# Self-Repairing Co

BY Armando Fox AND David Patterson

Digital computing performance has improved 10,000-fold in the past two decades: what took a year of number crunching in 1983 takes less than an hour nowadays, and a desktop computer from that era can't match the processing power of one of today's handheld organizers.

We pay a price for these enhancements, though. As digital systems have grown in complexity, their operation has become brittle and unreliable. Computer-related failures have become all too common. Personal computers crash or freeze up regularly; Internet sites go offline often. New software upgrades, designed to augment performance, may leave things worse than they were before. Inconvenience aside, the situation is also an expensive one: annual outlays for maintenance, repairs and operations far exceed total hardware and software costs, for both individuals and corporations.

By embracing the inevitability of system failures, RECOVERY-

- ORIENTED computing returns service faster

Our group of research collaborators at Stanford University and the University of California at Berkeley has taken a new tack, by accepting that computer failure and human operator error are facts of life. Rather than trying to eliminate computer crashes—probably an impossible task—our team concentrates on designing systems that recover rapidly when mishaps do occur. We call our approach recovery-oriented computing (ROC).

We decided to focus our efforts on improving Internet site software. This kind of highly dynamic computing system must evolve and expand quickly in response to consumer demands and market pressures—while also serving users who expect instant access at any time. Consider the example of the Google search engine, which in just a few years has gone from locating hundreds of millions of Web pages of English text to three billion pages in more than 20 languages in a dozen formats, plus images. Meanwhile the number of daily Google searches has grown from 150,000 to 150,000,000—the site is now 1,000 times busier than it was at the outset.

Because of the constant need to upgrade the hardware and software of Internet sites, many of the engineering techniques used previously to help maintain system dependability are too expensive to be deployed. Hence, we expect Internet software to be a good proving ground for our ideas and perhaps a model for other computing systems, including desktop and laptop machines. If ROC principles can help the big animals in the computational jungle, they might do the same for the smaller species [*see box on page 58?*].

Following a proven engineering strategy first adopted during the era of cast-iron-truss railroad bridges in the 19th century, our initial step was to see what we could learn from previous failures. Specifically, we asked: Why do Internet systems go down, and what can be done about it? We were a bit surprised to find out that operator error was a leading cause of system problems. Traditional efforts to boost the dependability of software and hardware have for the most part overlooked the possibility of human mistakes, yet in many cases operators' miscues accounted for more downtime than any other cause.

Operators may face such difficulties because computer designers and programmers have frequently sacrificed ease of use in the quest for better performance. Database software, for example, can require a full-time staff of trained administrators to manage it. Ironically, because hardware and software have grown cheaper over time, operator salaries are now often the biggest expense in running complex Internet sites.

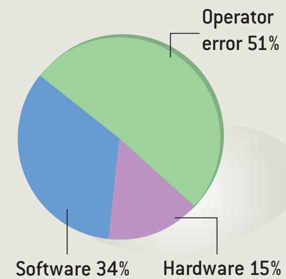
With these issues in mind, our team is exploring four principles to guide the construction of "ROC-solid" computing systems. The first is speedy recovery: problems are going to happen, so engineers should design systems that recover quickly. Second, suppliers should give operators better tools with which to pinpoint

## WHOSE FAULT WAS THAT?

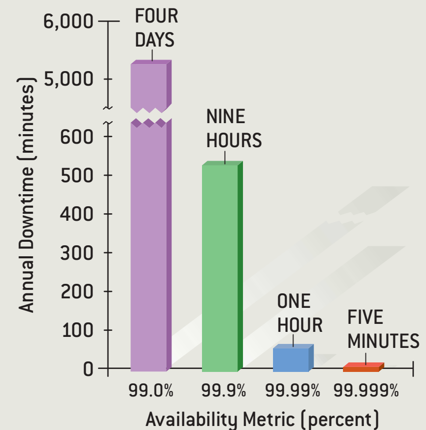
TRADITIONAL ENGINEERING approaches to raising the reliability of computer systems largely ignore the possibility of operator error. But in many cases human mistakes account for more downtime (time during which the system was not functioning) than hardware problems or software bugs. The pie chart (*right*) depicts a breakdown of typical failure causes for three Internet sites.

