Accepting Failure: Availability through Repair-centric System Design

Aaron Brown

Qualifying Exam Proposal

3 April 2001

Abstract

Motivated by the lack of rapid improvement in the availability of Internet server systems, we introduce a new philosophy for designing highly-available systems that better reflects the realities of the Internet service environment. Our approach, denoted repair-centric system design, is based on the belief that failures are inevitable in complex, human-administered systems, and thus we focus on detecting and repairing failures quickly and effectively, rather than just trying to avoid them. Our proposal is unique in that it tackles the human aspects of availability along with the traditional system aspects. We enumerate a set of design techniques for building repair-centric systems, outline a plan for implementing these techniques in an existing cluster email service application, and describe how we intend to quantitatively evaluate the availability gains achieved by repair-centric design.

1 Introduction and Motivation

Electronic information storage, processing, and dissemination have become an integral part of life in modern society. Businesses are built around the on-line delivery of information and content, and consumers are rapidly embracing the benefits of electronic communication and commerce. We have reached the point where many of our day-to-day activities—from e-mail communication to online banking to entertainment—depend on the proper functioning of a vast infrastructure of complex, large-scale computer systems.

The rapid expansion of this "Internet society" has put enormous pressures on the architects and operators of the computational infrastructure. Today's server systems must deliver the availability needed to meet user expectations for timely service that is always on, 24 hours a day, 7 days a week. When service availability falls short, the economic costs of lost business and customers are severe: one report estimates that typical per-hour outage costs range from the hundreds of thousands of dollars (for online retailers and airlines) to millions of dollars (for brokerages) [4] [56]. These economic costs come in addition to negative fallout from the publicity that accompanies outages at a high-profile services (such as the famous EBay outage of 1999 [41]). In the words of Kal Raman, CIO of drugstore.com, "availability is as important as breathing in and out [is] to human beings" [15].

With such an importance placed on availability, and such high penalties for system unavailability, one would expect to see rapid progress by industry and academia in addressing the availability challenge. Surprisingly, despite high-profile advertising by all major vendors purporting to offer high-availability solutions, and despite entire communities of researchers focusing on topics such as reliability, maintainability, and fault tolerance, outages and failures remain frequent. In a survey by InternetWeek, 65% of IT managers reported that their web sites suffered an outage at least once in the previous 6-month period; 25% reported three or more outages during that period [56].

Why has progress been so slow? A review of the literature and practice in the reliability, maintainability, and fault-tolerance communities reveals the problem: the designers of today's server systems operate under a set of perceptions that do not accurately reflect the realities of the modern Internet environment. Specifically, systems are commonly built and evaluated under the following three assumptions:

1. Failure rates of hardware and software components are low and improving;
2. Systems can be modeled for reliability analysis and failure modes can be predicted;
3. Humans do not make mistakes during maintenance.

Taken together, these assumptions result in a mindset that emphasizes failure avoidance as the way to improve availability. Our claim, however, is that the above assumptions are based on fundamentally incorrect perceptions of today's environment, and thus the mindset of availability-through-failure-avoidance is built on a weak foundation as well.

In response, we propose a new set of assumptions, ones that we believe are supported by practical experience with modern server systems:

1. Hardware and software will fail despite technological efforts to reduce failure rates;
2. Models of modern systems are inherently incomplete; failure modes cannot be predicted *a priori*;
3. Humans are *human*: fallible, inquisitive, experimental, biased, stubborn.

In contrast to the original set of assumptions, these result in a philosophy that emphasizes the importance of failure detection, recovery, and human-aware repair procedures. If failures are inevitable in today's complex, human-maintained server systems, then failure avoidance is inherently a losing battle, and only through improving detection and repair will we be able to make further gains in availability.

We will conduct a detailed analysis of these assumptions below in Section 1.2, presenting evidence that reveals the fallacies in today's assumptions and motivates our new set of assumptions. First, however, we will lay out the hypothesis of this proposal: that building systems around the philosophy of our new assumptions will result in a significant improvement in availability.

1.1 Hypothesis and Contributions

> *"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" — Shimon Peres [51]*

We claim that the slow progress toward significant improvements in availability is a result of focusing too much attention on avoiding failures rather than repairing them, on trying to increase mean-time-to-failure (MTTF) while ignoring mean-time-to-repair (MTTR)[1]. Following the guidelines of our new assumptions and their analysis below, we can deduce that failures will always occur despite the best efforts of system design and modeling. Drawing on the wisdom in the quotation by Shimon Peres that starts this section, we thus conclude that further significant gains in availability can only come once failures are accepted as a normal, unavoidable fact of system operation, and systems are designed with fast and effective repair mechanisms that mitigate the long-term impact of those failures. We call this philosophy *repair-centric design*.

We believe that repair-centric design can achieve and demonstrate significant improvements in availability by producing systems that:

1. expect faults to occur, and dedicate system resources to proactively detecting them
2. do not attempt to mask faults, but rather expose them to higher levels of the system so that they can be handled end-to-end and so that applications can participate in their repair
3. provide mechanisms that help diagnose new, unexpected failure modes and symptoms
4. continuously verify proper operation of repair and maintenance mechanisms
5. take into account human behavior by providing maintenance facilities that compensate for human error, allow for experimentation, and promote training of the administrator's mental system model.

After presenting evidence to motivate the new assumptions from which these rules are derived, the remainder of this proposal will describe the detailed mechanisms that we intend to use to implement, demonstrate, and evaluate these availability-enhancing techniques. Should our work be successful, we believe that we

1. Recall that availability is traditionally defined as the ratio of MTTF to MTTF+MTTR, that is, availability = MTTF/(MTTF+MTTR).

will have made the following contributions to the state-of-the-art in system availability research:

1. a new model for thinking about high availability based on a more accurate perception of the modern computing environment, with a focus on coping with failure rather than avoiding it
2. an elucidation of system design techniques that reflect this new model
3. an evaluation methodology to quantitatively demonstrate the improvements in availability enabled by this new model and its associated design techniques
4. a characterization of common system failure modes and maintainability tasks, used to drive this evaluation methodology
5. a prototype system implementation that demonstrates a quantitatively-measurable increase in availability in the face of realistic failures and maintenance procedures.

1.2 Analysis of Assumptions

We now turn to a detailed analysis of the assumptions underlying our new repair-centric design approach, debunking the assumptions underlying existing system design as we go.

1.2.1 Failure rates of hardware and software components

The first assumption apparent in current system designs is that the failure rates of hardware and software components are low, and still decreasing. While this may be true for many hardware components, it does not mean that failure rates will approach zero any time soon. For example, while the mean-time-to-failure ratings of disk drives have been rising precipitously in the past several years (and are often in the range of 100 or more years), disks are still designed with an 5-7 year expected lifetime of their mechanical components, a quantity not factored into MTTF [13]. Furthermore, even solid-state components with very low individual failure rates can show significant failure behavior when enough of them are used in the same system. For example, the production cluster run by the Google search engine experiences a node failure rate of 2-3% per year, with one-third of those failures attributable to the failure of a DRAM or memory bus [25]. The rate of failures due to memory may seem almost negligible—1% per year—except that Google's cluster has 6000+ nodes, and that 1% failure rate translates to more than one failure per week due to a solid-state component. The fact that the motherboards used in Google's nodes do not support ECC (a mechanism that could mitigate a large portion of those memory failures) shows how system designers turn a blind eye to failures that they incorrectly assume to be negligible.

When we turn our attention to software components, the assumption of low and decreasing failure rates looks even worse. While it is frequently acknowledged that all software has bugs that can induce unavailability, it is a common perception that these bugs can and will be eradicated over time with improved software design methodologies, debugging, and testing [33]. Evidence supporting this perception abounds; the ultimate example is the software used to control the Space Shuttle, which is essentially bug-free due to its slow development cycle and rigid development methodology [16]. However, this evidence relies on an assumption of stability: that software and development tools are allowed to mature, providing time for improved design and testing. This assumption is not true for most software deployed in today's Internet services world, where "there are so many people in the gold rush to get their applications online first," according to a division head of Sun's consulting wing. Major internet portals are deploying code written by gumshoe engineers with little more than a week of job experience [8]. In the words of Debra Chrapraty, former CIO of E*Trade, a major online brokerage service, "We used to have six months of development on a product and three months of testing. We don't live that way any more. . . . In Internet time, people get sloppy" [41].

In summary, blind adherence to this first assumption has resulted in system designs that minimize the importance of failures as an expected part of system operation, relegating failure detection, recovery, and repair to secondary importance, and often resulting in failure-handling mechanisms that are missing, inef-

fective, or inelegantly bolted-on as an afterthought. This has a direct effect on availability, as effective failure handling and fast repair are crucial factors in maintaining availability. Furthermore, even non-failed hardware and software components are frequently replaced and upgraded to newer and better versions; these upgrade procedures often have the same impact on a system as a failure of the concerned component. Our repair-centric design assumptions make it clear that hardware and software failures and upgrades will inevitably happen, and that therefore systems must be designed to tolerate and repair them.

1.2.2 Modeling of systems and failures

The usual approach to analyzing and improving system reliability is to construct a model of the system using one of an overflowing toolbox of modeling techniques—Markov chains, reliability block diagrams, fault trees, Petri nets, Bayesian belief networks, and so on [20]. These models are then used to predict the impact and propagation of faults through the system, allowing system reliability to be evaluated and improved. Although the literature of the fault-tolerance and reliability communities is saturated with research related to the generation and use of these models, it all makes one fundamental assumption: that the system structure, component failure modes, and component interactions are well-enough known that they can be specified in the form of a model.

