Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies

David Patterson, Aaron Brown, Pete Broadwell, George Candea , Mike Chen, James Cutler , Patricia Enriquez*, Armando Fox⁼, Emre Kiciman , Matthew Merzbacher*, David Oppenheimer, Naveen Sastry, William Tetzlaff , Jonathan Traupman, and Noah Treuhaft

Computer Science Division, University of California at Berkeley (unless noted)
*Computer Science Department, Mills College
Computer Science Department, Stanford University
IBM Research, Almaden

Contact Author: David A. Patterson, patterson@cs.berkeley.edu

Abstract

It is time to declare victory for our performance-oriented research agenda. Four orders of magnitude increase in performance since the first ASPLOS means that few outside the CS&E research community believe that speed is *the* problem of computer hardware and software. Current systems crash and freeze so frequently that people become violent.[1] Faster flakiness should not be the legacy of the 21st century.

Recovery Oriented Computing (ROC) takes the perspective that hardware faults, software bugs, and operator errors are facts to be coped with, not problems to be solved. By concentrating on Mean Time to Repair (MTTR) rather than Mean Time to Failure (MTTF), ROC reduces time to recover from these facts and thus offer higher availability. Since a large portion of system administration is dealing with failures, ROC may also reduce total cost of ownership. One to two orders of magnitude reduction in cost over the last 20 years mean that the purchase price of hardware and software is now a small part of the total cost of ownership.

In addition to giving the motivation, definition, and techniques of ROC, we introduce quantitative failure data for Internet sites and the public telephone system, which suggest that operator error is a leading cause of outages. We also present results of using six ROC techniques in five case studies: hardware partitioning and fault insertion in a custom cluster; software fault insertion via a library, which shows a lack of grace when applications face faults; automated diagnosis of faults in J2EE routines without analyzing software structure beforehand; a fivefold reduction in time to recover a satellite ground station's software by using fine-grained partial restart; and design of an email service that supports undo by the operator.

If we embrace availability and maintainability, systems of the future may compete on recovery performance rather than SPEC performance, and on total cost of ownership rather than system price. Such a change in milestones may restore our pride in the architectures and operating systems we craft.[2]

(Target: 6000 words total, about 20 double spaced pages. Now at 10300 words, 25 pages, with 73 refs = 3 pages. Note 2200 words and 6 pages in title, abstract, captions, figures, tables, footnotes, references.)

---

[1] A Mori survey for Abbey National in Britain found that more than one in eight have seen their colleagues bully the IT department when things go wrong, while a quarter of under 25 year olds have seen peers kicking their computers. Some 2% claimed to have actually hit the person next to them in their frustration. Helen Petrie, professor of human computer interaction at London's City University. says "There are two phases to Net rage - it starts in the mind then becomes physical, with shaking, eyes dilating, sweating, and increased heart rate.You are preparing to have a fight, with no one to fight against." From *Net effect of computer rage*, by Mark Hughes-Morgan, Associated Press, February 25, 2002.

[2] Dear Reviewer: Similar to the OceanStore paper at the last ASPLOS, this paper is early in the project, but lays out the perspectives and has initial results to demonstrate importance and plausibility of these potentially controversial ideas. We note that the Call for Papers says *''New-idea'' papers are encouraged; the program committee recognizes that such papers may contain a significantly less thorough evaluation than papers in more established areas. The committee will also give special consideration to controversial papers that stimulate interesting debate during the committee meeting.* We hope our novel and controversial perspective can offset the lack of performance measurements on a single, integrated prototype.

1. Motivation

The main focus of researchers and developers for the 20 years since the first ASPLOS conference has been performance, and that single-minded effort has yielded a 12,000X improvement [HP02]. Key to this success has been benchmarks, which measure progress and reward the winners. Benchmarks let developers measure and enhance their designs, help customers fairly evaluate new products, allow researchers to measure new ideas, and aid publication of research by helping reviewers to evaluate it.

Not surprisingly, this single-minded focus on performance has neglected other aspects of computing: dependability, security, privacy, and total cost of ownership, to name a few. For example, the cost of ownership is widely reported to be 5 to 10 times the cost of the hardware and software. Figure 1 shows the same ratios for Linux and UNIX systems: the average of UNIX operating systems on RISC hardware is 3:1 to 15:1, while Linux on 80x86 rises to 7:1 to 19:1.

Such results are easy to explain in retrospect. Faster processors and bigger memories mean more users on these systems, and it s likely that system administration cost is more a function of the number of users than of the price of system. Several trends have lowered the purchase price of hardware and software: Moore s Law, commodity PC hardware, clusters, and open source software. In addition, system administrator salaries have increased while prices have dropped, inevitably leading to hardware and software in 2002 being a small fraction of the total cost of ownership.

The single-minded focus on performance has also affected availability, and the cost of unavailability. Despite marketing campaigns promising 99.999% of availability, well managed servers today achieve 99.9% to 99%, or 8 to 80 hours of downtime per year. Each hour can be costly, from $200,000 per hour for an ISP

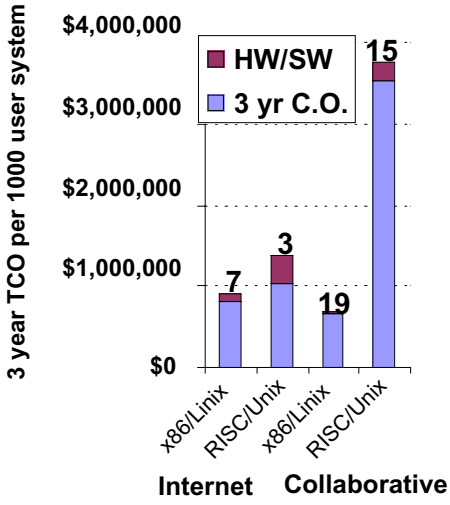


Figure 1. Ratio of Three Year Cost of Ownership to Hardware/Software Purchase Cost for x86/Linix and RISC/Unix systems [Gillen 2002]. These results are normalized to the Total Cost of Ownsership per thousand users. To collect this data, in the second half of 2001 IDC interviewed 142 companies. Note that several costs typically associated with ownership were *not* included: space, power, media, communications, HW/SW support contracts, and downtime. The companies had average sales of $2.4B/year, and sites had 3 to 12 servers supporting 1100 to 7600 users/site. The sites were divided into two services: Internet/Intranet (firewall ,Web serving, Web caching, B2B, B2C) and Collaborative (calendar, email, shared file, shared database.)

like Amazon to $6,000,000 per hour for a stock brokerage firm [Kembe00].

The reasons for failure are not what you might think. Figure 2 shows the failures of the Public Switched Telephone Network. Operators are responsible for about 60% of the problems, with hardware at about 20%, software about 10%, and overloaded telephone lines about another 10%.

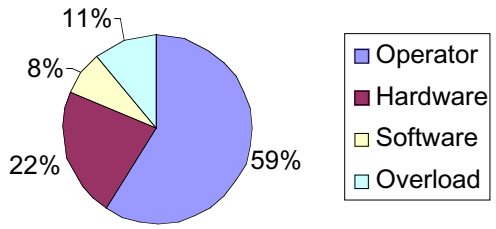


Figure 2. Percentage of blocked calls in 2000 by cause: human, hardware, software, and overload. Representing over 200 telephone outages in the U.S. that affected at least 30,000 customers or lasted 30 minutes, as required by FCC. Rather than report outages, telephone switches record the number of attempted calls blocked during an outage, which is an attractive measure of failure.

Table 1 shows percentages of outages for three Internet services: a High Traffic site, an Online Services site, and a Global Content site. These measures show that operators are again leading causes of outages, with the troubled tiers being the front end, with its large fraction of resources, or the network, with its distributed nature and its difficulty to diagnosis; note almost all the unknown failures are associated with the network.

	Front-end			Network			Backend			Unknown		Total		
Human	42%	25%	4%	4%	8%		8%	8%	8%	4%		50%	38%	25%
Hardware				8%	17%		8%					17%		17%
Software	17%	17%			25%		8%	8%				25%	25%	25%
Environment		4%											4%	
Unknown		8%		8%	21%	33%				4%		8%	33%	33%
Total	58%	54%	4%	17%	25%	83%	25%	17%	8%	4%	4%	100%	100%	100%

Table 1. Percentage Failures for Three Internet sites, by type and tier. The three sites are an Online Service site, and a Global Content, and a Read Mostly site. (Failed data was shared only is we assured anonymity.) All three services use multi-tiered systems with geographic distribution over a WAN to enhance service availability. The number of computers varies from about 500 for the Online Service to 5000 for the Read Mostly site. Only 20% of the nodes are in the front end of the Content site, with 99% of the nodes in the front ends of the other two. Collected in 2001, these data represent six weeks to six months of service.