For many industries, computer system downtime can be costly or even life-threatening. Engineers call the proportion of time a computing system functions correctly its availability, which is measured in "nines" [*graph*]. A system that runs without crashing 99.999 percent of the time, for example, has an availability of "five nines," which corresponds to about two hours of downtime over 25 years of operation. Rather than reducing the number of failures, proponents of recovery-oriented computing advocate methods to shorten the time needed to bring systems back online. Raising availability from two nines to five nines, for instance, shrinks total recovery time from 90 hours to five minutes a year.

REASONS FOR WEB-SITE FAILURES




DOWNTIME IN "NINES"




## Overview/*High-Dependability Computing*

- Despite the undoubted power of today's computers, users continue to be tormented by their systems' stubborn unreliability. Recovery-oriented computing (ROC) design practices could do much to solve this predicament.
- ROC principles—which comprise efforts to engineer rapid-recovery capabilities, software tools to locate faults quickly, "undo" functions to reverse human operators' mistakes, and the means to inject errors to test systems' ability to return to service—may eventually take much of the frustration out of computing.
- Benchmarking programs that evaluate the speed with which computer systems return to full service would encourage industry efforts at improving dependability.

SLIM FILMS (*preceding pages*); JOHNNY JOHNSON (*above*)

Two methods to ACHIEVE HIGH RELIABILITY: ensure fewer breakdowns or bring systems back online faster.

the sources of faults in multicomponent systems. Third, programmers ought to build systems that support an "undo" function (similar to those in word-processing programs), so operators can correct their mistakes. Last, computer scientists should develop the ability to inject test errors; these would permit the evaluation of system behavior and assist in operator training. We plan to release all the ROC-inspired software applications we write to the computing community at no cost.

To foster the adoption of our approach, we also advocate the development and distribution of benchmark programs that would test the speed of a computing system's recovery. This software would measure the computer industry's progress in raising reliability and encourage companies to work toward achieving this end.

## Quick Comeback

MANY A USER REBOOTS his or her personal computer routinely—either preemptively, because the machine is behaving strangely, or reactively, because it has crashed or seized up. Rebooting works for large computers, too, because it wipes the slate clean and fixes a whole class of so-called transient failures—that is, problems that appear intermittently.

Unfortunately, most systems take a long time to reboot and, worse, may lose data in the process. Instead we believe that engineers should design systems so that they reboot gracefully. If one were to look inside a computer, one would see that it is running numerous different software components that work together. During online shopping, for instance, some software modules let customers search through the available merchandise; others permit items to be added to a "shopping cart." Still other software enables the completion of the purchase. Yet another layer of programming choreographs all these functions to produce the overall experience of using the Web site by ensuring that each line of code does its job when required.

Frequently, only one of these modules may be encountering trouble, but when a user reboots a computer, all the software it is running stops immediately. If each of its separate subcomponents could be restarted independently, however, one might never need to reboot the entire collection. Then, if a glitch has affected only a few parts of the system, restarting just those isolated elements might solve the problem. Should that prove ineffective, reinitializing a larger set of subcomponents might work. The trick is to be able to restart one module without accidentally confusing its peers into thinking something is really wrong, which is akin to swapping out the bottom plate in a stack without allowing the other plates to fall—a difficult but doable feat.

George Candea and James Cutler, Stanford graduate students on our team, have focused on developing this independent-rebooting technique, which we call micro-rebooting. Cutler's experience includes building systems that use inexpensive ground receivers assembled from off-the-shelf PCs, low-cost consumer radios and experimental software to capture incoming satellite data. In use, ground-station failures were common, and if a human operator was not available to reactivate the equipment manually, the satellite signal could be lost—and with it, all the data for that orbit.

Last year Candea and Cutler tested micro-rebooting on the ground-station software. They and others modified each receiving-station software module so that it would not "panic" if other subcomponents were reinitialized. The students first consulted the human operators to learn about the most frequent causes of failures and then experimented to determine which sets of subcomponents would have to be reinitialized to cure those specific maladies. They succeeded in automating the recovery process for a range of recurring problems, cutting the average restoration time from 10 minutes to two—fast enough for a ground station that has faltered to reacquire the satellite signal and continue collecting data for the latest orbit.