In practice, however, large server systems are constructed from a combination of off-the-shelf and custom hardware and software components. To the designer, much of the system therefore appears as opaque black boxes with unquantifiable interactions, making the modeling task difficult or impossible. Furthermore, server systems are deployed and changed quickly, without the long observation periods needed to empirically measure component failure modes and distributions. Often, the assumption cannot even be made that system components will fail-fast; nearly all models require this assumption, whereas practical experience reveals it to rarely be true [57]. And as we have seen above, the false perception of increasing hardware and software reliability makes direct estimation of failure modes highly suspect.

Moreover, large servers are complex, tightly-coupled systems that perform a *transformational* function, consuming user requests, transforming databases, and synthesizing new results, all under the guidance of human maintainers. In the system taxonomy defined by sociologist and risk investigator Charles Perrow, these are exactly the kind of system that is highly susceptible to unexpected interactions [48]. Perrow's theories predict that such systems are by their very nature subject to "normal accidents": accidents (outages or failures in the case of servers) that arise from multiple and unexpected hidden interactions of smaller failures and the recovery systems designed to handle those failures. When viewed individually, normal accidents appear as very unlikely, rare situations arising from bizarre and improbable combinations of factors. Perrow's claim is that normal accidents are inevitable and unpredictable, despite the best attempts to model and compensate for failures.

Thus, adherence to this second assumption results in system designs where unmodeled failures are not expected, despite Perrow's arguments that they are likely to occur; as a result, provisions are rarely made for diagnosing and repairing such unexpected failures, which allows them to have a significant availability impact when they do occur. The second of our new set of repair-centric design assumptions makes Perrow's theories explicit: by assuming that we cannot predict failure modes *a priori*, we increase the chance that system designs can tolerate unexpected "normal accidents".

1.2.3 Human maintenance behavior

All large systems rely on human beings for maintenance and repair. At the very least, humans must perform the physical actions of repairing, replacing, and expanding hardware. Most systems require human intervention for software configuration and upgrading, and many require human intervention in the performance tuning loop. The task of diagnosing and fixing failures and other aberrant behavior is also a standard task of the human administrator.

One of the most pervasive assumptions underpinning the design of modern server systems is that humans do not make mistakes when performing these functions. This assumption is rarely stated explicitly,

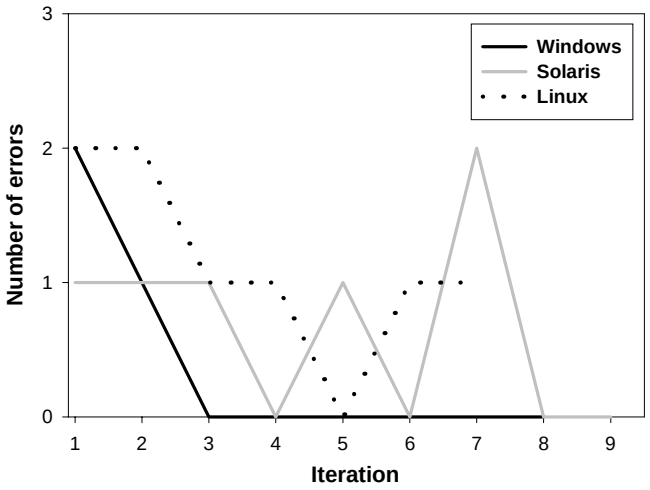


Figure 1: User errors made during simple maintenance task. The graph plots the number of errors made by five human subjects attempting to repair failed disks on each of three software RAID systems. On each iteration of the trial, each subject was required to repair one faulty disk on each system; plotted error counts are aggregated across the subjects.

but it can be inferred from several factors. First, the typical maintenance interface for a system provides no margin for error: if the wrong command is executed or the wrong piece of hardware is removed, there is often no recourse, and results can be catastrophic. Systems may provide guides for the human (such as lights on hardware and confirmation dialogs in software), but it is well-known in psychology that humans often construct a (frequently incorrect) mental model of the system, and that they will often ignore warnings and guides that conflict with their mental model [58]. Furthermore, the typical maintenance interface does not provide a way for human administrators to calibrate their mental models, for example by experimenting with different potential solutions to a problem—there is no easy way to try and back-out a particular solution.

If we turn to the standard literature, we find that practically all of the reliability modeling approaches treat human-performed repair operations as opaque procedures with a certain time distribution, but with no probability of causing further faults. This is a fundamental problem with characterizing repair simply by a time distribution or mean-time-to-repair (MTTR); the human may be modeled as a source of initial system faults, but repair itself is considered to be perfect.

Unfortunately, the assumption of human perfection is blatantly untrue. Psychologists tell us that human error rates are unquestionably non-zero, and can rise to 10% to 100% in stressful situations (as one might expect during off-hours emergency system maintenance) [49]. Humans make mistakes even in simple tasks such as digit recognition [30]; we can hardly expect them to do better when dealing with unwieldy maintenance interfaces to complex server systems. Experimental data collected during simple human repair experiments confirms this. Figure 1 reports on a study in which 5 trained subjects were asked to perform a simple maintenance task: replacing a failed disk in a software RAID volume on three different platforms [5]. As can be seen from the graph, errors were made on all platforms, and on two of the three, error rates did not significantly decrease with additional system familiarity.

Anecdotal reports paint an even bleaker picture. Of particular concern is that the problem of human operator error has been well known for a long time, yet there does not seem to have been any significant reduction in its impact over time. Data from the late 1970s reveals that operator error accounted for 50-70% of failures in electronic systems, 20-53% of missile system failures, and 60-70% of aircraft failures [11]. In the mid-1980s, a study of failures in fault-tolerant Tandem systems revealed that 42% were due to system administration errors—again human error [21]. Data collected on the causes of failures in VAX systems reveals that in 1993, human operators were responsible for more than 50% of failures, and that the error rate was rising as hardware and software failures become less important [45]. And more recently, in

1999, a Vice President at Oracle was reported as claiming that one "can have the best technology on the planet, but half the failures we analyze are human error" [41].

From the collective picture painted by all of this data, it is hard to deny that human-induced failures have remained a major and unaddressed problem in system availability. This motivates our third and final repair-centric design assumption, that humans are, in effect, *human*: prone to errors, inquisitive and experimental in their problem-solving approaches, and often stubborn in reconciling mental models with actual system behavior.

2 Proposed Techniques for Repair-centric System Design

In this section, we lay out our proposals for system design techniques that reflect the philosophy introduced in Section 1. Our discussion will remain at a high level in this section; proposed implementation details will be discussed below, in Section 3. After constraining the problem somewhat by making a few assumptions, we will consider each of the design points listed in Section 1.1 in turn.

2.1 System Assumptions

We begin with the assumption that our target system is a *service*: a single-function server system designed to supply a service to remote users that access the system over a network. This assumption greatly simplifies our task by removing the need to support arbitrary applications and user interactions with the system. Users are constrained to a fixed, external interface, and as long as we maintain that interface, we are free to construct the internals of the system in whatever special-purpose manner we desire.

We further assume that the service is deployed on a distributed cluster hardware architecture. Cluster designs are mandatory to meet the demands of scalability and incremental growth required by today's network services, and they also offer several properties that we will leverage to improve repair. In particular, clusters can easily be partitioned into multiple disjoint, smaller clusters to provide fault isolation, and their shared-nothing design allows for tolerance of partial failure.

Finally, we assume that the software implementing the service is constructed in a modular fashion, with well-defined components and interfaces between them. This assumption will prove critical in our fault detection and diagnosis techniques, as it allows for interface boundaries where testing and monitoring can be inserted. We assume modules can be distributed across the cluster. Many novel service infrastructures are being designed with modularity [18] [46], and even traditional multi-tier architectures can be seen as modular on a coarse granularity.

2.2 Design Techniques

2.2.1 Expecting failures

The first design point enumerated in Section 1.1 states that repair-centric systems should expect failures to occur, and dedicate system resources to proactively detecting them. In essence, we want to detect latent faults before they have the chance to manifest as failures, and likewise reduce the latency for detecting any failures that do slip through and manifest themselves.

One way to do this is to continuously probe all hardware and software modules of the system to verify their correct operation. We propose to do this by integrating online, proactive verification at all module interfaces in the system. Unlike the limited heartbeat-based testing typically used in fault-tolerant systems to verify module behavior, we propose to use injection of realistic test inputs into the system's regular interfaces. Such testing offers the ability to verify that a module is performing acceptably and delivering correct behavior at a semantic level, as opposed to simply verifying that it is reachable by the communications layer. Furthermore, by extending our testing to encompass both normal and non-destructive erroneous inputs, we can verify the correct behavior of both common-case and error-handling code paths.

In addition to this direct injection of test inputs, every module should, where possible, perform sanity checks on data received from other modules. In many cases, it is easier to check data against a specification than to transform it to match that specification; the canonical example is checking the order of purportedly-sorted data (vs. actually sorting it). More relevant examples might include checking email message headers for consistency with their associated bodies, or verifying the integrity of pointers in a data structure. We do not ask that modules be able to repair discovered inconsistencies, merely that they detect and report them. Where checking data based on a natural structure is impossible, checksums should be used as an ancillary method of verifying integrity [27].