We are not alone in calling for new challenges. Jim Gray [1999] called for *Trouble-Free Systems*, which can largely manage themselves while providing a service for millions of people. Butler Lampson [1999] called for systems that *work*: they meet their specs, are always available, adapt to changing environment, evolve while they run, and grow without practical limit. Hennessy [1999] proposed the new target to be Availability, Maintainability, and Scalability. IBM Research [2001] recently announced a new push in Autonomic Computing, whereby they try to make systems smarter about managing themselves rather than just faster. Finally, Bill Gates [2002] set *trustworthy systems* as the new target for his operating system developers, which means improved security, availability, and privacy.

The Recovery Oriented Computing (ROC) project presents one perspective on how to achieve the goals of these luminaries. Our target is services over the network, including both Internet services like Yahoo and Enterprise services like corporate email. The killer metrics for such services are availability and total cost of ownership, with Internet services also challenged by rapid scale-up in demand and deployment and rapid change of software.

Section 2 of this paper surveys other fields, from disaster analysis to civil engineering, to look for ideas to guide the design of such systems. Section 3 presents the ROC hypotheses of concentrating on recovery to make systems more dependable and less expensive to own. Section 4 lists six techniques we have identified to guide ROC. Section 5, the bulk of the paper, shows five cases we have created to help evaluate these principles. Section 6 describes related work, and Section 7 concludes with a discussion and future directions for ROC.

2. Inspiration From Other Fields

Since current systems are fast but fail prone, we thought we'd try to learn from other fields for new directions and ideas. They are disaster analysis, human error analysis, and civil engineering design.

2.1 Disasters and Latent Errors in Emergency Systems

Charles Perrow [1990] analyzed disasters, such as the one at the nuclear reactor on Three Mile Island (TMI) in Pennsylvania in 1979. To try to prevent disasters, nuclear reactors are redundant and rely heavily on "defense in depth," meaning multiple layers of redundant systems.

Reactors are large, complex, tightly coupled systems with lots of interactions, so it's very hard for operators to understand the state of the system, its behavior, or the potential impact of their actions. There are also errors in implementation and in the measurement and warning systems which exacerbate the situation. Perrow points out that in tightly coupled complex systems bad things will happen, which he calls *normal accidents*. He says seemingly impossible multiple failures--which computer scientists normally disregard as statistically impossible--do happen. To some extent, these are correlated errors, but latent errors also accumulate in a system awaiting a triggering event.

He also points out that the emergency systems are often flawed. Since unneeded for day-to-day operation, only an emergency tests them, and latent errors in the emergency systems can render them useless. At TMI, two emergency feedwater systems had the corresponding valve in each system next to

each other, and they were manually set to the wrong position. When the emergency occurred, these backup systems failed. Ultimately, the containment building itself was the last defense, and they finally did get enough water to cool the reactor. However, in breaching several levels of defense in depth, the core was destroyed.

Perrow says operators are blamed for disasters 60% to 80% of the time, including TMI. However, he believes that this number is much too high. The postmortem is typically done by the people who designed the system, where hindsight is used to determine what the operators really should have done. He believes that most of the problems are designed in. Since there are limits to how much you can eliminate in the design, there must be other means to mitigate the effects when "normal accidents" occur.

Our lessons from TMI are the importance of removing latent errors, the need for testing recovery systems to ensure that they will work, and the need to help operators cope with complexity.

2.2 Human Error and Automation Irony

Because of TMI, researchers began to look at why humans make errors. James Reason [1990] surveys the literature of that field and makes some interesting points. First, there are two kinds of human error. *Slips* or *lapses*--errors in execution--where people don't do what they intended to do, and *mistakes*--errors in planning--where people do what they intended to do, but did the wrong thing. The second point is that training can be characterized as creating mental production rules to solve problems, and normally what we do is rapidly go through production rules until we find a plausible match. Thus, humans are furious pattern matchers. Reason s third point is that we are poor at solving from first principles, and can only do it so long before our brains get tired. Cognitive strain leads us to try least-effort solutions first, typically from our production rules, even when wrong. Fourth, humans self detect errors. About 75% of errors are detected immediately after they are made. Reason concludes that human errors are inevitable.

A second major observation, labeled *the Automation Irony*, is that automation does not cure human error. The reasoning is that once designers realize that humans make errors, they often try to design a system that reduces human intervention. Often this just shifts some errors from operator errors to design errors, which can be harder to detect and fix. More importantly, automation usually addresses the easy tasks for humans, leaving the complex, rare tasks that they didn t successfully automate to the operator. As humans are not good at thinking from first principles, humans are ill suited to such tasks, especially under stress. The irony is automation reduces the chance for operators to get hands-on control experience, which

prevents them from building mental production rules and models for troubleshooting. Thus automation often decreases system visibility, increases system complexity, and limits opportunities for interaction, all of which can make it harder for operators to use and make it more likely for them to make mistakes when they do use them. Ironically, attempts at automation can make a situation worse.

Our lessons from human error research are that human operators will always be involved with systems and that humans will make errors, even when they truly know what to do. The challenge is to design systems that are synergistic with human operators, ideally giving operators a chance to familiarize themselves with systems in a safe environment, and to correct errors when they detect they've made them.

2.3 Civil Engineering and Margin of Safety

Perhaps no engineering field has embraced safety as much as civil engineering. Petroski [1992] said this was not always the case. With the arrival of the railroad in the 19th century, engineers had to learn how to build bridges that could support vehicles that weighed tons and went fast.

They were not immediately successful: between the 1850s and 1890s about a quarter of the of iron truss railroad bridges failed! To correct that situation, engineers started studying failures, as they learned from bridges that fell than from those that survived. Second, they started to add redundancy so that some pieces could fail yet bridges would survive. However, the major breakthrough was the concept of a *margin of safety*; engineers would enhance their designs by a factor of 3 to 6 to accommodate the unknown. The safety margin compensated for flaws in building material, mistakes curing construction, putting too high a load on the bridge, or even errors in the design of the bridge. Since humans design, build, and use the bridge and since human errors are inevitable, the margin of safety was necessary. Also called the *margin of ignorance*, it allows safe structures without having to know everything about the design, implementation, and future use of a structure. Despite use of supercomputers and mechanical CAD to design bridges in 2002, civil engineers still multiply the calculated load by a small integer to be safe.

A cautionary tale on the last principle comes from RAID. Early RAID researchers were asked what would happen to RAID-5 if it used a bad batch of disks. Their research suggested that as long as there were standby spares on which to rebuild lost data, RAID-5 would handle bad batches, and so they assured others. A system administrator told us recently that every administrator he knew had lost data on RAID-5 one time in his career, although they had standby spare disks. How could that be? In retrospect, the quoted MTTF of disks assume nominal temperature and limited vibration. Surely, some RAID systems were exposed to

higher temperatures and more vibration than anticipated, and hence had failures much more closely correlated than predicted. A second flaw that occurred in many RAID systems is the operator pulling out a good disk instead of the failed disk, thereby inducing a second failure. Whether this was a slip or a mistake, data was lost. Had our field embraced the principle of the margin of safety, the RAID papers would have said that RAID-5 was sufficient for faults we could anticipate, but recommend RAID-6 (up to two disk failures) to accommodate the unanticipated faults. If so, there might have been significantly fewer data outages in RAID systems.

Our lesson from civil engineering is that the justification for the margin of safety is as applicable to servers as it is for structures, and so we need to understand what a margin of safety means for our field.

3. ROC Hypotheses: Repair Fast to Improve Dependability and to Lower Cost of Ownership

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

The Peres quote above is the guiding proverb of Recovery Oriented Computing (ROC). We consider errors by people, software, and hardware to be facts, not problems that we must solve, and fast recovery is how we cope with these inevitable errors. Since unavailability is approximately MTTR/MTTF, reducing time to recover by a factor of ten is just as valuable as stretching time to fail by a factor of ten. From a research perspective, we believe that MTTF has received much more attention than MTTR, and hence there may be more opportunities for improving MTTR. One ROC hypothesis is that is *recovery performance* is more fruitful for the research community and more important for society than traditional performance in the 21st century. Stated alternatively, Peres Law is will shortly be more important than Moore s Law.