In dependability lingo, the percentage of time a computer system is functioning correctly is termed its availability, which is typically measured in "nines." A system that is functioning correctly 99.999 percent of the time, for example, has an availability of "five nines," corresponding to about two hours of downtime over 25 years of operation. In contrast, well-managed mainstream computing systems are available only 99 to 99.9 percent of the time ("two to three nines"). Going from two nines to five nines would save almost 90 hours of downtime a year, which is easy to appreciate when downtime costs big money [*see box on page 00*]. Two methods exist to achieve high reliability: ensuring fewer breakdowns or, failing that, bringing systems back online faster.

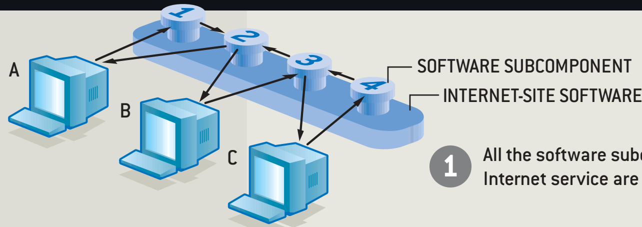
For the satellite ground-station operators, a fivefold-faster return to service was much more valuable than a fivefold increase in the time between failures (better reliability), even though either measure would yield the same level of improved availability. We believe that a variety of computing systems exhibit such a threshold.

Although much effort had to be expended to modify the ground-station software manually, Candea and one of us (Fox) are now investigating whether the technique can be applied in an automated fashion to Web sites programmed using the Java 2 Enterprise Edition, a popular development framework for Internet software.

The most common way to fix Web site faults today is to reboot the entire system, which takes anywhere from 10 seconds (if the application alone is rebooted) to a minute (if the entire machine is restarted). According to our initial results, mi-

# HOW SELECTIVE REBOOTING CUTS DOWNTIME

THREE WEB USERS (A, B and C) access a site. A and B are using subcomponent 2; C does not.




SOFTWARE SUBCOMPONENT

INTERNET-SITE SOFTWARE

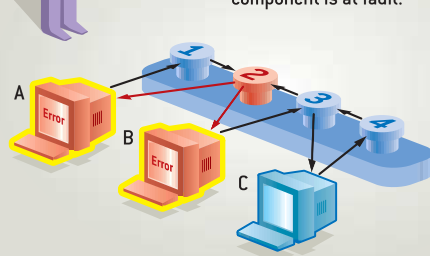
1 All the software subcomponents in the Internet service are working correctly.

## OLD METHOD

?

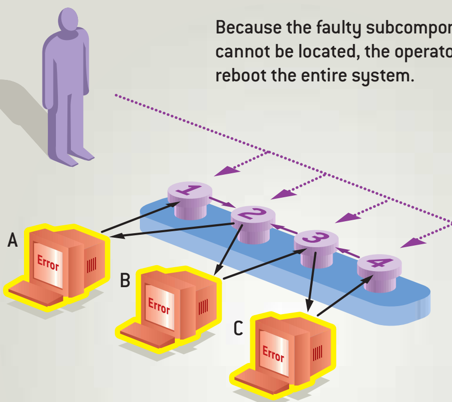
HUMAN OPERATOR

2 Subcomponent 2 fails. Users A and B receive an error message. The human operator cannot determine which component is at fault.

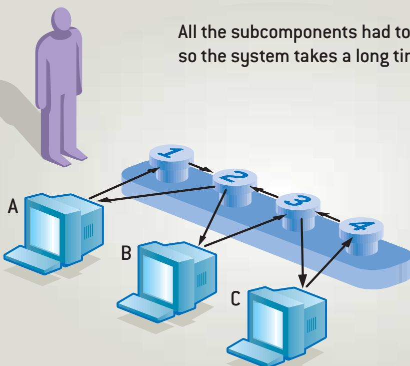



DOWNTIME CLOCK

3 Because the faulty subcomponent cannot be located, the operator must reboot the entire system.




4 All the subcomponents had to be rebooted, so the system takes a long time to recover.

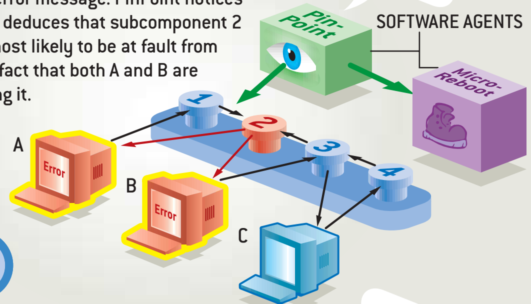



## NEW METHOD

"Looks like subcomponent #2 has failed"