So far we have only discussed verification at the module granularity. It is important as well to provide coarser-granularity, system-wide failure discovery by verifying consistency across modules. Of particular interest is a technique suggested by fault-detection strategies used in reliability-conscious industrial plant design [34]. The idea is to develop and use global "conservation laws" to ensure that data traveling through the system is not lost, delayed, corrupted, or fabricated, much as laws of fluid flow and energy conservation can be used to verify the integrity of plumbing and electrical systems. An example conservation law for an email service might be that the flux of incoming email messages should match the flux of messages and headers written to disk. We believe that conservation laws might also be extended to capture performance characteristics as well, allowing them to detect overflowing queues, wedged worker modules, and other bottlenecks.

Note that our online verification proposals offer unique fault-detection abilities beyond traditional off-line development-cycle testing, as they find the system in a production configuration under the system's actual workload. This allowing testing in realistic operational states that may never occur in a development lab, and may also provide a better ability to detect the effects of the nondeterministic Heisenbugs that escape traditional debugging procedures.

2.2.2 Exposing faults

Our second design point emphasizes that faults should be exposed, not masked. This is in some sense a religious argument. Nearly all extant reliable or fault-tolerant system designs attempt to mask faults to the greatest extent possible; the goal is to provide the illusion of perfectly-functioning hardware or software to higher layers of the system. Were it possible to completely mask faults, this would be a justified philosophy. But in practice, while faults can be masked from a functionality point of view, they cannot be completely masked in terms of performance. Moreover, many Internet service applications can tolerate some failures, and would rather operate with high performance on some non-failed subset of their data than suffer performance degradations or resource depletion due to fault-masking techniques [17].

Thus we propose that all modules in the system have a mechanism for reporting faults and "health" information to all of the other modules with which they communicate. This mechanism must be asynchronous so that faults can be reported immediately, but should also support polling for routine "checkups". Note that we are not advocating the elimination of fault-handling mechanisms such as automatic redundancy; instead, we are merely asking that the system keep higher levels informed of the existence of faults and repair actions, so that those higher levels can participate in the repairs. Again drawing on the example of an email server, the email application that receives an I/O failure response may want to distinguish a soft error while reading a MIME attachment from a hard error that scrambled the superblock of the email store; in the first case, recovery is trivial by simply returning the partial message to the user, while in the second case it would be prudent to switch to a backup copy of the repository and initiate a repair procedure.

2.2.3 Aiding diagnosis

The exposure of fault information discussed in the previous suggestion has another benefit besides offering applications a way to make tradeoffs upon faults. Fault and status information can be propagated up to a human being's monitoring station, providing the human administrator with detailed information that can be

used to troubleshoot a misbehaving system. We thus arrive at our third design point: that repair-centric systems should help in the difficult task of diagnosing new and unexpected failures.

Of course, the problem with simply propagating all status information to a human is that the data is overwhelming. Furthermore, as failures tend to propagate through systems, the human is likely to be overwhelmed with specious alarms when a failure occurs. The challenge is in *root-cause analysis*: identifying the subset of components truly at fault. One promising approach to root-cause analysis is a technique known as *dependency analysis*, in which a model of dependencies between system modules is used to identify possible culprits of observed symptoms while eliminating those modules that could not possibly be the root cause [6] [31]. Unfortunately, one of our main tenets of repair-centric systems is that *a priori* models (like the required dependency model) are not knowable for complex, dynamic systems.

The solution again lies in module interfaces. As a request is processed by the system, work is done on its behalf by the various system modules. We propose to associate a piece of state, a ticket, with each request. As that request filters through the system (possibly sitting in queues, possibly being merged with other requests via caching), its ticket is stamped by the each module it encounters. The stamp should include what resources (disks, networks, nodes, etc.) were used to fulfill the request. When the request is complete, the stamps on its ticket provide direct evidence of the relevant dependency chain; a request that fails or performs poorly returns with exactly the evidence needed to identify possible culprit modules. In some sense, as a diagnostic aid this technique is similar to log analysis, but it provides automatic correlation of logs across modules and resources, making the human diagnostician's job easier.

2.2.4 Continuously-verified repair and maintenance mechanisms

As repair-centric systems rely on repair and maintenance to handle failures and maintain availability, we expect that the mechanisms provided for these tasks should be efficient, reliable and trustworthy. Unfortunately, it is well-known that repair and maintenance code is some of the hardest code to write and test, and is therefore some of the most buggy. For example, Perrow's studies of several large industrial systems reveals that ineffective or broken warning systems, safety systems, and control systems are a major contribution to system accidents [48]. Furthermore, it is not unheard of to find shipping software products with significant bugs in their recovery code; an illustrative example is the software RAID-5 driver shipped with Solaris/x86, which improperly handles a double-disk-failure by returning fabricated garbage data in response to I/O requests [7]. Often times the bugs are in performance, not functionality; again choosing an example from software RAID, the Linux software RAID-5 driver puts such low priority on disk reconstruction that repair can take hours even with small volumes and moderate user loads [7].

Repair-centric systems must address the problem of buggy and ineffective maintenance code. The only way to do this is to exercise the maintenance code to expose bugs and allow them to be corrected before they are needed in an emergency. This must be done frequently and in the context of a deployed system in order to ensure that system reconfiguration, software aging, and varying system workload do not alter the effectiveness of maintenance mechanisms.

Note that this conceptually simple idea presents several practical implementation challenges that must be solved, including the selection of a fault workload to trigger the maintenance code, verification of the effectiveness of the maintenance mechanisms, and most importantly, containment of the tests and the injected faults so that they do not affect the production behavior of the system. We will discuss these problems and possible solutions in more detail when we consider our implementation of this technique, in Section 3.2.2.

2.2.5 Human-centric maintenance

Our last design point is that repair-centric systems must take human characteristics into account in the design of their administration and maintenance interfaces. We believe that there are two powerful techniques that could vastly improve resilience to human error, thereby reducing human-initiated failures and improving the reliability of repair and maintenance procedures, including hardware and software upgrades.

The first technique is conceptually simple: provide a way for any maintenance action performed on the system to be undone. Undo is a concept that is ubiquitous in computer systems, with the notable exception of system maintenance. Every word processor, spreadsheet, or graphics package worth its salt recognizes that humans try things out, change their minds, and make mistakes; they all provide an undo function. Yet, incomprehensibly, system designers seem to assume that system administrators are perfect, and thus fail to provide such functionality. An administrator who accidently replaces the wrong disk after a failure should be able to reinsert it without loss of data, but most RAID systems do not allow this [5]. It should be possible to back out a software upgrade gone bad with a single command, without loss of user data. An experiment in performance tuning should be instantly reversible in the case where it hurts more than it helps. The list goes on and on. Scenarios like these are common in practice; many major site outages are caused by errors made during upgrades or maintenance, errors not easily corrected because of the lack of undo [41].

Of course, implementing fully-general undo for all maintenance tasks is not an easy goal. In many cases, the resource requirements needed to provide undo may be prohibitive. But we believe that, for constrained problems such as the environment of a specific service, and for certain key maintenance tasks, undo is feasible, if expensive. A major contribution of this work will be discovering the extent to which undo can be provided for system maintenance.

Our second technique for reducing human error is to improve operator training. One of the biggest causes of human error during maintenance is unfamiliarity with the system; this is hardly surprising, since maintenance tasks (especially failure repair) are uncommon and often unique. Psychologists have shown that humans construct mental models of the operation of systems, then use those models to perform problem diagnosis and to predict the effects of potential actions; these models are empirical and can only be calibrated by experience with the system and its maintenance procedures [12] [30] [38] [58]. Practical experience bears out these theories: in one example, the current “uptime champion” of Tandem system installations is one in which operators are regularly trained on a separate pilot system before being allowed to interact with the production system [2]. The widespread use of simulators for training aircraft pilots in emergency-handling maneuvers is another example of where operator training has proven effective in reducing accidents and catastrophes.

We propose to build enforced training into the standard operating procedures of repair-centric systems. Repair-centric systems should regularly require their operators to carry out repair and maintenance procedures, using fault injection to generate scenarios requiring maintenance. Such enforced “fire-drills” train new operators and guarantee that veteran ones stay on their toes, and more importantly provide the opportunity for operators to gain experience with the system’s failure modes and maintenance procedures. By combining both targeted and random fault injection, they allow both training on known procedures as well as providing experience with diagnosing new and unseen problems. They further allow the system to evaluate its operators—in some sense, this technique is an extension of the continuous verification of Section 2.2.4 to include the human operator as a repair mechanism.

Of course, like the tests of automated recovery mechanisms described in Section 2.2.4, these operator training procedures must take place in a controlled environment that protects the rest of the system from the simulated failures and operator maintenance actions. Finally, it remains to be seen whether operators will tolerate such enforced training, especially if it is revealed as such. Open research issues remain as to the most effective frequency of such training sessions, and whether they should be announced to the operator.

2.2.6 Summary

Taken together, the five techniques described in the previous sections provide the foundation for increased availability via a focus on repair and an attention to human behavior. The verification approaches of Section 2.2.1 will help detect impending hardware failure, reproducible bugs in little-used error-handling code, and lurking Heisenbugs, allowing them to be repaired more quickly. When problems slip by the test-

ing infrastructure (most likely due to the unexpected interactions of a "normal accident"), the diagnosis techniques of Section 2.2.3 will help the human diagnostician find them faster, the reporting techniques of Section 2.2.2 will help the application bypass them, and the recovery-testing techniques of Section 2.2.4 will provide confidence that any automatic recovery mechanisms work properly. When the system requires human maintenance like upgrades, expansion, or fault repair, the proactive training techniques of Section 2.2.5 will help ensure that further errors are not made due to unfamiliarity with the system. At the same time, the undo mechanism of Section 2.2.5 will allow for experimentation with repair techniques while providing a backstop in case the upgrade or repair fails, or if catastrophic maintenance errors are made.