A side benefit of reducing recovery time is its impact on cost of ownership. Lowering MTTR reduces money lost to downtime. Note that the cost of downtime is not linear. Five seconds of downtime probably costs nothing, five hours may waste a day of wages and a day of income of a company, and five weeks drive a company out of business. Thus, reducing MTTR may have nonlinear benefits on cost of downtime (see section 5.5 below). A second benefit is reduced cost of administration. Since a third to half of the system administrator s time may be spent with recovering from failures or preparing for the possibility of failure before an upgrade, ROC may also lower the people cost of ownership. The second ROC hypothesis is that research opportunities and customer s emphasis in the 21st century will be on total cost of ownership rather than on the conventional measure of price of purchase of hardware and software.

Progress moved so quickly on performance in part because we had a common yardstick--benchmarks--to measure success. To make such rapid progress on recovery, we need the similar incentives. With any benchmark, one of the first questions is whether it is realistic. Rather than guess why systems fail, we need to have the facts to act as a fault workload. Section 2 above shows data we have collected so far from Internet services and from telephone companies.

Although we are more interested in the research opportunities of MTTR, we note that our thrust complements research in improving MTTF, and we welcome it. Given the statistics in section 2, there is no danger of hardware, software, and operators becoming perfect and thereby making MTTR irrelevant.

4. Six ROC techniques

Although the tales from disasters and outages seem daunting, the ROC hypotheses and our virtual world let s us try things that are impossible in physical world, which may simplify our task. For example, civil engineers might need to design a wall to survive a large earthquake, but in a virtual world, it may be just as effective to let it fall and then replace a few milliseconds later. Our search for inspiration from other fields led to new techniques as well as some commonly found. Six techniques guide ROC:

1. *Redundancy to survive faults*. Our field has long used redundancy to achieve high availability in the presence of faults.
2. *Partitioning to contain failures and reduce cost of upgrade*. As we expect services to use clusters of independent computers connected by a network, it should be simple to use this technique to isolate a subset of the cluster upon a failure or during an upgrade. Partitioning can also help in the software architecture so that only a subset of the units needs to recover on a failure.
3. *Fault insertion to test recovery mechanisms and operators*. We do not expect advances in recovery until it is as easy to test recovery as it is today to test functionality and performance. Fault insertion is not only needed in the development lab, but in the field to see what happens when a fault occurs in a given system with its likely unique combination of versions of hardware, software, and firmware. Assuming the partitioning mechanism above, we should be capable of inserting faults in subsets of live systems without endangering the service. If so, then we can use this combination to train operators on a live system by giving them faults to repair. Fault insertion also simplifies running of availability benchmarks. .A related technique is supplying test inputs to modules of a service in order see if they provide the proper result. Such a mechanism reduces MTTR by reducing time to error detection.

4. *Aid in diagnosis of the cause of error*. Clearly, the system should help the operator determine what to fix, and this aid can reduce MTTR. In the fast changing environment of Internet services, the challenge is to provide aid that can keep pace with the environment.
5. *Non-overwriting storage systems and logging of inputs to enable operator undo*. We believe that undo for operators would be a very significant step in providing operators with a trial and error environment. To achieve this goal, we must preserve the old data. Fortunately, some commercial file systems today offer such features, and disk capacity is the fastest growing computer technology.
6. *Orthogonal mechanisms to enhance availability*. Fox and Brewer [2000] suggest that independent modules that provide only a single function can help a service. Examples are deadlock detectors from databases [Gray 1978]; hardware interlocks in Therac [Leveson 1993]; virtual machine technology for fault containment and fault tolerance [Bres95]; firewalls from security, and disk and memory scrubbers to repair faults before they accessed by the application.

Figure 3 puts several of these techniques together. We track a service during normal operation using some quality of service metric, perhaps hits per second. Since this rate may vary, you capture normal behavior using some statistical technique; in the figure, we used the 99% confidence interval. We then insert a fault into the hardware or software. The operator and system then must detect fault, determine the module to fix, and repair the fault. The primary figure of merit is repair or recovery time including the operator and the errors he or she might make [Brown02].

Despite fears to the contrary, you can accommodate the variability of people and still get valid results with just 10 to 20 human subjects [Neilsen93], or even as few as 5 [Neilsen02]. . Although few systems researchers work with human subjects, they are commonplace in the social sciences and in the Human Computer Interface community.

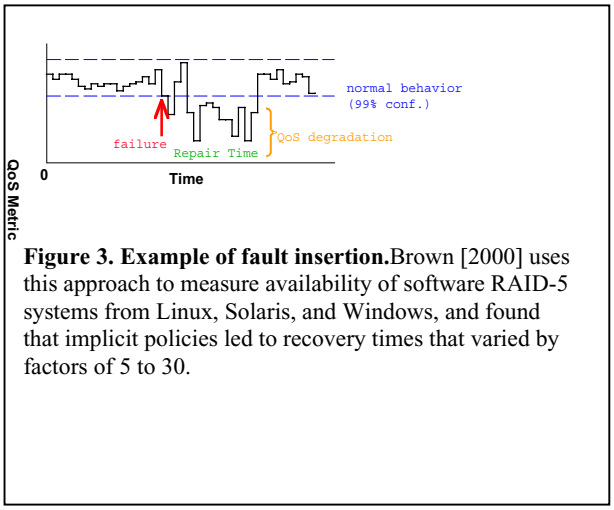


Figure 3. Example of fault insertion. Brown [2000] uses this approach to measure availability of software RAID-5 systems from Linux, Solaris, and Windows, and found that implicit policies led to recovery times that varied by factors of 5 to 30.

5. Case Studies of ROC Techniques

Given the definition, hypotheses, and techniques of ROC, how well do they work? This section gives five case studies using the six techniques above to indicate their benefits, especially highlighting the value of fault insertion in evaluating new ideas. Fitting this conference's roots, we go from hardware to software.

5.1 Hardware partitioning and fault insertion: ROC-1

The ROC-1 hardware prototype is a 64-node cluster composed of custom-built nodes, called *bricks*, each of which is an embedded PC board [Opp02]. Figure 4 shows a brick. For both space and power efficiency, the bricks are each packaged in a single half-height disk canister. Unlike other PCs, each brick has a diagnostic processor (DP) with a separate diagnostic network, whose purpose was to monitor the node, to isolate a node, and to insert errors. The idea was to turn off power to key chips selectively, including the network interfaces. The DP is an example of an orthogonal mechanism to partition the cluster and to insert faults, which are ROC techniques #6, #2, and #3.

ROC-1 did successfully allow the DP to isolate subsets of nodes; turning off the power of the network interface reliably disconnected nodes from the network. It was less successful at inserting errors by controlling power. It just took too much board area, and the chips contained too many functions for this power control to be effective.

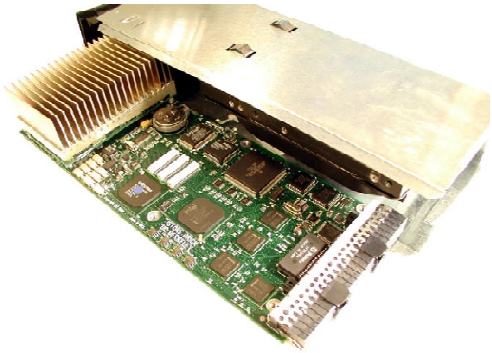


Figure 4. A ROC-1 brick. Each brick contains a 266 MHz mobile Pentium II processor, an 18 GB SCSI disk, 256 MB of ECC DRAM, four redundant 100 Mb/s network interfaces connected to a system-wide Ethernet, and an 18 MHz Motorola MC68376-based diagnostic processor. Eight bricks fit into a tray, and eight trays form the cluster, plus redundant network switches. See [Opp02] for more details.

The lesson from ROC-1 is that we can offer hardware partitioning with standard components, but the high amount of integration suggests that if hardware fault insertion is necessary, we must change the chips internally to support such techniques.