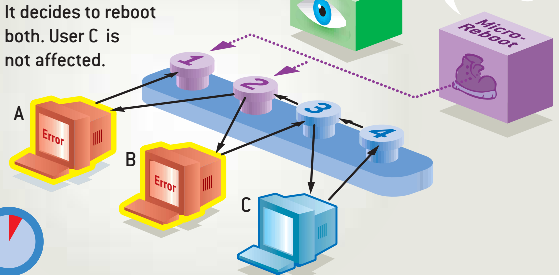
SOFTWARE AGENTS

2 Subcomponent 2 fails. Users A and B receive an error message. PinPoint notices and deduces that subcomponent 2 is most likely to be at fault from the fact that both A and B are using it.

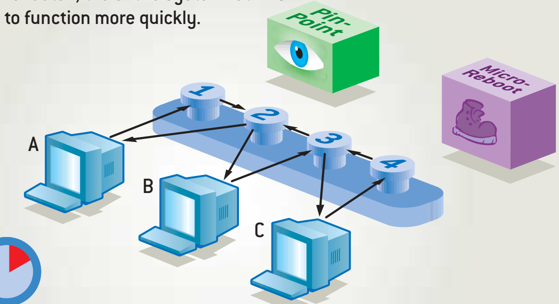



"When #2 fails, #1 is usually affected, too"

3 Micro-Reboot consults its own database and learns that when subcomponent 2 fails, 1 is usually also affected. It decides to reboot both. User C is not affected.




4 Because only two subcomponents were rebooted, the entire system returns to function more quickly.




SAMUEL VELASCO

## Computers may end up 10,000 TIMES FASTER yet no more dependable than today's machines.

cro-rebooting just the necessary subcomponents takes less than a second. Instead of seeing an error message, a user would experience a three-second delay followed by resumption of normal service.

### Pinpointing Problems

IT'S ONE THING to fix known or likely errors, but a related challenge is to find the unanticipated ones. System operators could use assistance in tracking down problems more quickly, the second of our ROC principles.

When building a traditional high-dependability computer system, programmers start with a complete description of all hardware and software elements. They then follow with a carefully constructed failure-analysis tree, which traces the myriad ways the system can break down—so that those may be prevented or, if they do occur, corrected. In contrast to single-source systems, Internet services are heterogeneous, using components from multiple vendors. Further, these modules often change rapidly as the service evolves. Failures frequently arise from unexpected interactions between components, rather than resulting from a bug in a single piece of software. When this kind of dynamic fault occurs, a Web user who happened to be accessing the service at the time may receive an error message.

To help analyze these complex malfunctions, graduate students Emre Kiciman and Eugene Fratkin of Stanford and Mike Chen of Berkeley created PinPoint, a ROC-based computer program that attempts to determine which components are at fault. Every time someone surfs to a PinPoint-enabled Web site, the program traces which software components participated in delivering service to that user. When a particular access request fails—for example, the user gets an error message from the site—PinPoint notes this fact. Over time, the monitoring application analyzes the mix of components that were activated in both failed and successful requests, using standard data-mining techniques. By doing this, PinPoint can find out which components are suspected of causing most of the failures. The additional information gathered by the failure-analysis code slows the system down by at most 10 percent. Unlike the traditional solution—which requires elaborate preplanning every time the software suite changes—PinPoint works with any combination of software components.

### Wiping Away Errors

PERHAPS THE GREATEST challenge in boosting system reliability is ensuring a margin of safety against random errors input by the operator; this rationale underlies our third ROC principle, which concerns the undo command. The first word processors did not have this capability, which made them frustrating, if not terrifying, to use. A single erroneous global substitution could destroy an entire file. The undo function—which affords users the ability to cancel any command—removed the anxiety from word processing.

Operators of today's large computing systems have no such option. When the foundations of information technology were laid, no one considered it important to be able to expunge mistakes. That's because an undo function requires more work to construct, consumes a significant amount of storage space and probably slows systems down somewhat.