3 Proposed Implementation Plan

In the previous sections we have laid out a vision of a radical and highly ambitious approach to improving system availability that turns today's assumptions on their ear. We now must face the challenges of demonstrating and evaluating this novel approach to system design. Even if we constrain our demonstration to a single system, this task will not be easy. However, the potential benefits of its success are great: systems with vastly improved availability that reduce the maintenance burden on the human administrator and solve some of the most economically-important problems in the Internet technology sector. Thus we believe that it is possible and worthwhile to make the attempt. Our goal is to produce a prototype system that can demonstrate measurable improvements in availability in the face of realistic failure and maintenance situations, including hardware and software failures, upgrades, and system expansion. However, recognizing the possibility of unforeseen obstacles that may lay hidden on the path to this ultimate goal, we will present a staged implementation plan that prioritizes the repair-centric design techniques by impact and importance and provides an incremental evaluation methodology. We begin by selecting a base platform on which we will build our prototype repair-centric system implementation.

3.1 Target System

We plan to implement our techniques in the context of an Internet email service application. To reduce our implementation effort as much as possible, we will start with an existing system that already meets our system design assumptions and that has been constructed to already provide reasonably high availability under the traditional non-repair-centric design philosophy. We choose email as an application because it is a widely-used Internet service with hard-state storage requirements, relaxed semantics for user-visible interactions, and just enough complexity to make our implementation interesting without being overwhelming.

The existing email system that we plan to use as our base is the NinjaMail cluster email server [46]. NinjaMail is an email server supporting SMTP, POP, and IMAP mail protocols; it is built on top of the Ninja cluster-based Internet service infrastructure environment, a research environment currently being developed at Berkeley [23]. NinjaMail and the underlying Ninja infrastructure provide an ideal platform for experimentation with repair-centric design. First, NinjaMail meets the assumptions that we laid out in Section 2.1: it is a service application with well-defined external interfaces, it is designed to run on a shared-nothing cluster, and it is highly modular in design. NinjaMail (and the Ninja environment) are written in Java, a language that supports modularity, easy extension of interfaces, and has demonstrable productivity and reliability benefits. Furthermore, the Ninja environment provides a host of mechanisms that are needed for a repair-centric system design, including: intrinsic support for cluster partitioning, mandatory for experimentation with online fault-injection and administrator training; automatic management of worker processes, including built-in support for rolling software upgrades; a transactional file system that may prove useful for implementing undo; and asynchronous inter-module communication with intermediate queues, again potentially useful for undo. Finally, we have easy access to the expertise of the NinjaMail and Ninja developers.

We choose to start with an existing application rather than writing the system from scratch for several reasons. First, by using an existing implementation, we inherit implementations of the tedious-to-write but-

necessary pieces of the system, including external protocol processors, hard-state replica management algorithms, and request dispatchers. Ninja claims to have solved the difficulties of building scalable cluster network service applications; we can leverage their efforts rather than reinventing the wheel, leaving us free to focus primarily on the repair-centric aspects of the system. Second, an existing implementation provides us with a baseline comparison point against which to validate our progress in improving availability. Finally, a successful implementation of repair-centric design in an existing system speaks more for our techniques' generality than a one-off fully custom system design.

Of course, there are downsides to using an existing system. Paramount is the danger of being constrained by the existing design. We believe that NinjaMail is sufficiently lightweight that this will not be a problem; if it turns out to be, we can still reuse a good portion of the NinjaMail components while discarding the framework itself. There is also the danger of working with a good-sized research system, both in understanding the design and in keeping up-to-date with system developments. Indeed, NinjaMail is currently being rewritten to improve its functionality and code consistency. However, the NinjaMail developers are committed to generating a documented public release in the near future (early snapshots should be ready by the end of April 2001). We intend to take the public release as a base, and work with the NinjaMail developers to identify any future changes that need to be integrated.

3.2 Implementation Plan

As described above, we divide our implementation plan into stages, prioritized by the potential impact of each of the repair-centric design techniques. Since our NinjaMail target platform is still somewhat in a state of flux, we focus primarily on the issues we expect to arise during our implementation rather than technical implementation specifics.

3.2.1 Stage 1: Undo

The capability of undoing all maintenance actions, described above in Section 2.2.5, is probably the most powerful and fundamental of our repair-centric design techniques. Even in the absence of the other techniques, undo provides mechanisms for repairing human errors during repair and rolling back system changes that have lead to undesirable results. As such, it is the backstop for any repair-centric system, guaranteeing that repairs themselves cannot lead to further problems. It will thus be our first implementation focus.

Since our focus is on availability and repair, we want to be able to trust our undo mechanism, and thus there is no place for complex schemes that attempt to provide completely generic, fine-grained undo. For our prototype implementation in NinjaMail, then, we want to provide the simplest undo scheme that is sufficient to meet the goals we set out in Section 1.1. We therefore propose to implement two forms of undo, one to handle human errors in replacing cluster components, and the other to allow more generic undo of system state changes and software upgrades subject to restrictions on its time- and space-granularity.

The first form of undo is motivated by our earlier work in maintainability benchmarking, which revealed that humans are prone to accidentally replacing the wrong component after a failure, either by accident or due to a misunderstanding of the system's organization [5]. While it is easy for systems to detect when the wrong component has been replaced, they often do not bother, and furthermore even when incorrect replacement is detectable, it is often not recoverable. For example, our survey of software RAID systems revealed that none of the major implementations allow an incorrectly-removed disk to be reinserted without loss of data [5].

To address this, we propose an undo mechanism that allows incorrectly-removed components to be immediately replaced without loss of data. The mechanism is simple: all requests being sent to a particular hardware (or software) component in the system must be stored in a (perhaps non-volatile) queue; they are not removed from the queue until satisfied or acknowledged. When the component in question is accidentally removed, requests are preserved in the queue, and can be satisfied without loss when the component is reintegrated into the system. The Ninja environment's use of asynchronous queue-based communication

provides a good match to this undo implementation, although the use of queues may have to be extended to the lower-level devices. There are several potential complications with this scheme, most notably what to do with interrupted requests that are not idempotent. We do not believe this to be a serious problem for an email system like NinjaMail in which the application semantics are loose enough to tolerate duplication and reordering, and most of the removable hardware components in an email system like NinjaMail (disks, network links, etc.) are at least partially based on idempotent or repeatable requests.

Our second proposed undo implementation is more general, and is designed to address recovery from state changes such as software upgrades or system reconfiguration. To avoid the complexity of a fully-general undo, we propose a maintenance-undo with the following restricted semantics:

1. undo is available only during certain periods of system operation: it is automatically activated at the beginning of maintenance procedures and must be turned off once maintenance is successfully completed, permanently committing any changes made;
2. undo is coarse-grained in space: state is rolled back across the entire cluster, or an independent partition of the cluster;
3. undo is coarse-grained in time: state is rolled back to a particular snapshot of system state, not action by action. Snapshots are automatically taken at the start of maintenance actions, are nestable, and can also be manually initiated;
4. no attempt is made to distinguish state changes to user versus system data; user requests are logged and replayed once the system is rolled back to a snapshot.

These restricted semantics provide the ability to recover from arbitrary changes to the system, at the dual costs of having to manually define undo points (snapshots and commits) and to replay user requests after restoration to an earlier undo point; during the rollback-replay period, the system may provide an inconsistent or out-of-date view of user data. We believe that these restricted semantics are acceptable for an email system like NinjaMail. The manual definition of undo and commit points naturally correspond to the beginning and end of maintenance interactions with the system, and can be automatically identified by the system. Logging user requests is simple in an email system, as they consist of idempotent actions like receiving and deleting messages and updating message metadata. The potential inconsistencies and out-of-date views of the system are tolerable in an email application, where timeliness of email delivery is not mandatory and where existing protocols such as IMAP provide only limited consistency guarantees to mailbox views. We will have to evaluate the extent to which extra system resources are required to replay changes while still queuing new incoming requests.

To implement these semantics in NinjaMail, we need to implement snapshots and user request logging. User request logging is straightforward: we must simply duplicate and save the stream of incoming email when undo is active, and likewise log user requests for mailbox metadata manipulation. Snapshots are more complicated. They are basically system-wide checkpoints of hard state only; all worker modules in NinjaMail use soft-state and can simply be restarted once the hard-state of a checkpoint is restored. We see two possible implementation avenues for checkpointing system hard-state: either by copying state wholesale by reusing existing mechanisms in Ninja for node recovery, or modifying the Ninja hash table and transactional file system to support copy-on-write versioning; some of this support already exists in the Ninja file system. The low cost of physical storage today combined with the fact that undo is not active during normal system operation may allow us to use simpler but more expensive techniques like the first, but the final choice of mechanism is speculative at best at this point, and will require further investigation of Ninja's internal workings.

If taking frequent system snapshots turns out not to be prohibitively expensive in the NinjaMail system, we may also investigate relaxing the first restriction above (that undo is only available during maintenance). Recent work has shown that undo-like checkpointing mechanisms can prove useful in tolerating non-reproducible software bugs (Heisenbugs) if user-visible event histories need not be precisely preserved (as in email) [35]; thus, if our undo mechanism were available at all times during system operation,

it could be used to recover from non-human-initiated software failures as well as from human maintenance errors.