5.2 Software Fault Insertion: FIG

The awkwardness of hardware fault isolation in ROC-1 inspired an experiment in software fault insertion. *FIG* (Fault Injections in glibc) is a lightweight, extensible tool for injecting and logging errors at the application/system boundary. FIG runs on UNIX-style operating systems and works by interposing a library between the application and the glibc system libraries that intercepts calls from the application to

the system. When a call is intercepted, our library then chooses, based on testing directives from a control file, whether to allow the call to complete normally or to return an error that simulates a failure of the operating environment. Although our implementation of FIG targets functions implemented in glibc, it could easily be adapted to instrument any shared library.

To test the effectiveness of FIG, we started with three mature applications: the text editor Emacs (using X and without using X), the open source flat file database library Berkeley DB (with and without transactions) and the open source http server Apache. We assumed if fault insertion helped evaluate the dependability of mature software, then it would surely be helpful for newly developed software.

We inserted errors in a dozen system calls, and Table 2 shows results for `read()`, `write()`, `select()`, and `malloc()`. We tried Emacs with and without the X widowing system, and it fared much better without it: EIO and ENOMEM caused crashes with X, and more acceptable behavior resulted without it. As expected, Berkeley DB under non-transactional mode did not handle errors gracefully, as write errors could corrupt the database. Under transactional mode, it detected and recovered from all but memory errors properly. Apache was "best of show", as it was the most robust of the applications tested.

One lesson from FIG is that even mature, reliable programs have mis-documented interfaces and poor error recovery mechanisms. We conclude that application development can benefit from a comprehensive testing strategy that includes mechanisms to introduce errors from the system environment, showing the value of that ROC principle (#3). FIG provides a straightforward method for introducing errors. Not only can FIG be used in development for debugging recovery code, but in conjunction with hardware partitioning, it can be used in production to help expose latent errors in the system.

System Call	`read()`		`write()`		`select()`	`malloc()`
˚Error	EINTR	EIO	ENOSPC	EIO	ENOMEM	ENOMEM
Emacs - no X	O.K.	exits	warns	warns	O.K.	crash
Emacs - X	O.K.	crash	O.K.	crash	crash / exit	crash
Berkeley DB - no xact	halts	halts	DB corrupt	DB corrupt	n/a	halts
Berkeley DB - xact	halts	halts	O.K.	O.K.	n/a	halts
Apache	O.K.	request dropped	request dropped	request dropped	O.K.	n/a

Table 2. Reaction of applications to faults inserted in four system calls. EINTER = Exception /Interrupt, EIO = I/O error, ENOSPC = no disk space, ENOMEM = no main memory space. See [BST02] for more details.

Even with this limited number of examples, FIG also allows us to see both successful and unsuccessful application programming. Three examples of successful practices are *resource preallocation*: requesting all necessary resources at startup so failures do not occur in the middle of processing; *graceful degradation*:

offering partial service in the face of failures to ameliorate downtime; and *selective retry*: waiting and retrying a failed system call a bounded number of times, in the hope that resources will become available. FIG helps evaluate a suspect module, while the next case study aids the operator to find the culprit.

5.3 Automatic Diagnosis: Pinpoint

What is the challenge? A typical Internet service has many components divided among multiple tiers as well as numerous (replicated) subcomponents within each tier. As clients connect to these services, their requests are dynamically routed through the system. The large size of these systems results in more places for faults to occur. The increase in dynamic behavior means there are more paths a request may take through the system, and thus results in more potential failures due to "interaction'' faults among components. Also, rapid change in hardware and software of services make automated diagnosis harder.

Fault diagnosis techniques traditionally use dependency models, which are statically generated dependencies of components to determine which components are responsible for the symptoms of a given problem. Dependency models are difficult to generate and they are difficult to keep consistent with an evolving system. Also, they reflect the dependencies of *logical* components but do not differentiate *replicated* components. For example, two identical requests may use different instances of the same replicated components. Therefore, dependency models can identify which component is at fault, but not which *instance* of the component. Hence, they are a poor match to today's Internet services.

Instead, we use dynamic analysis methodology that automates problem determination in these environments without dependency analysis. First, we dynamically trace real client requests through a system. For each request, we record its believed success or failure, and the set of components used to service it. Next, we perform standard data clustering and statistical techniques to correlate the failures of requests to components most likely to have caused them.

Tracing real requests through the system let's use determine problems in dynamic systems where dependency modeling is not possible. This tracing also allows us to distinguish between multiple instances of what would be a single logical component in a dependency model. By performing data clustering to analyze the successes and failures of requests, we find the combinations of components that are most highly correlated with the failures of requests, under the belief that these components are causing the failures. By analyzing the components used in the failed requests, but are not used in successful requests, we provide high accuracy with relatively low number of false positives. This analysis detects individual faulty

components, as well as faults occurring due to interactions among multiple components. Additionally, this analysis tends to be robust in the face of multiple independent faults and spurious failures. We call this system *Pinpoint*; see [Chen02] for more details.

We implemented a prototype of Pinpoint on top of Java2 Enterprise Edition (J2EE) middleware. Our prototype does not require any modifications to J2EE applications. The only application knowledge our prototype requires are application-specific checks to enable external fault detection and, even in this case, no application component needs to be modified. Because of this, Pinpoint can be a problem determination aid for almost any J2EE application. To evaluate performance impact, we compared an application hosted on an unmodified J2EE server and on our version with logging turned on, and found that the overhead of Pinpoint was just 8.4%.

The measure of success in problem determination is not only the fraction of faults correctly identified, but also the rate of false positives. Depending on the circumstances, you may be more concerned with high hit rates or with low false positive rates. From statistics comes a technique, coincidentally abbreviated *ROC* (receiver operating characteristics), to plot the trade-off between hit rate and false-positive rate. A sensitivity "knob" is changed that typically increases both hit rate and false positives. Good results are up

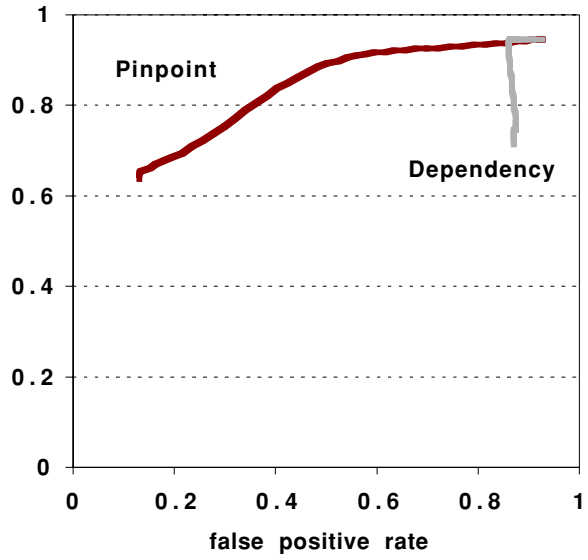


Figure 5. A ROC analysis of hit rate vs. false positive rate. To validate our approach, we ran the J2EE Pet Store demonstration application and systematically injected faults into the system over a series of runs. We ran 55 tests that included single-component faults and faults triggered by interactions components. This graph show single component faults. It is important to note that our fault injection system is kept separate from our fault detection system.The clustering method used for the Pinpoint analysis was the unweighted pair-group method using arithmetic averages with the Jaccard similarity coefficient. [Jain 1988] The sensitivity "knob" set the distance (between elements) at which we stopped merging elements/clusters together. The dependency analysis looks at the components that are used in failed requests. The sensitibity knob is the list of components that occurred in at least X% of the failed requests, where X is the sensitivity setting. When 0%, it is the set of components that are used in all failed requests; when 100%, it is the set used in any of the failed requests. The two lines are Bezier curve interpolation of the data points. For more detail, see [Chen02].

(high hit rate) and to the *left* (low false positive rate). For example, if there were 10 real faulty modules, a hit rate of 0.9 with a false positive rate of 0.2 would mean the system correctly identified 9 of the 10 real ones and supplied 2 false ones.

Figure 5 compares Pinpoint to a more traditional dependency analysis using ROC. Even at a low sensitivity setting, Pinpoint starts at a false positive rate of just 0.1 combined with a hit rate above 0.6. Raising the sensitivity pushes the false positive rate of 0.5 with a hit rate to 0.9. In contrast, the simple dependency approach starts with a very high false positive rate: 0.9. The sensitivity setting can improve the hit rate from 0.7 to 0.9, but the false positive rate starts at 0.9 and stays there.