To demonstrate a better approach, our group is working on an undo capability for e-mail systems that is aimed at the place where messages are stored. Berkeley graduate student Aaron Brown and one of us (Patterson) have recently completed the prototype of an e-mail system featuring an operator undo utility. We are testing it now [*see box on page 61*].

Suppose a conventional e-mail storage server gets infected by a virus. The system operator must disinfect the server, a laborious job. Our system, however, would record all the server's activities automatically, including discarded messages. If the system gets infected, the operator could employ the undo command to "turn back the clock" to before the arrival of the virus. Software that attacks that virus could then be downloaded. Finally, the operator could "play forward" all the e-mail messages created after the infection, returning the system to normal operation. The newly installed antivirus software would filter all subsequent e-mail traffic. In this way, the operator could undo the damage without losing important messages. To prevent potential confusion

THE AUTHORS

*ARMANDO FOX* and *DAVID PATTERSON* have studied how to improve the reliability of computing systems for many years. Fox has been assistant professor at Stanford University since 1999. As a Ph.D. student at the University of California, Berkeley, he joined with Professor Eric A. Brewer in constructing prototypes of today's clustered Internet services and showing how they can support mobile computing applications, including the first graphical Web browser for personal digital assistants. Fox also helped to design microprocessors at Intel and later founded a small mobile computing company. He received his other degrees in electrical engineering and computer science from the Massachusetts Institute of Technology and the University of Illinois. Patterson has spent the past quarter of a century as a professor at Berkeley. He received all his advanced degrees in computer science from the University of California, Los Angeles. Patterson is best known for work on the simplification of microprocessor architecture, for building reliable digital storage systems and for co-authoring the classic books *Computer Architecture: A Quantitative Approach* and *Computer Organization and Design: The Hardware/Software Interface*. He first published in *Scientific American* two decades ago.

## STRIVING FOR RELIABLITY

COMPUTER SYSTEMS and their "organs"—microprocessors, applications and communications networks—are becoming ever more powerful. But they are also becoming ever more complex and therefore more susceptible to failure. As the costs of administration, oversight and downtime expand in response, scientists and engineers in the computer industry are working to enhance the dependability and reliability of their products. Significantly, many of their efforts aim to take humans (and the errors they inevitably engender) out of the loop.

Concerned about security holes, bugs and other weaknesses in its present product line, Microsoft's management took the unusual step recently of halting software development for an entire month to focus on what it calls Trustworthy Computing. This issue of dependability has grown in importance as more administrators adopt the company's Windows operating system to run Web servers. Operating-system developers at Microsoft attended classes to learn techniques that improve security and reliability for desktop systems and are now refining Windows for the next version, called Palladium. Engineers plan to cull potential weak points from current products while developing new features that boost defenses against hackers.

Little research exists on ways to reduce the lifetime cost of computer ownership—the price individuals and companies pay for running their systems. Programmers at Hewlett-Packard Laboratories and IBM Research are working to cut those expenses by adding new capabilities or developing products that can manage themselves. Hewlett-Packard officials envision a globally networked system of computational and storage resources that monitor, heal and adapt themselves without operator intervention. HP's Planetary Computing project concentrates on developing corporate data centers that could contain as many as 50,000 individual office computers, a collection 10 times as large as today's counterparts.

IBM's scheme borrows ideas from control theory—the use of feedback to stabilize closed-loop systems and artificial intelligence—mimicking or otherwise capturing expert human skills or intelligence to solve complex problems. These concepts will help to create data centers that can diagnose problems on their own, adjust their configurations to match changes in demand, repair themselves and defend against hacker attacks. Drawing an analogy with the body's autonomic nervous system, IBM management calls this goal Autonomic Computing.

When designers of other engineering systems have discovered a propensity for operator error, they often have attempted to remove the need for human input. Removing human operators can lead to a well-established pitfall known as the Automation Irony. Because designers can typically reduce but not eliminate the need for human intervention, such efforts frequently make things worse. That's because engineers generally automate the tasks that are easy, leaving the hard jobs for people. These measures mean that administrators must carry out difficult tasks intermittently on unfamiliar systems—a sure recipe for failure.