Finally, although this is not a focus of our work, note that the mechanisms that we develop to provide undo may be serendipitously useful for enhancing NinjaMail's maintenance functionality: snapshots provide on-line, easily-recovered backups, and can also be used as a data transfer mechanism for lazy remote mirroring.

3.2.2 Stage 2: Exercising maintenance mechanisms

With the undo mechanism in place to provide operators with a fail-safe recovery mechanism during maintenance and repair, the next stage of the implementation is to enable verification and improvement of the system's maintenance and repair mechanisms, so that repair is more efficient and the last-resort undo is less necessary. This is accomplished by implementing the verification technique described in Section 2.2.4 and the training technique described in Section 2.2.5.

These two techniques require a very similar set of mechanisms, and benefit from being implemented together. Both are based on the injection of synthetic faults into the system to trigger situations that require maintenance or repair; the goals of Section 2.2.4 are to verify the operation of automatic mechanisms designed to handle these situations, while the training techniques of Section 2.2.5 help the human operator develop the skills to handle those situations where the automatic mechanisms do not exist or have failed.

The natural synergies between automatic and human maintenance lead us to an implementation strategy in which we periodically subject the system to a broad range of targeted and random faults and allow its automatic maintenance mechanisms to attempt repair. In those cases where repair is unsuccessful, we keep records of the triggering faults and use them as training cases for the human operator, supplemented by additional targeted training cases for non-repair maintenance procedures such as system upgrades. The advantage of this approach is that it naturally self-tunes for the level of automatic maintenance provided by the system: a fully-self-maintaining system will not require much human maintenance, and thus there is little need to train the human operator. On the other hand, a more typical system will only be able to automatically handle a small subset of the possible system failure modes. In such a case, our proposed implementation will automatically compensate by increasing the amount of human training as it builds up a growing number of training cases.

To carry out this implementation, we need to solve several challenging problems: defining the injectable faults, verifying the system or human response to the fault, and most importantly, containing the tests and the injected faults so that they do not affect the production behavior of the system. Our work for the first task is minimal, since we can directly reuse the same real-world fault data set and fault-injection infrastructure described below in Section 4.1 for use in the evaluation of the implementation. For the second task, we should be able to verify many repair actions directly, especially for targeted faults (for example, a simulated disk failure is correctly repaired if the affected disk is removed and a different disk is inserted). For the random faults, we hope to appeal to eventually appeal to the testing infrastructure of stage 3 to verify restoration of proper system operation.

Finally, the last and most pernicious problem concerns fault containment. To ensure that any faults triggered by our fault injection do not permanently destroy important data or propagate into the part of the system not under test, we must isolate part of the system while the tests are being performed. Luckily, the NinjaMail environment offers several mechanisms that we can leverage. First, it intrinsically supports a notion of cluster partitions: requests can be temporarily directed away from a subset of the system's nodes, isolating them from user traffic. We can modify this mechanism to mirror requests to isolated partitions, providing them with workload while still maintaining isolation. Furthermore, the base Ninja environment provides the capability to remove and add nodes from the system, including the mechanisms to resynchronize data onto newly-added nodes. We intend to use these mechanisms to enable "rolling testing": automatically decommissioning a small subset of the cluster at a time, isolating it from the rest of the system via

network routing changes, and performing fault injection without worry of damaging the remaining part of the system still in production use.

There is still the concern that, during training, a well-intentioned human operator might err and make changes to the production partition of the system rather than the training partition. We think this can be avoided by making the production system training-aware and having it reject maintenance while training is in progress; we could also associate an undo window with training to further protect the production system.

3.2.3 Stage 3: Online verification

With the implementation of stage 2 completed, our prototype system should provide significantly improved reliability and efficacy of both automated and human-driven repair. The next step in our implementation plan improves the ability of the system to invoke those repair mechanisms when needed by emphasizing the anticipation of faults and their rapid discovery.

The implementation of online verification in NinjaMail follows directly from the outline sketched in Section 2.2.1, and from standard techniques in offline software testing such as input selection, result verification, and coverage analysis. Our implementation should be simplified by the existing modular structure of NinjaMail, as we can position our testing and verification code to simply interpose between module interfaces, rather than having to embed it directly into the modules themselves.

However, the fact that we are planning to introduce self-testing into an online, production environment adds a new twist. In particular, we need to ensure that our tests are non-destructive to user data (such as messages and mailbox indices). The simplest way to do this is to configure the tests to operate only on special test data, which may be derived by replicating existing user data. While this is probably appropriate for the simplest tests, much more discovery power is gained by testing on the real data. We believe that we can reuse stage 2's rolling testing mechanisms to do this, allowing potentially-destructive tests (such as using fully-random input) to be performed online without fear of losing data.

A second twist in our online self-testing implementation is that we want to detect dynamic performance problems with our tests as well as just testing correctness. This is important, since failures often manifest as performance faults rather than correctness faults, such as in the case of a failing disk [57]. Furthermore, performance faults often indicate that repair is needed to detect and remove system bottlenecks or to correct software misconfigurations. Our tentative approach to detecting performance faults is to maintain performance statistics for each targeted test. If a particular execution of a test takes longer than a statistically-defined maximum amount of time (computed, for example, as the 99.9% upper confidence bound on the previous test execution times), then we flag a performance fault. It remains to be seen if a statistically-robust measure can be defined that detects performance faults while not generating false alarms due to normal workload fluctuation.

We must also consider the means by which our built-in self-testing is invoked. Ultimately, an appropriate goal would be to have the system perform testing autonomously, automatically deciding when to invoke tests. Initially, though, we intend to simply expose the testing infrastructure to an external master control module, allowing us to experiment with different testing frequencies and also allowing us to invoke self-testing for specific verification tasks, for example to ensure proper functioning of a newly-replaced or upgraded component.

Finally, we are particularly interested in examining the utility of global conservation metrics as a system-wide error-detection mechanism; we identified an example law for email in Section 2.2.1, and hope to develop others after further study of the dynamics of the NinjaMail system. We expect to implement such checking by adding data-flow measurement points to system modules, and periodically polling them via an external master controller that implements the conservation-law analysis.

3.2.4 Stage 4: Diagnostic aids

The final stage of our implementation is to implement diagnostic aids that will help the human operator identify and correct new, unexpected faults in the system. This implementation stage covers the techniques

described in Section 2.2.2 and Section 2.2.3: explicit fault information collection, and dependency collection for root-cause analysis. Note that this stage is essentially orthogonal to the previous stages; it is an optimization to repair, rather than a fundamental change in how repair is viewed in system design.

The implementation of this fourth stage in the NinjaMail system follows directly from the discussions in Section 2.2.2 and Section 2.2.3. We want to modify the inter-module interfaces in the NinjaMail system to provide asynchronous notification of failures and to track the resources touched by requests as they travel through the system. The interface modifications themselves are straightforward. The challenge in the first task is to devise a representation for faults and status information that is general enough to capture module-specific failure information yet can still be used by all levels of the system. The simplest approach is to build a hierarchical representation that is specific to the structure of the NinjaMail system. A more general technique would be to distill all failure information into performability metrics that represent deviation from some baseline of performance, probability of data retrieval, consistency and timeliness of data, and so forth.

We must also determine if and how NinjaMail's internal components can make use of the propagated status information. For example, we believe we can modify NinjaMail to make use of failure information from its storage modules (along with the relaxed consistency guarantees of email) to allow it to provide partial service even in the case of failure. We also intend to collect the propagated fault and health information from the edges of the system and use it to produce status displays for the human administrator.

However, in order to make these status displays usable during failure situations, we need to implement the second half of stage 4, direct dependency collection by tracking resources touched by requests as they travel through the system. We believe that this is relatively simple in NinjaMail, as all communication within the system passes through a common layer that associates a queue with each module. It should be possible to use the sequence of queues touched by a request as an indication of the system dependencies associated with that request. The only obvious challenges here involve handling work is not associated with a request (solved by generating a pseudo-request for it), correlating work done on behalf of multiple requests (solved by careful bookkeeping), and integrating the dependency information with the health data in a useful manner to the human diagnostician (hopefully simplified by the constraints of a single-application service).

4 Evaluation Plan

Having laid out our four-stage implementation plan, we now describe how we intend to benchmark our progress. We have two approaches. First, we want to verify each stage's implementation and confirm the effectiveness of its techniques. These evaluations are like microbenchmarks of each stage. Second, we want to measure the availability of the entire system in order to evaluate the effectiveness of repair-centric design. These evaluations are like system macrobenchmarks, not directly evaluating the specific design techniques but illustrating their contributions to improved availability. Both the micro- and macro-benchmarks rely on some common techniques, which we will describe first.

4.1 Common techniques: workload, fault injection, fault model, and humans

All of our benchmarks require that the system be subjected to a realistic user workload. To do this, we intend to use the SPECmail2001 email benchmark [54]. This is an industry-standard benchmark for email systems that simulates a typical ISP-user email workload, both in terms of mail delivery and mail retrieval. In addition to providing workload, SPECmail measures email performance, allowing us to gauge the performance overhead introduced by our repair-centric modifications relative to the base NinjaMail system.