We were pleased with both the accuracy and overhead of Pinpoint, especially given its ability to match the dynamic and replicated nature of Internet services. One limitation of Pinpoint is that it can not distinguish between sets of tightly coupled components that always used together. We are looking at inserting test inputs to isolate such modules. Another limitation of Pinpoint, as well as existing error determination approaches, is that it does not work with faults that corrupt state and affect subsequent requests. The non-independence of requests makes it difficult to detect the real faults because the subsequent requests may fail while using a different set of components.

This case study showed an example of aiding diagnosis (ROC technique #4) using fault insertion (#3) to help evaluate it. Pinpoint can reduce time of recovery by reducing the time for the operator to find the fault, but the next case study reduces time for a system to recover after the module is identified.

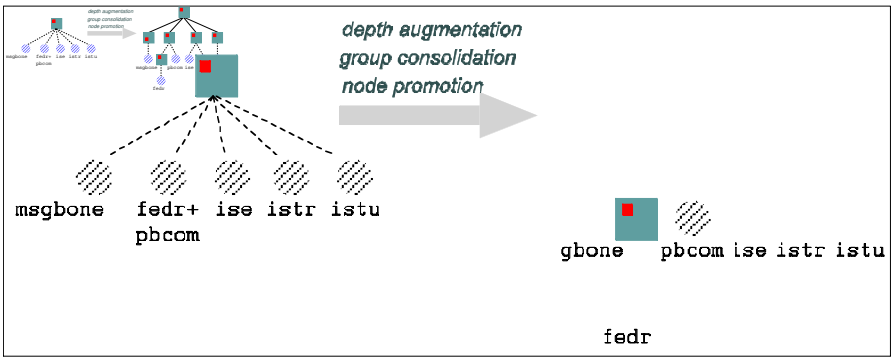
5.4 Recursive Restartability and Fine Grain Partitioning: Mercury

In Mercury, a satellite ground station, we employed fine grain partial restarts to reduce the mean-time-to-recover of the control software by almost a factor of six. Besides being a significant quantitative improvement, it also constituted a qualitative improvement that lead to a near-100% availability of the ground station during the critical period when the satellite passes overhead.

Recursive restartability [Candea01a] is an approach to system recovery predicated on the observation that, in critical infrastructures, most bugs cause software to crash, deadlock, spin, livelock, leak memory, corrupt the heap, and so on, leaving a reboot or restart as the only high-confidence way of bringing the system back into operation [Brewer01, Gray78]. Reboots are an effective and efficient workaround because they are easy to understand and employ, provide a high confidence way to reclaim stale or leaked

resources, and unequivocally return software to its start state-- generally the best tested and understood state of the system. Unfortunately, most systems do not tolerate unannounced restarts, resulting in unnecessarily long downtimes associated with potential data loss. A recursively restartable (RR) system, however, gracefully tolerates successive restarts at multiple levels. Fine grain partitioning enables bounded, partial restarts that recover a failed system faster than a full reboot. RR also enables strong fault containment, diagnosis, and benchmarking [Candea01b]

We applied the RR ideas to Mercury, a ground station for low earth orbit satellites, which currently controls communication with two orbiting satellites. Mercury is built from COTS antennas and radios, driven by x86-based PCs running Linux with most of the software written in Java. Mercury breaks with the satellite community tradition by emphasizing low production cost over mission criticality.



Failed component	msgbone	fedr	pbcom	ise/istr	istu
MTTF	1 month	10 min	1 month	5 hours	5 hours
MTTR before	**28.9** sec	**28.9** sec	**28.9** sec	**28.9** sec	**28.9** sec
MTTR after	**4.7** sec	**5.0** sec	**21.9** sec	**6.1** sec	**5.8** sec

Figure 6. Applying Recursive Restartability to a satellite ground station. The Mercury instances before and after applying RR are described by the two restart trees. Mercury consists of a number of components: a communication backbone for XML messages (msgbone), an XML command translator (fedr), a serial port to TCP socket mapper (pbcom), a position and azimuth estimator (ise), a satellite tracker (istr), and a radio tunner (istu). The first row of the table indicates the observed MTTFs of these components based on 2 years of operation. The last two rows show the effect of the transformations on the system s time-to-recover, as a function of the component that fails.

Figure 6 shows that a recursively restartable system, of which Mercury is an example, can be described by a restart tree--a hierarchy of restartable components in which nodes are highly failure-isolated. A restart at any node results in the restart of the entire subtree rooted at that node. A restart tree does not capture *functional* dependencies between components, but rather the *restart* dependencies. Subtrees represent *restart groups*, which group together components with common restart requirements. Based on information of which component(s) failed (e.g., using Pinpoint methods described above), an *oracle*

dictates which node in the tree must be restarted. The goal of a perfect oracle is to always choose a minimal set of subsystems that need restarting, in order to minimize the time-to-recover. If a particular restart does not cure the failure, the oracle may choose to hike the tree and restart at a higher level. Restarting groups that are higher up in the restart tree leads to a longer but higher confidence recovery.

To minimize time-to-recover in Mercury, [Candea02]. describes a number of transformations we applied. Figure 6 summarizes the result of these transformations. The most compelling result is a six-fold reduction in recovery time for failures occurring in `fedr`. Since this component failed very often due to JVM crashes, the quantitative decrease in time-to-recover led to a qualitative increase in availability. Using the frequency of failures in Figure 6, the weighted average recovery time falls from 28.9 seconds to 5.6 seconds.

It is already an accepted fact that not all downtime is the same. Downtime under a heavy or critical workload is more expensive than downtime under a light or non-critical workload, and unplanned downtime being generally more expensive than planned downtime [Brewer01]. Relevant for the service space, Mercury constitutes an interesting example in the mission-critical space: downtime during satellite passes is very expensive because we may lose science data and telemetry. Worse, if the failure involves the tracking subsystem and the recovery time is too long, we may lose the whole pass; the antenna and spacecraft will get sufficiently misaligned that the antenna cannot catch up in time when the tracking subsystem comes back online. A large MTTF does not guarantee a failure-free pass, but a short MTTR can help guarantee that we will not lose the whole pass because of a failure. As in Section 3, here is another example where the cost of downtime is not linear in its length. Such quantitative change in recovery time leading to a qualitative change in system behavior has been observed in a number of large scale infrastructures. For example, on September 11, 2001 the long recovery time of the CNN.com web site prevented the whole web site from recovering under increased load [LeFebvre01].

Applying recursive restartability to a mission-critical system allowed us to reduce its time to recover from various types of failures and achieve nearly 100% availability with respect to satellite passes. The Mercury ground station is an example of an end-to-end ROC system incorporating fine grain partitioning hooks for fault insertion, logging-based aid for diagnosis of error, and strong orthogonal mechanisms, or ROC techniques #2, #3, #4, and #6. Mercury showed how to reduce time to recover from software or hardware faults, while the next technique helps recover from operator faults as well.

5.5 Recovery via Undo: Design of an Undoable Email System

Our last case study is an Undo layer designed to provide a more forgiving environment for human system operators; it illustrates the ROC techniques of using non-overwriting storage systems with input logging (#5) and orthogonal mechanisms (#6). We have two goals here. The first is to provide a mechanism that allows operators to recover from their inevitable mistakes; recall from Section 2 that operator errors are a leading cause for service failures today. The second goal is to give operators a tool that allows them to retroactively repair latent errors that went undetected until too late. Here we leverage the fact that computers operate in a virtual world in which the effects of latent errors can often be reversed, unlike in physical systems like TMI.

We have chosen email as the target application for our first implementation of a ROC Undo layer. Email has become an essential service for today s enterprises, often acting as the communications "nervous system" for businesses and individuals alike, and it is one of the most common services offered by network service vendors. Email systems also offer many opportunities for operator error and retroactive repair. Examples of operator error that can be addressed by our undo layer include: misconfigured filters (spam, antivirus, procmail, and so on) that inadvertently delete user mail, accidental deletion of user mailboxes, mail corruption when redistributing user mailboxes to balance load, and installation of buggy or broken software upgrades that perform poorly or corrupt mail. Retroactive repair is useful in an email system when viruses or spammers attack: with an undo system, the operator can "undo" the system back to the point before the virus or spam attack began, retroactively install a filter to block the attack, then "redo" the system back to the present time. Furthermore, the undo abstraction could propagate to the user, allowing the user to recover from inadvertent errors such as accidentally deleting messages or folders without involving the sysadmin.