Will the path toward truly dependable computing be additional automation, leading to hands-off computing, or will it be streamlined design combined with tools that dramatically improve the effectiveness and productivity of human operators? Only time will tell. *—A.F. and D.P.*

among users—who may notice that some e-mails have been eradicated—the system could send a message saying that notes were deleted during an attempt to stop the spread of a virus.

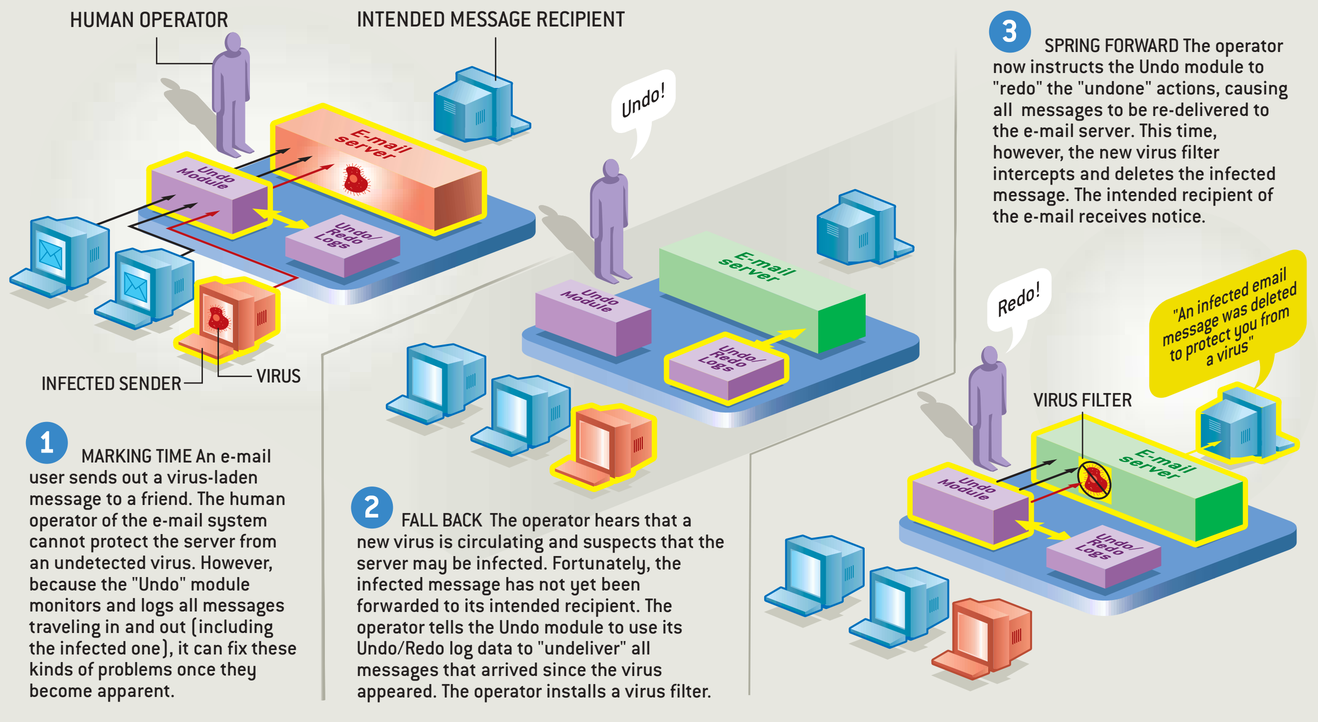
### Injecting Test Errors

LAST AMONG OUR ROC principles is the idea of harnessing errors to do good: we advocate testing the system periodically by inserting artificial faults. This practice would aid in evaluating the recovery performance of a system and also in coming up with new methods to make it more robust. In an industry analogy, designers of microprocessor chips have regularly added circuits to simplify the testing of chips, even though these additions increase chip size and remain unused after the microprocessors leave the factory. Manufacturers consider these test circuits worth the effort; they lower the cost of ensuring that the completed chips work as planned. Part of this benefit arises because the test circuits allow designers to inject "failures" artificially to verify that the chip detects and recovers from them correctly.