In order to evaluate our repair-centric design techniques we must be able to simulate failures and maintenance events without having to wait for them to occur naturally. To do this, we use fault injection to create failures and bring the system into states where maintenance is required. Accurate fault injection requires two things: a fault model that is capable of inducing a set of failures representative of what is seen in the real world, and a harness to inject those faults. In our previous work, we have defined fault models

for storage systems based on disk failures [7], but we will need a much broader fault model representative of failures in large network service systems in order to benchmark our email system. To get this model, we have undertaken to collect failure and maintenance data from a set of commercial sites; we plan to distill this data into set of injectable faults to use in our experiments.

Obtaining realistic fault and maintenance data is a major goal of this research, as it is central to our evaluation and also to the techniques to be implemented in stage 2. However, we do not yet have any of this data. Real-world failure data is sensitive, and companies are often unwilling to release it to researchers. We are currently working with several companies that have expressed some interest in sharing failure data, including Microsoft, IBM, and Hotmail, and we are committed to making at least one of these partnerships a success. However, in the case that all of our attempts fail, we will have no choice but to take the unsatisfactory path of conjuring up a set of injectable faults based on our own intuition and anecdotal evidence of system failure modes.

In addition to whatever realistic, targeted fault workload that we develop, we believe it is essential to also test with randomly-injected faults. While these tests may not simulate common-case failure situations, they are more likely to stimulate unexpected "Heisenbugs" or simulate the kinds of failures seen in "normal accidents"; they may capture the long tail in the distribution of possible system failures, after the targeted tests have covered the common case.

With a fault model in place, we need a harness to inject those faults. We believe that this harness should be built directly into our prototype email system. The reason for this choice is that an integrated fault-injection infrastructure is needed for stage 2 of our implementation anyway, and the work necessary to build it internally is likely less than that needed to bolt it on externally. To build this integrated infrastructure, we anticipate having to perform tasks such as modifying the low-level device drivers of our cluster to simulate observed hardware failure modes (including performance degradations); using Chen's software-fault-injection model [9] to simulate random bugs in NinjaMail's software modules; and modifying the Ninja communications infrastructure to simulate overloaded or slow modules.

Note that the one disadvantage of building-in the fault-injection infrastructure is that we will not be able to use it to measure the baseline availability of an unmodified NinjaMail system; our comparisons will have to be with a version of NinjaMail unmodified except for the introduction of the fault-injection infrastructure.

Finally, a note about human experiments. Many of our evaluation methodologies inescapably require participation by human subjects acting as system administrators and carrying out the human components of repair and maintenance. We intend to carry out these human-participation experiments using similar protocols as in our previous work on maintainability benchmarks [5], with computer-savvy subjects that are familiar with and trained on the details of system operation.

4.2 Microbenchmarks: testing and evaluating the stages

Stage 1: Undo. Recall that we defined two components to our undo implementation: the ability to disconnect and reconnect any system resource without loss of data, and a more general state-restoring maintenance-undo. To evaluate the first, we can carry out microbenchmarks in which we disconnect each removable component in the NinjaMail system and verify that it can be reattached without losing state, all while the system is under load. To evaluate the more general maintenance-undo, we will attempt to perform and undo each of the common maintenance operations identified in our study of real-world maintenance data (such as software and hardware upgrades). We will also test undoing combinations of maintenance operations. Note that fully evaluating the benefits of undo requires an availability benchmark, as described below.

Stage 2: Exercising maintenance mechanisms. Recall that stage 2 has two components: exercising and evaluating built-in recovery mechanisms, and training the system operators to reduce their error rates.

Evaluating the effectiveness of the first component of stage 2 is difficult since we are not actively developing recovery mechanisms in NinjaMail beyond its existing ones. It may be that our stage 2 implementation reveals problems with these existing mechanisms (especially as they are described as "flaky at best" by their authors [46]), which would directly demonstrate its effectiveness. If it does not, however, then we may be left with evaluating stage 2 on secondary criteria, such as how well it can verify the results of an automatic recovery, or how well it is able to cover the built-in repair mechanisms. We may have to purposely break existing recovery mechanisms in known ways to perform these analyses.

Evaluating the effectiveness of the second component of stage 2, training, is conceptually simple but requires a clever implementation. Effectively, we want to see if automatic training reduces human error rates, but we want to do so without having to carry out the kind of long-term, multi-week human study that would be needed to measure this directly. We need to compress the timescale somehow; one way to do this would be to divide our population of human subjects into groups and give each group a different amount of direct training with the system. This could be done by having each group perform a different number of maintenance tasks or diagnoses selected to match what the system would automatically present during training. After this variable amount of training, each group would then have to solve the same set of system problems and perform the same set of maintenance tasks; we would measure their error rates and their efficiency, allowing us to calculate the effect of the variable amounts of training on these metrics. The only limitation of this compressed-timescale approach is that we do not capture the effects of short- and long-term memory. While there are ways around this (for example, by spreading training sessions out over a longer time period), further consideration and possibly pilot experiments will be necessary to work out an appropriate training evaluation methodology.

Stage 3: Online verification. The goal of built-in online verification is to improve fault detection. A system's performance on this task can be measured by placing the system under a realistic workload, injecting faults from the realistic and random fault models, and measuring both the percentage of detected faults and the latency from fault injection to detection. We will compare these results to the same tests performed on the unmodified NinjaMail system.

Stage 4: Diagnostic aids. Two key metrics for diagnostic aids are correctness and selectivity. Correctness measures whether diagnostic aids correctly include the actual fault location as a possible culprit; selectivity measures how much can they reduce the space of possible fault culprits, To measure these quantities, we will inject targeted and random faults from our fault models into the system. To calculate correctness, we simply compute the ratio of the number of fault diagnoses including the true fault location to the total number of fault injects. To measure selectivity, we compute the ratio of the number of possible fault locations suggested by the dependency analysis model to the total number of potentially-faulty resources in the system. As with undo, fully evaluating the benefits of diagnostic aids requires an availability benchmark.

4.3 Macrobenchmark: Availability benchmarking

The stage-by-stage evaluation techniques described above will be useful for evaluating the effectiveness of our implementations of each of the repair-centric design techniques. The ultimate measure of our work, however, is a quantitative measurement of the availability improvement due to our repair-centric design philosophy. To take this measurement, we can perform *availability benchmarks* on both our modified and the baseline NinjaMail systems. Availability benchmarking is a technique we introduced and developed in previous work [7]. Note that the formulation of availability benchmarks that we describe here is a conglomeration of our original availability benchmarks [7] with our work on maintainability benchmarks [5]. The availability benchmarks that we describe below can be applied at any point during the implementation, and certainly when it is complete.

In a nutshell, an availability benchmark measures the deviations in the quality of service delivered by a system under a representative workload as realistic faults are injected into the system. We have already

identified a workload, fault model, and fault-injection harness in Section 4.1, above. We believe that several quality-of-service metrics are appropriate. For direct measurement of availability, we will use the deviation of the SPECmail2001 performance metric (messages/minute) from its normal range, and a measure of the system's error rate (lost or corrupted messages and mailboxes). As secondary metrics, we will also measure the amount of time spent by humans in administering the system, and the number of availability degradations resulting from human actions.

As indicated by our secondary metrics, our availability benchmarks will require human participation. This is unavoidable: several of our design points (undo, diagnostic aids, training) directly address human-computer interaction, and are not truly measurable without human experiments. The difficulty of performing these experiments may limit the number of system-wide availability benchmarks that we can carry out.

5 Relation to Related Work

Repair-centric design philosophy: The notion of repair-centric design is relatively novel in the systems and reliability communities, and there is very little existing work on designing systems that repair failures rather than trying to avoid them. One notable exception is in recent work on the design of frameworks for Internet services in which worker modules rely only on soft-state and are therefore restartable. In these schemes, such as the TACC framework described by Fox et al. [18] and the Ninja framework itself [46], application code implemented as worker modules is designed to be fully restartable at any time; all important hard state is kept elsewhere in the system and can be reloaded upon restart. This design acknowledges that application code can be buggy and provides fast restart and recovery mechanisms. More recent work has attempted to formalize the properties of such restartable systems and to devise the most appropriate ways to perform restart-based recovery [8] [28]. Furthermore, anecdotal reports suggest that these design techniques are used by production Internet services [8].

Our proposed work rests on the same philosophical underpinnings as this previous work, but goes beyond it in two ways. First, we include a focus on the sorely-neglected problem of human maintenance by providing techniques such as undo, root-cause-analysis, and training to assist in human diagnosis and repair of system problems as well as regular system maintenance. Second, we introduce the idea of online self-testing in the context of server systems, including testing at module interfaces as well as potentially-destructive tests of recovery mechanisms, with the goals of preempting catastrophic failures and reducing fault detection latency. To our knowledge, such potentially-destructive fault-injection-based testing has never been proposed for use in a running production server system, although some IBM mainframe designs have come close, as described below.

The human role in availability: The human operator's role in maintaining system availability has been well-understood for years. In 1975, Jack Goldberg told the fault-tolerance community that "fool tolerance" should be an important aspect of fault-tolerant system design [19]. In the late 1970s and early 1980s, much work was done on the analysis of the human's impact on diagnostics and maintenance in the context of military and industrial electronics systems; entire symposiums were dedicated to devising solutions to understanding human diagnostic behavior, improving human training, and avoiding human errors [50]. Even today, human operator error is considered in the design of industrial systems such as aircraft control systems [37] and train control systems [29], although as the latter work shows, designers often still neglect the impacts of stress and workload on human error rates.