To support recovery from operator error and retroactive repair, our ROC undo model supports a 3-step undo process that we have coined "*the three R s*": *Rewind, Repair*, and *Replay*. In the rewind step, all system state (including mailbox contents as well as OS and application hard state) is reverted to its contents at an earlier time (before the error occurred). In the repair step, the operator can make any changes to the system he or she wants. Changes could include fixing a latent error, installing a preventative filter or patch, or simply retrying an operation that was unsuccessful the first time around (like a software upgrade of the email server). Finally, in the replay step, the undo system re-executes all user interactions with the system,

reprocessing them with the changes made during the repair step. There are of course many challenges in implementing the 3R model, notably in replaying interactions that may have already become externally visible before the undo cycle was initiated. Further details describing our approaches are in [Brown02].

We are building a prototype implementation of our email-targeted undo layer. Our prototype is designed as a proxy layer that wraps an existing, unmodified IMAP- and SMTP-based mail server. We chose to build the undo layer as a proxy to allow recovery from operator error during major events, such as software upgrades of the mail server. Figure 7 illustrates the operation of the proxy.

Besides the proxy, the other major component of our prototype is a non-overwriting storage layer that sits underneath the mail server. This layer enables the rewind phase of undo by providing access to historical versions of the system s hard state. The time-travel layer can be implemented using file system snapshots (such as those provided by the Network Appliance WAFL file system [Hitz95]), although we are investigating using a more flexible versioning system such as the Elephant file system [Santry99].

An analysis of traffic logs from our departmental mail server for 1200 users indicates that the storage overhead of keeping the undo log is about 250 MB/day using our unoptimized prototype. A single 120-GB EIDE disk, which costs just $180 in 2002, stores more than a year of log data; not much to get the 3Rs.

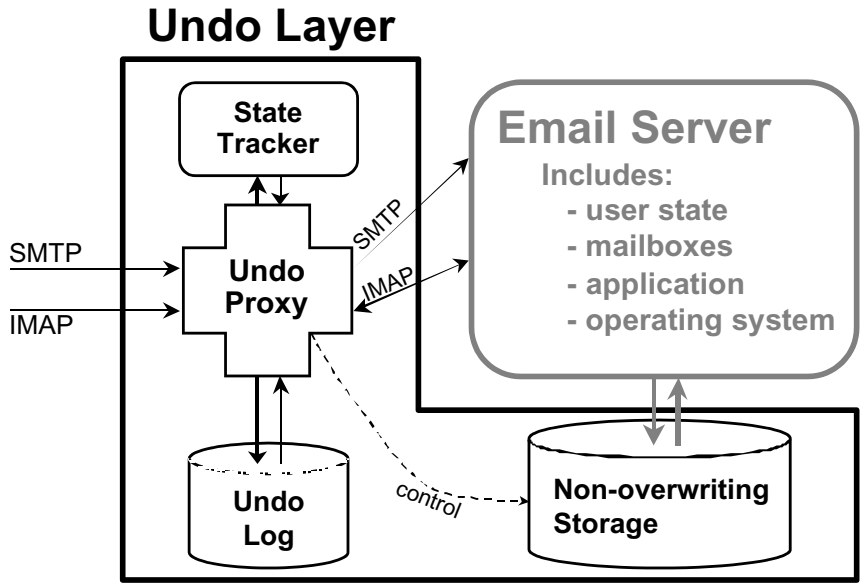


Figure 7. Architecture of Undo layer for Email. During normal operation, the proxy snoops IMAP and SMTP traffic destined for the mail server and logs mail delivery and user interactions. The proxy also monitors accesses to messages and folders and uses that information to track state that has been made externally visible; this allows the system to issue appropriate compensating actions when the repair/replay cycle invalidates previously-externalized state. Upon an undo request, the non-overwriting storage layer rolls back to the undo point, the operator performsany needed repairs, and then the proxy replays the logged user interactions that were lost during the rollback. The proxy continues to log incoming mail during the undo cycle, although users are only given read-only access until replay has completed.

6. Related Work

We are not the first to have thought about recovery; many research and commercial systems have addressed parts of the ROC agenda that we have defined in this paper. The storage community probably comes closer to embracing the ROC ideal than any other. Most of the commercial storage vendors offer products explicitly designed to improve recovery performance; a good example is EMC s TimeFinder system [EMC02]. TimeFinder automatically partitions and replicates storage; EMC suggests that the alternate partitions be used for ROC-like isolated on-line testing and for fast post-crash recovery of large service systems like Microsoft Exchange. In the research community, recovery-enhancing techniques have emerged serendipitously from originally performance-focused work, as in the development of journaling, logging, and soft-update-based file systems [Rosenblum92] [Seltzer93] [Hitz95].

Recovery-oriented work in the OS community is rarer, but still present. Much of it focuses on the ability to restart quickly after failures. An early example is Sprite s "Recovery Box", in which the OS uses a protected area of non-volatile memory to store crucial state needed for fast recovery [Baker92]. This basic idea of segregating and protecting crucial hard-state to simplify recovery reappears frequently, for example in the derivatives of the Rio system [Chen96][Lowell97][Lowell98], and in recent work on soft-state/hard-state segregation in Internet service architectures [Fox97] [Gribble00]. Most of these systems still use increased performance as a motivation for their techniques; recovery benefits are icing on the cake.

The database community has long paid attention to recovery, using techniques like write-ahead logging [Mohan92] and replication [Gray96] to protect data integrity in the face of failures. Recovery performance has also been a topic of research starting with the POSTGRES system, where they redesigned the database log format to allow near-instantaneous recovery [Stonebraker87]. A recent example of recovery is Oracle 9i, which includes a novel Fast Start mechanism for quick post-crash recovery [Lahiri01] and a limited version of an undo system that allows users to view snapshots of their data from earlier times [Oracle01]. In general, however, transaction performance has far overshadowed recovery performance, probably due to the influence of the performance-oriented TPC benchmarks.

Finally, the traditional fault-tolerance community has occasionally devoted attention to recovery. The best example of this is the work on software rejuvenation, which periodically restarts system modules to flush out latent errors [Huang95] [Garg97] [Bobbio98]. Another illustration is in work on built-in self-test in embedded systems, in which components are designed to proactively scan for possible latent errors and

to immediately fail and restart when any are found [Steininger99]. In general, however, the fault-tolerance community does not believe in Peres s law, and therefore focuses on MTTF under the assumption that failures are generally problems that can be known ahead of time and should avoided.

Our ROC approach differentiates itself from this previous work in two fundamental ways. First, ROC treats recovery holistically: a ROC system should be able to recover from any failure at any level of the system, and recovery should encompass all layers. Contrast this with, say, database or storage recovery, where recovery is concerned only with the data in the database/storage and not the entire behavior of the service built on top. Second, ROC covers a much broader failure space than these existing approaches. In particular, ROC addresses human-induced failures, which are almost entirely ignored in existing systems work. ROC also makes no assumptions about what failures might occur. Traditional fault-tolerance work typically limits its coverage to a set of failures predicted by a model; in ROC, we assume that anything can happen and we provide mechanisms to deal with unanticipated failure.

Each ROC technique in this paper draws on a background of prior work. Although space limitations prevent us from going into full detail, we provide some pointers here for the interested reader.

- *Measurements of availability*: The seminal work in availability data collection is Gray s study of Tandem computer system failures [Gray86]. More recent work include Murphy s availability studies of VAX systems and of Windows 2000 [Murphy95] [Murphy00].
- *Fault insertion*: Fault insertion has a long history in the fault-tolerance community and covers a range of techniques from heavy-ion irradiation [Carreira99] to software simulation of hardware bugs [Segal88] [Arlat92] and programming errors [Chandra98] [Kao93]. Most existing work uses fault insertion as an offline technique used during system development, although there are a few systems that have been built with the capability for online fault injection (notably the IBM 3090 and ES/9000 mainframes [Merenda92]).
- *Problem diagnosis*: There are several standard approaches to problem diagnosis. One is to use models and dependency graphs to perform diagnosis [Choi99] [Gruschke98] [Katker97] [Yemini96]. When models are not available, they can either be discovered [Kar00] [Brown01] [Miller95], or alternate techniques can be used, such as Banga s system-specific combination of monitoring, protocol augmentation, and cross-layer correlation [Banga00]. Our Pinpoint example

demonstrates another approach by tracking requests through the system as is done in distributed resource utilization monitors [Reumann00].