Our group proposes a software equivalent for computer systems. When operators employ the selective-rebooting strategy, for example, test errors would help determine which software components to reboot to counter a particular kind of failure. If the problem propagated to only one or two other elements, operators could reboot only those. If the flaw came to involve a great many components, it might be more sensible to reboot the entire system. We have started using error injection to characterize the fault-propagation behavior of Internet sites built using Java 2 Enterprise Edition.

Another version of this software could permit potential buyers to see how a computer system handled failures—a benchmark program that could inform their purchase decisions. Developed by Berkeley graduate students Pete Broadwell, Naveen Sastry and Jonathan Traupman, the Fig application tests the ability of programs to cope correctly with unexpected errors in the standard C library, a part of the operating system used by nearly all software programs. Fig stands for Fault Injection in *glibc* (a version of

SAMUEL VELASCO




## TURNING THE CLOCK BACK ON PROBLEMS

**1** MARKING TIME An e-mail user sends out a virus-laden message to a friend. The human operator of the e-mail system cannot protect the server from an undetected virus. However, because the "Undo" module monitors and logs all messages traveling in and out (including the infected one), it can fix these kinds of problems once they become apparent.

**2** FALL BACK The operator hears that a new virus is circulating and suspects that the server may be infected. Fortunately, the infected message has not yet been forwarded to its intended recipient. The operator tells the Undo module to use its Undo/Redo log data to "undeliver" all messages that arrived since the virus appeared. The operator installs a virus filter.

**3** SPRING FORWARD The operator now instructs the Undo module to "redo" the "undone" actions, causing all messages to be re-delivered to the e-mail server. This time, however, the new virus filter intercepts and deletes the infected message. The intended recipient of the e-mail receives notice.

SAMUEL VELASCO

the standard C library used by numerous programmers).

Error injection would also permit computer programmers to test their repair mechanisms, which is difficult to do. Fig would allow operators to try their hands at diagnosing and repairing failures, perhaps on a small experimental system if not on the real thing. The program has been used several times and is available for no charge on our ROC Web site [*see More to Explore below*].

## Benchmarking Recovery

THE HISTORY OF the computer industry makes us strong believers in the importance of measuring technical progress and publicizing it. When computer firms finally adopted standard benchmarking programs to compare performance (after many years of delay), customers could at last see the relative merits of each product clearly. Companies that trailed in technology were forced to spend more on engineering and subsequently could gauge the effect of their innovations using standard test metrics. The resulting test data led to a cascade of performance improvements.

By focusing on evaluating recovery time, Berkeley graduate students Brown and David Oppenheimer, together with one of us (Patterson), are working to recreate that kind of competition for computer system dependability. Products that are shown to come back from crashes quickest would win greater sales. We envision a test suite that would incorporate failures common in real systems, including everyday errors caused by humans, software and hardware. Prospective buyers could insert these faults into systems and then monitor the time to recovery. It is ironic that current computer marketing efforts more typically quote availability (system downtime) metrics, which are much more difficult to measure than recovery time.

When scientists and engineers focus their efforts, they can achieve amazing progress in a relatively short time. The meteoric, 30-year rise of both computer performance and cost-effectiveness proves it. If the industry continues traveling blindly down the current path, computers in 2023 may end up another 10,000 times faster yet no more dependable than today's machines. But with dependability-enhancing software tools and benchmarks to inspire us, computing may one day become as reliable as users expect it to be. SA

### MORE TO EXPLORE

**To Engineer Is Human: The Role of Failure in Successful Design.** H. Petroski. Vintage Books, 1992.

**The Mythical Man-Month: Essays on Software Engineering.** F. Brooks. Second edition. Addison-Wesley, 1995.

**Managing the Risks of Organizational Accidents.** J. Reason. Ashgate Publishing Company, 1997.

**Autonomic Computing.** IBM Research, 2001. Available at **www.research.ibm.com/autonomic/manifesto**

**Building a Secure Platform for Trustworthy Computing.** Microsoft, August 2002. Available at **www.microsoft.com/security/whitepapers/secure_platform.asp**

**Planetary Computing.** Web site at HP Labs: **www.hpl.hp.com/org/issl/current_research_themes.htm**

**Recovery Oriented Computing.** Web site at University of California at Berkeley and Stanford University : **http://roc.cs.berkeley.edu**

**Turing.** C. Papadamitriou. MIT Press (in press).