Despite this earlier work, human error is almost entirely neglected in the work of the modern computer system design community. In the proceedings of the past several years of the main systems conferences, the Reliability and Maintainability Symposia, and the Fault-Tolerant Computing Symposia, only one author, Roy Maxion, has published on designing computer systems that take into account human error-proneness in order to provide availability or dependability. Maxion's work covers two forms of human maintenance error. In one paper, he identifies poor or unlocatable documentation as a major contributor to

human system maintenance errors that cause unavailability, defines an imprecise querying technique to help improve documentation effectiveness, then evaluates his technique in user studies, although he does not extrapolate these results into a prediction of improved availability [40]. In another, he identifies user-interface flaws as a contributor to human error, and provides a methodology for locating them [39]. While Maxion's work is motivated by the same general concerns as our proposed approaches to reducing the availability impact of human error, his targets are orthogonal to ours: we are concerned with reducing human errors by improving familiarity with existing interfaces through training and by providing a way to undo errors, rather than by trying to improve those existing interfaces to make training and undo less necessary. This distinction follows from our unique philosophy of trying to cope with problems such as poor user interface design rather than solving them outright.

Undo: While the concept of undo is found frequently in computer systems, particularly in productivity applications, to our knowledge it has never been applied to large-scale system maintenance tasks such as software upgrades or failure repair procedures. Perhaps the most similar idea is that of using file-system snapshots to provide a limited form of recovery from user errors such as accidental file deletions, as in the Network Appliance WAFL file system [26], but this is not a true undo in that it cannot reconstruct changes made to files since the last snapshot. Combining such an approach with logging, as is done in databases [44], would provide an undo facility similar to what we are proposing, but existing systems tend not to use such combinations of facilities to provide the capability of undoing large-scale system maintenance operations like hardware or software reconfigurations, repairs, or upgrades.

The techniques that we have proposed for our maintenance-undo, however, do follow directly from techniques used in other areas of computer system design. Checkpoints are a commonly-used mechanism for recovering from failures of long-running computations and database systems, and the idea of logging and replaying messages and user requests to roll a system forward from a checkpoint is also well-explored in the literature [3] [14] [35] [36] [44]. The work in checkpointing and rolling-forward entire applications has focused primarily on the challenges of transparently and consistently checkpointing application and operating system state and on performing distributed checkpoints. Our use of these techniques is somewhat different. First, our transparency requirements are limited, as the semantics of our email application allow some user-visible inconsistencies, and as we control the details of our service implementation. Furthermore, while undoing a botched maintenance operation is similar to recovering from an application failure, our maintenance-undo can take advantage of its restricted semantics (a defined time window for undo, coarse space- and time-granularities) to achieve a much simpler and less-performance-critical implementation than these generic checkpointing and roll-forward schemes. Finally, unlike a database system where checkpoints and logging are used to provide ACID semantics for transactions [22] [44], our email system will have little need for such strict semantics due to the properties of the application itself.

Online verification: Our online verification proposals break down into two categories: verification through passive checking, and active testing via input injection. Passive checking has been a mainstay of fault-tolerant system design since its inception, implemented either as multi-way component replication with comparison and voting logic [21] or as domain-specific external checking logic such as ECC logic in the memory system or parity prediction in state-machine design [52] [53]. Our goal is to extend such passive verification techniques, typically seen at only in expensive fault-tolerant hardware, into software modules in an Internet service environment.

The second category of online verification, on-line self-testing via input injection, is an extension of both traditional heartbeat protocols and built-in self-test (BIST) techniques. We have already discussed how our approach of using realistic inputs provides far greater testing power than heartbeats (in Section 2.2.1, above). BIST is a well-known fault-tolerant design technique that is used frequently in high-end fault-tolerant systems and mainframes, but historically it has been used offline, either during manufacturing or immediately before a system enters production use [55]. Recently, there has been more interest in on-line BIST in embedded safety controller systems (such as automotive ABS systems); work by Stein-

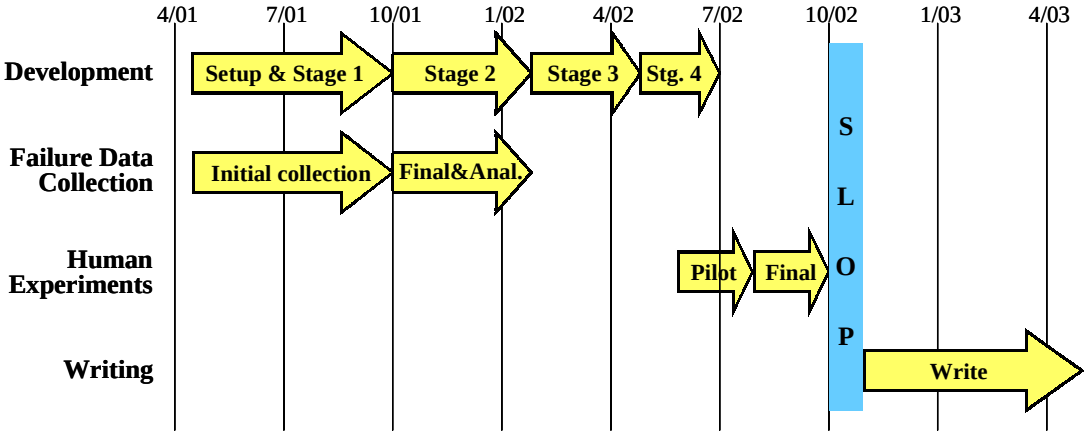
inger and Scherrer on an automotive steer-by-wire system model has demonstrated that on-line BIST can significantly improve overall availability in a redundant system design by detecting dormant faults in less-frequently-exercised modules and forcing components to fail silently before they have a chance to corrupt the system [55]. This is exactly what we hope to show in the context of a non-embedded server environment, using a practical implementation rather than a model-based calculation.

Integrated fault injection: Recall that we have proposed that systems be constructed with integrated fault-injection mechanisms as a way to dynamically test system recovery code and operator-driven repair actions. While we believe our proposal is unique in that we expect to perform these tests in an online manner, our notion of a built-in fault injection infrastructure is one that goes back in time all the way to the IBM 3090 and ES/9000 mainframes. These systems had highly-sophisticated built-in infrastructures for injecting faults into hardware data and control paths and for explicitly triggering recovery code [42]. These infrastructures were used for test-floor verification of the mainframes' recovery and serviceability mechanisms under production-like customer workloads, although as far as we know were not activated in field-deployed systems.

Diagnosis: Root-cause analysis is a well-known problem in system management, and many techniques have been developed to address it. One of the most interesting is the combination of monitoring, protocol augmentation, and cross-layer data correlation developed by Banga for use in diagnosing network problems in Network Appliance servers [1]. Unfortunately, this approach is system-specific, requiring custom code and expert protocol knowledge. A more generic approach is dependency analysis, which uses traversal-based techniques to subset potential root causes from an overall dependency graph based on observed symptoms and alarms [10] [24] [32] [59]. Dependency analysis requires a good dependency model, and unfortunately there has been little work on obtaining the needed dependency models from complex systems. The authors of the works referenced above simply assume the existence of dependency models, and the existing work on computing dependency models either assumes the unlikely existence of a system configuration repository [31], requires a compiler that can statically extract resource hierarchies [43], or uses perturbation experiments that are unlikely to work in the complex, asynchronous systems that underlie Internet services [6]. While our proposed direct approach to tracing request dependencies is similar to existing work that uses request-stamping to track resource utilization in distributed systems [47], we believe that its application as a diagnostic aid is unique and that it solves the problem of obtaining dependencies in cases where the system implementation can be modified.

6 Timeline

The goal is to complete the research described herein (including dissertation writing) by May 2003. An aggressive timeline for accomplishing this is as follows:



We recognize that this timeline may be optimistic given the scope of the proposed research, and we acknowledge the possibility that potential new graduate students in the project may want to assume some of the work in achieving our ultimate vision. We believe that the staged implementation plan described in Section 3.2 allows us to accommodate unforeseen delays and complications by eliminating stages or handing them off to other students.

As a bare minimum, we are committed to accomplishing the following: stages 1 and 2 of the NinjaMail implementation, the collection of real-world failure and maintenance data, and the human experiments needed to generate an evaluation of the overall availability improvements due to repair-centric design. We believe that this set of work still constitutes a coherent parcel of research, as it captures the most novel human aspects of the repair-centric techniques that we have proposed.

In addition to this bare minimum, we feel that also accomplishing stage 3 is a realistic goal even in the case of unforeseen complications. As stage 4 is somewhat orthogonal to the other 3 stages and can be developed in parallel, it is the obvious candidate to hand off should the need arise.

7 Conclusions

We have proposed a radical approach to high-availability system design that breaks from tradition by recognizing failures as inevitable and by providing mechanisms for efficiently detecting and repairing them. Our approach is unique in that it treats the human component of highly-available systems as inseparable from the system component and thus includes design techniques to simplify human maintenance tasks, compensate for human errors, and reduce human error rate. We believe that our human- and repair-centric approach will provide significant improvements in availability relative to standard fault-tolerance techniques, especially in the dynamic environment of Internet service applications, and we hope to demonstrate those improvements quantitatively in our extensions to the existing NinjaMail email service application. If successful, we believe that this work and its new human- and repair-centric design paradigms will comprise a significant contribution to the state-of-the-art in system availability research.

References

[1] G. Banga. Auto-diagnosis of Field Problems in an Appliance Operating System. *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[2] W. Bartlett. Personal communication, March 2001.