- *Undo*: Our model of an "undo for operators," based on the three R s of rewind, repair, and replay, appears to be unique in its ability to support arbitrary changes during the repair stage. However, there are many similar systems that offer a subset of the three R s, including systems like the EMC TimeFinder [EMC02], a slew of checkpointing and snapshot systems [Elnozahy96] [Borg89], and traditional database log recovery [Mohan92]. Additionally, our undo implementation relies heavily on existing work in non-overwriting storage systems.
- *Benchmarking people*: While including human behavior in our ROC benchmarks and systems may be novel in the systems community, it is something that has been done for years in the HCI community. (Landauer [1997] gives an excellent survey of HCI methods and techniques.) Our intentions are slightly different, however: HCI is primarily concerned with the interface, whereas we want to address the system s behavior when the human is included, regardless of interface. In that sense our work is most similar to work in the security community on the effectiveness of security-related UIs, such as Whitten and Tygar s study of PGP [Whitten99].

7. Discussion and Future Directions

If it s important, how can you say it s impossible if you don t try?
Jean Monnet, a founder of the European Union

We have presented statistics on why services fail, finding that operator error is a major cause of outages and hence portion of cost of ownership. Following Peres' Law, ROC assumes that hardware, software, and operator faults are inevitable, and fast recovery both improves availability and cost of ownership. We argued that failure data collection, availability benchmarks involving people, and margin of safety will be necessary for success. We presented six ROC techniques, some of which were inspired by other fields. They are redundancy, partitioning, fault insertion, diagnosis aid, nonoverwriting storage, and orthogonal mechanisms. We list five case studies that use those techniques: a cluster with hardware partitioning and fault insertion, a software library that inserts faults, middleware that diagnosis faults, partitioned software that recovers more quickly, and design of an email service with operator undo.

Habit is a powerful attraction for researchers to continue to embrace of performance, as we certainly have been successful and know how to do more. Alas, it does not pass a common sense test. If we wouldn't

spend our own money to buy a faster computer, since our old one is still fast enough, why are almost all of us researching how to make systems faster?

ROC looks hard now because we're not sure we know how or if we can do it. But as the quote above suggests, if you agree that dependability and maintainability is important, how can say its impossible if you don't try?

At this early stage, the horizon is filled with important challenges and ooportunities:

- We need a theory of constructing dependable, maintainable sites for networked services.
- We need a theory of good design for operators as well as good design for end users. Using an airline analogy, its as though we have good guidelines for passengers but not for pilots.
- We need a more nuanced definition of failure than up or down. Perhaps we can find the information technology equivalent of blocked calls (Figure 2) collected by telephone companies?
- We need to economically quantify cost of downtime and ownership, for if there are not easy ways to measure them for an organization, who will buy new systems that claim to improve them?
- We need to continue the quest for real failure data and to develop useful availability and maintainability benchmarks. This quest let's us measure progress, lower barriers to publication by researchers, and humiliate producers of undependable computer products.
- The design of our initial email prototype is intentionally simplistic and is primarily as a testbed for examining policies governing externalized actions. In future versions, we intend to extend the prototype by providing undo on a per-user basis (to allow users to fix their own mistakes), by providing read-write access during the undo cycle by synthesizing consistent states from the information in the log, and by adding hooks to the mail server to reduce the proxy s complexity and improve the system s recovery time further. Our intent is that this prototype will eventually leverage all six ROC techniques.
- Given the difficulty of hardware fault insertion, virtual machines may be worth investigating as a fault insertion device. Trusting a VM is more akin to trusting a processor than it is to trusting a full OS [CN01]. They may also help with partitioning and even recovery time, as it may be plausible to have hot standby spares of virtual machines to take over upon a fault. Finally, a poorly behaving system can occupy so many resources that it can be hard for the operator to login and kill the offending processes, but VM provides a way out.

Acknowledgements

NSF ITR, NSF Career Award, + company support: Acies Network, Microsoft.

References

[Arlat92] J. Arlat, A. Costes, et al. Fault Injection and Dependability Evaluation of Fault-Tolerant Systems. *LAAS-CNRS Research Report 91260*, January 1992.

[Baker92] M. Baker and M. Sullivan. The recovery box: Using fast recovery to provide high availability in the unix environment. *Proc. of the Summer USENIX Conf.*, June 1992.

[Banga00] G. Banga. Auto-diagnosis of Field Problems in an Appliance Operating System. *Proc. of the 2000 USENIX Annual Technical Conf.*, San Diego, CA, June 2000.

[Bobbio98] A. Bobbio and M. Sereno. Fine grained software rejuvenation models. *Proc. IEEE Int'l.Computer Performance and Dependability Symp*., Sep. 1998, pp. 4—12.

[Borg89] A. Borg, W. Blau, W. Graetsch et al. Fault Tolerance Under UNIX. *ACM Transactions on Computer Systems*, 7(1):1—24, February 1989.

[Bres95] T.C. Bressoud and F.B. Schneider. Hypervisor-based fault tolerance. In Proc. Fifteenth ACM Symp. on Oper. Sys. Principles (SOSP-15), Copper Mtn., CO, Dec. 1995.

[Brewer 01] Brewer, E.A. Lessons from giant-scale services. *IEEE Internet Computing*, vol.5, (no.4), IEEE, July-Aug. 2001. p.46-55.

[BST02] Broadwell, P, Naveen Sastry Jonathan Traupman. Fault Injection in glibc (FIG), www.cs.berkeley.edu/~nks/fig/paper/fig.html.

[Brown00] A. Brown and D.A. Patterson. Towards Availability Benchmarks: A Case Study of Software RAID Systems. *Proc 2000 USENIX Annual Technical Conf.*, San Diego, CA, June 2000.

[Brown01] A. Brown, G. Kar, and A. Keller. An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment. *Proc 7th IFIP/IEEE Int'l.Symp.on Integrated Network Management* (IM VII), Seattle, WA, May 2001.

[Brown02] A. Brown and D.A. Patterson, Design of an Email system with Undo. *In preparation*.

[Candea01a] Candea, G.; Fox, A. Recursive restartability: turning the reboot sledgehammer into a scalpel. Proc. 8th Workshop on Hot Topics in Operating Systems, 2001. p.125-30.

[Candea01b] Candea, G.; Fox, A, Designing for high availability and measurability, 1st Workshop on Evaluating and Architecting System dependability G teborg, Sweden, July 1, 2001.

[Candea02] Candea, G.; Fox, A. et al. Recursive Restart for Mercury, in preparation..

[Carreira99] J. Carreira, D. Costa, and J. Silva. Fault injection spot-checks computer system dependability. *IEEE Spectrum* 36(8):50-55, August 1999.

[Chandra98] S. Chandra and P. M. Chen. How Fail-Stop are Faulty Programs? *Proc. of the 1998 Symp.on Fault-Tolerant Computing (FTCS)*, June 1998.

[Chen96] P. M. Chen, W. T. Ng, S. Chandra, et al. The Rio File Cache: Surviving Operating System Crashes. In *Proc. of the 7th Int'l.Conf. on ASPLOS*, pages 74--83, 1996.

[Chen02] Chen, M. et al, " Pinpoint: Problem Determination in Large, Dynamic Internet Services," in prepartion., 2002.

[CN01] Peter M. Chen and Brian Noble. When virtual is better than real. In Proc. Eighth Symp.on Hot Topics in Operating Systems (HotOS-VIII), Elmau, Germany, May 2001.

[Choi99] J. Choi, M. Choi, and S. Lee. An Alarm Correlation and Fault Identification Scheme Based on OSI Managed Object Classes. *1999 IEEE Int'l.Conf. on Communications*, Vancouver, BC, Canada, 1999, 1547—51.

[Elnozahy96] 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.

[EMC02] EMC TimeFinder, http://www.emc.com/products/software/timefinder.jsp.

