for business critical applications. `http://resonate.com`, 2002.

[88] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proc. 13th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, 1991.

[89] S. Saroiu, K. P. Gummadi, R. J. Dunn, S. D. Gribble, and H. M. Levy. An analysis of Internet content delivery systems. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, 2002.

[90] G. G. Schulmeyer and G. R. MacKenzie. *Verification and Validation of Modern Software-Intensive Systems*. Prentice Hall, Englewood Cliffs, NJ, 2000.

[91] D. Scott. Making smart investments to reduce unplanned downtime. Tactical Guidelines Research Note TG-07-4033, Gartner Group, Stamford, CT, March 19 1999. PRISM for Enterprise Operations.

[92] M. I. Seltzer and C. Small. Self-monitoring and self-adapting operating systems. In *Proc. 6th Workshop on Hot Topics in Operating Systems*, Cape Cod, MA, 1997.

[93] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems: Design and Evaluation*. AK Peters, Ltd., Natick, MA, 3rd edition, 1998.

[94] L. Spainhower. Why do systems fail? review studies 1993-1998. Presentation at IFIP WG 10.4 (Dependable Computing and Fault Tolerance). 41st Meeting. St. John, U.S. Virgin Islands, Jan. 2002.

[95] L. Spainhower and T. A. Gregg. IBM S/390 parallel enterprise server G5 fault tolerance: A historical perspective. *IBM Journal of Research and Development*, 43(5/6), 1999.

[96] M. Stonebraker. The design of the Postgres storage system. In *Proc. 13th Conference on Very Large Databases*, Brighton, England, 1987.

[97] Sun_Microsystems. J2EE platform specification. http://java.sun.com/j2ee/, 2002.

[98] M. A. Swartwout and R. J. Twiggs. SAPPHIRE - Stanford's first amateur satellite. In *Proceedings of the 1998 AMSAT-NA Symposium*, Vicksberg, MI, October 1998.

[99] B. Tuthill, K. Johnson, S. Wilkening, and D. Roe. *IRIX Checkpoint and Restart Operation Guide*. Silicon Graphics, Inc., Mountain View, CA, 1999.

[100] L. Wall and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., 1991.

[101] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. M. R. Kintala. Checkpointing and its applications. In *Proc. 25th International Symposium on Fault-Tolerant Computing*, 1995.

[102] K. Whisnant, R. Iyer, P. Hones, R. Some, and D. Rennels. Experimental evaluation of the REE SIFT environment for spaceborne applications. In *Proc. International Conference on Dependable Systems and Networks*, Washington, DC, 2002.

[103] A. Whitaker, M. Shaw, and S. Gribble. Scale and performance in the Denali isolation kernel. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, MA, 2002.

[104] R. J. Willett. Design of recovery strategies for a fault-tolerant no. 4 electronic switching system. *The Bell System Technical Journal*, 61(10):3019–3040, Dec 1982.

[105] N. Wirth. The programming language Oberon. *Software—Practice and Experience*, 18(7):671–690, 1988.

[106] W. Xie, H. Sun, Y. Cao, and K. Trivedi. Modeling of online service availability perceived by Web users. Technical report, Center for Advanced Computing and Communication (CACC), Duke University, 2002.

[107] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource Reservation Protocol. *IEEE Network*, 7(5), Sept. 1993.