[3] A. Borg, W. Blau, W. Graetsch et al. Fault Tolerance Under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[4] J. B. Bowles, J. G. Dobbins. High-Availability Transaction Processing: Practical Experience in Availability Modeling and Analysis. *Proceedings of the 1999 Annual Reliability and Maintainability Symposium*, 268–273, 1999.

[5] A. Brown. Towards Availability and Maintainability Benchmarks: A Case Study of Software RAID Systems. *UC Berkeley Computer Science Division Technical Report UCB//CSD-01-1132*, Berkeley, CA, January 2001.

[6] A. Brown, G. Kar, and A. Keller. An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment. To appear in *Proceedings of the Seventh IFIP/IEEE International Symposium on Integrated Network Management (IM VII)*, Seattle, WA, May 2001.

[7] A. Brown and D. A. Patterson. Towards Availability Benchmarks: A Case Study of Software RAID Systems. *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[8] G. Candea and A. Fox. Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel. *Submission to the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII).*

[9] P. M. Chen, W. T. Ng, S. Chandra et al. The Rio File Cache: Surviving Operating System Crashes. *Proceedings of the 1996 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1996.

[10] J. Choi, M. Choi, and S. Lee. An Alarm Correlation and Fault Identification Scheme Based on OSI Managed Object Classes. *1999 IEEE International Conference on Communications*, Vancouver, BC, Canada, 1999, 1547–51.

[11] J. M. Christensen and J. M. Howard. Field Experience in Maintenance. In [50], 111–133.

[12] K. D. Duncan. Training for Fault Diagnosis in an Industrial Process Plant. In [50], 553–574.

[13] J. G. Elerath. Specifying Reliability in the Disk Drive Industry: No More MTBF's. *Proceedings of the 2000 Annual Reliability and Maintainability Symposium*, 194–199, 2000.

[14] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *CMU Technical Report CMU TR 96-181*, Carnegie Mellon University, 1996.

[15] S. Fisher. E-business redefines infrastructure needs. *InfoWorld*, 7 January 2000. Available from www.infoworld.com.

[16] C. Fishman. They Write the Right Stuff. *FastCompany Magazine*, 6:95ff, December 1996. Available at http://www.fastcompany.com/online/06/writestuff.html.

[17] A. Fox and E. Brewer. Harvest, Yield, and Scalable Tolerant Systems. *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, AZ, March 1999.

[18] A. Fox, S. Gribble, Y. Chawathe et al. Cluster-based Scalable Network Services. *Proceedings of the 16th Symposium on Operating System Principles (SOSP-16)*, St. Malo, France, October 1997.

[19] J. Goldberg. New Problems in Fault-Tolerant Computing. *Proceedings of the 1975 International Symposium on Fault-Tolerant Computing*, 29–34, Paris, France, June 1975.

[20] A. Goyal, S. S. Lavenberg, and K. S. Trivedi. Probabilistic modeling of computer system availability. *Annals of Operations Research*, 8(1-4):285–306, March 1987.

[21] J. Gray. Why Do Computers Stop and What Can Be Done About It? *Symposium on Reliability in Distributed Software and Database Systems*, 3–12, 1986.

[22] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. San Francisco: Morgan-Kauffmann, 1993.

[23] S. Gribble, M. Welsh, R. Von Behren et al. The Ninja architecture for robust Internet-scale systems and services. *Computer Networks* 35(4):473–497.

[24] B. Gruschke. Integrated Event Management: Event Correlation Using Dependency Graphs. *Proceedings of 9th IFIP/IEEE International Workshop on Distributed Systems Operation & Management (DSOM98)*, 1998.

[25] J. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 3e (beta version)*. San Francisco: Morgan-Kauffmann, 2001.

[26] D. Hitz, J. Lau, M. Malcolm. File System Design for an NFS Server Appliance. *Network Appliance Technical Report TR3002*, March 1995.

[27] J. Howard, R. Berube, A. Brown et al. A Protocol-centric Design for Architecting Large Storage Systems. *Submission to the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*.

[28] Y. Huang, C. Kintala, N. Kolettis et al. Software Rejuvenation: Analysis, Module and Applications. *Proceedings of the 1995 International Symposium on Fault-Tolerant Computing*, 381–390, 1995.

[29] V. Joshi, L. Kaufman, and T. Giras. Human Behavior Modeling in Train Control Systems. *Proceedings of the 2001 Annual Reliability and Maintainability Symposium*. Philadelphia, PA, 2001.

[30] B. H. Kantowitz and R. D. Sorkin. *Human Factors: Understanding People-System Relationships*. New York: Wiley, 1983.

[31] G. Kar, A. Keller, and S. Calo. Managing Application Services over Service Provider Networks: Architecture and Dependency Analysis. *Proceedings of the Seventh IEEE/IFIP Network Operations and Management Symposium (NOMS 2000)*, Honolulu, HI, 2000.

[32] S. Kätker and M. Paterok. Fault Isolation and Event Correlation for Integrated Fault Management. *Fifth IFIP/IEEE International Symposium on Integrated Network Management (IM V)*, San Diego, CA, 1997, 583–596.

[33] S. Keene, C. Lane, J. Kimm. Developing Reliable Software. *Proceedings of the 1996 Annual Reliability and Maintainability Symposium*, 143–147, 1996.

[34] M. Lind. The Use of Flow Models for Automated Plant Diagnosis. In [50], 411–432.

[35] D. E. Lowell, S. Chandra, and P. Chen. Exploring Failure Transparency and the Limits of Generic Recovery. *Proceedings of the 4th Symposium on Operating System Design and Implementation.* San Diego, CA, October 2000.

[36] D. E. Lowell, P. M. Chen. Discount Checking: Transparent, Low-Overhead Recovery for General Applications. *University of Michigan Technical Report CSE-TR-410-99,* November 1998.

[37] W. E. MacKay. Is paper safer? The role of paper flight strips in air traffic control. *ACM Transactions on Computer-Human Interaction*, 6(4):341–369, December 1999.

[38] E.C. Marshall and A. Shepherd. A Fault-Finding Training Programme for Continuous Plant Operators. In [50], 575–588.

[39] R. Maxion and A. deChambeau. Dependability at the User Interface. *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, 528–535, 1995.

[40] R. Maxion and P. Syme. Mitigating Operator-Induced Unavailability by Matching Imprecise Queries. *Proceedings of the 26th International Symposium on Fault-Tolerant Computing*, 240–249, 1996.

[41] J. Menn. Prevention of Online Crashes is No Easy Fix. *Los Angeles Times,* 2 December 1999, C-1.

[42] A. C. Merenda and E. Merenda. Recovery/Serviceability System Test Improvements for the IBM ES/9000 520 Based Models. *Proceedings of the 1992 International Symposium on Fault-Tolerant Computing*, 463–467, 1992.

[43] B. Miller, M. Callaghan et al. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer* 28(11):37–46, November 1995.

[44] C. Mohan, D. Haderle, B. Lindsay et al. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1): 94–162, 1992.

[45] 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.

[46] Ninja: A Framework for Network Services. *Submission to the 18th Symposium on Operating System Principles (SOSP)*.

[47] J. Reumann, A. Mehra, K. Shin et al. Virtual Services: A New Abstraction for Server Consolidation. *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[48] C. Perrow. *Normal Accidents: Living with High-Risk Technologies*. Princeton, NJ: Princeton Press, 1999.

[49] R. H. Pope. Human Performance: What Improvement from Human Reliability Assessment. *Reliability Data Collection and Use in Risk and Availability Assessment: Proceedings of the 5th EureData Conference*, H.-J. Wingender (Ed.). Berlin: Springer-Verlag, April 1986, 455–465.

[50] J. Rasmussen and W. Rouse, eds. *Human Detection and Diagnosis of System Failures: Proceedings of the NATO Symposium on Human Detection and Diagnosis of System Failures.* New York: Plenum Press, 1981.

[51] D. Rumsfeld. Rumsfeld's Rules: Advice on government, business and life. *The Wall Street Journal Manager's Journal*, 29 January 2001.

[52] L. Spainhower and T. Gregg. G4: A Fault-Tolerant CMOS Mainframe. *Proceedings of the 1998 International Symposium on Fault-Tolerant Computing*, 432–440, 1998.

[53] L. Spainhower, J. Isenberg, R. Chillarege et al. Design for Fault-Tolerance in System ES/9000 Model 900. *Proceedings of the 1992 International Symposium on Fault-Tolerant Computing*, 38–47, 1992.

[54] SPEC, Inc. *SPECmail2001*. http://www.spec.org/osg/mail2001/.

[55] A. Steininger and C. Scherrer. On the Necessity of On-line-BIST in Safety-Critical Applications—A Case-Study. *Proceedings of the 1999 International Symposium on Fault-Tolerant Computing*, 208–215, 1999.

[56] T. Sweeney. No Time for DOWNTIME—IT Managers feel the heat to prevent outages that can cost millions of dollars. *InternetWeek*, n. 807, 3 April 2000.

[57] N. Talagala. Characterizing Large Storage Systems: Error Behavior and Performance Benchmarks. *Ph.D. Dissertation*, U.C. Berkeley, September 1999.

[58] C.D. Wickens and C. Kessel. Failure Detection in Dynamic Systems. In [50], 155–169.

[59] S. Yemini, S. Kliger et al. High Speed and Robust Event Correlation. *IEEE Communications Magazine*, 34(5):82–90, May 1996.