[Fox97] A. Fox, S. Gribble, Y. Chawathe et al. Cluster-based Scalable Network Services. *Proc. of the 16th Symp.on Operating System Principles (SOSP-16)*, St. Malo, France, October 1997.

[Fox99] Fox, A.; Brewer, E.A. Harvest, yield, and scalable tolerant systems. Proc.s 7th Workshop on Hot Topics in Operating Systems, , Rio Rico, AZ, USA, 29-30 March 1999. p.174-8.

[Garg97] S. Garg, A. Puliafito, M. Telek, K.S. Trivedi. On the analysis of software rejuvenation policies. *Proc. of the 12th Annual Conf. on Computer Assurance*, June 1997, pp. 88—96.

[Gates02] Gates, W. Miscrosoft email, Jamuary 15, 2002. Reported in N.Y.times.

[Gillen2002] Gillen, Al, Dan Kusnetzky, and Scott McLaron "The Role of Linux in Reducing the Cost of Enterprise Computing , IDC white paper, Jan. 2002, available at www.redhat.com.

[Gray78] J. Gray,: *Notes on Data Base Operating Systems. Advanced Course: Operating* Systems 1978: 393-481.

[Gray86] J. Gray. Why Do Computers Stop and What Can Be Done About It? *Symp.on Reliability in Distributed Software and Database Systems*, 3—12, 1986.

[Gray96] J. Gray, P. Helland, P. O Neil, and D. Shasa. The Dangers of Replication and a Solution. *Proc. of the 1996 ACM SIGMOD Int'l.Conf. on Data Management*, 1996, 173--182.
[Gray02] Gray, J. *What Next? A dozen remaining IT problems*, Turing Award Lecture , 1999

[Gribble00] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. *Proc. of the Fourth Symp.on Operating Systems Design and Implementation (OSDI 2000)*, 2000.

[Gruschke98] B. Gruschke. Integrated Event Management: Event Correlation Using Dependency Graphs. *Proc. of 9th IFIP/IEEE Int'l.Workshop on Distributed Systems Operation & Management (DSOM98)*, 1998.

[Hennessy99] J. Hennessy, The Future of Systems Research, *Computer*, August 1999 32:8, 27-33.

[HP02], Hennessy, J., D. Patterons, Computer Architecture: A Quatitative Approach, 3rd edition, Morgan Kauffman, San Francisco, 2002.

[Hitz95] D. Hitz, J. Lau, M. Malcolm. File System Design for an NFS Server Appliance. *Network Appliance Technical Report TR3002*, March 1995.

[Huang95] Y. Huang, C. Kintala, N. Kolettis, N.D. Fulton. Software rejuvenation: analysis, module and applications. *Proc. 25th Int'l.Symp.on Fault-Tolerant Computing*, June 1995, pp. 381—390.

[IBM 01]] IBM, Automonic Computing, http://www.research.ibm.com/autonomic/., 2001

[Jain88] Jain, A. and Dubes, R., () *_Algorithms for Clustering Data*. Prentice Hall, New Jersey. 1988.

[Kao93] W. Kao, R. Iyer, and D. Tang. FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behavior Under Faults. *IEEE Transactions on Software Engineering*, 19(11):1105—1118, 1993.

[Kar00] G. Kar, A. Keller, and S. Calo. Managing Application Services over Service Provider Networks: Architecture and Dependency Analysis. *Proc. of the Seventh IEEE/IFIP Network Operations and Management Symp.(NOMS 2000)*, Honolulu, HI, 2000.

[Katker97] S. K tker and M. Paterok. Fault Isolation and Event Correlation for Integrated Fault Management. *Fifth IFIP/IEEE Int'l.Symp.on Integrated Network Management (IM V)*, San Diego, CA, 1997, 583—596.

[Kembel2000] Kembel, R. *Fibre Channel: A Comprehensive Introduction*, p.8, 2000.

[Lahiri01] T. Lahiri, A. Ganesh, R. Weiss, and A. Joshi. Fast-Start: Quick Fault Recovery in Oracle. *Proc. of the 2001 ACM SIGMOD Conf.*, 2001.
Lampson, B. [1999] *Computer Systems Research-Past and Future ,* Keynote address, 17th SOSP, Dec. 1999.

[Landauer97] Landauer, T. K. Research Methods in Human-Computer Interaction. In *Handbook of Human-Computer Interaction, 2e*, M Helander et al. (ed), Elsevier, 1997, 203—227.

[LeFebvre01]. ?

[Lowell97] D. E. Lowell and P. M. Chen. Free Transactions with Rio Vista. In *Proc. of the 16th ACM Symp.on Operating Systems Principles*, October 1997, pp. 92-- 101.

[Lowell98] D. E. Lowell and P. M. Chen. Discount Checking: Transparent, Low-Overhead Recovery for General Applications. *University of Michigan CSE-TR-410-99*, November 1998.

[Lowell00] D. E. Lowell, S. Chandra, and P. Chen. Exploring Failure Transparency and the Limits of Generic Recovery. *Proc. of the 4th Symp.on Operating System Design and Implementation*. San Diego, CA, October 2000.

[Merenda92] A. C. Merenda and E. Merenda. Recovery/Serviceability System Test Improvements for the IBM ES/9000 520 Based Models. *Proc. of the 1992 Int'l.Symp.on Fault-Tolerant Computing*, 463—467, 1992.

[Miller95] B. Miller, M. Callaghan et al. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer* 28(11):37—46, November 1995.

[Mohan92] 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.

[Murphy95] 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.

[Murphy00] B. Murphy and B. Levidow. Windows 2000 Dependability. *Microsoft Research Technical Report, MSR-TR-2000-56*, June 2000.

[Nielsen 93] Nielsen, J., and Landauer, T. K. A mathematical model of the finding of usability problems. Proc. of the ACM INTERCHI 93 Conf., Amsterdam, The Netherlands, April 1993, 206—213.

[Nielsen02] Nielsen, J. "Why You Only Need To Test With 5 Users," www.useit.com/alertbox/20000319.html.

[Opp02] Oppenheimer, D. et al [2002]. ROC-1: Hardware Support for Recovery-Oriented Computing, *IEEE Trans. On Computers*, 51:2, February 2002.

[Oracle01] Oracle 9i Flashback Query. *Oracle Daily Feature*, August 13, 2001, available at technet.oracle.com/products/oracle9i/daily/Aug13.html.

[Perrow90] Perrow, Charles [1990], *Normal Accidents: Living with High Risk Technologies*, Perseus Books.

[Petroski92]. Petroski *, H. To engineer is human : the role of failure in successful design* /, Vintage Books,. NewYork, 1992

[Reason90] , Reason J. T.. *Human error*, New York : Cambridge University Press, 1990.

[Reumann00] J. Reumann, A. Mehra, K. Shin et al. Virtual Services: A New Abstraction for Server Consolidation. *Proc. of the 2000 USENIX Annual Technical Conf.*, San Diego, CA, June 2000.

[Rosenblum92] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *Proc. of the 13th Symp.on Operating System Principles*, pages 1--15, October 1991.

[Santry99] D. J. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch. Elephant: The File System that Never Forgets. In *Proc. of the IEEE Workshop on Hot Topics in Operating Systems (HOTOS)*, March 1999.

[Segal88] Z. Segall et al. FIAT fault injection based automated testing environment. *Proc. of the Int'l.Symp.on Fault-Tolerant Computing*, 1988, pp. 102 107.

[Seltzer93] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An Implementation of a Log-Structured File System for UNIX. *Proc. of the Winter 1993 USENIX Conf.*, San Diego, CA, January 1993, 307-326.

[Steininger99] A. Steininger and C. Scherrer. On the Necessity of On-line-BIST in Safety-Critical Applications A Case-Study . *Proc. of the 1999 Int'l.Symp.on Fault-Tolerant Computing*, 208—215, 1999.

[Stonebraker87] M. Stonebraker. The Design of the POSTGRES Storage System. *Proc. of the 13th Int'l.Conf. on Very Large Data Bases (VLDB)*, September 1987.

[Whitten99] Whitten, A. and J. D. Tygar. Why Johnny Can t Encrypt: A Usability Evaluation of PGP 5.0. *Proc. of the 9th USENIX Security Symp* ., August 1999.

[Yemini96] S. Yemini, S. Kliger et al. High Speed and Robust Event Correlation. *IEEE Communications Magazine*, 34(5):82—90, May 1996.