acmqueue

Current Issue Past Issues Topics

December 6, 2004

Volume 2, issue 8

PDF

Coping with Human Error

Errors Happen. How to Deal.

Aaron B. Brown, IBM Research

Human operator error is one of the most insidious sources of failure and data loss in today's IT environments. In early 2001, Microsoft suffered a nearly 24-hour outage in its Web properties as a result of a human error made while configuring a name resolution system. Later that year, an hour of trading on the Nasdaq stock exchange was disrupted because of a technician's mistake while testing a development system. More recently, human error has been blamed for outages in instant messaging networks, for security and privacy breaches, and for banking system failures.

Although these scenarios are not as spectacularly catastrophic as their analogues in other engineering disciplines—the meltdown of the Chernobyl nuclear plant or the grounding of the Exxon Valdez oil tanker, for example—their societal consequences can be nearly as severe, causing financial uncertainty, disruption to communication, and corporate instability. It is therefore critical that the designers, architects, implementers, and operators of today's IT infrastructures be aware of the human error problem and build in mechanisms for tolerating and coping with the errors that will inevitably occur. This article discusses some of the options available for embedding "coping skills" into an IT system.

THE INEVITABILITY OF HUMAN ERROR

Human error happens for many reasons, but in the end it almost always comes down to a mismatch between a human operator's mental model of the IT environment and the environment's actual state. Sometimes this confusion arises from poorly designed status feedback mechanisms, such as the perplexing error messages that Paul Maglio and Eser Kandogan discuss elsewhere in this issue (see "Error Messages: What's the Problem?" on page

50), but other times the mismatch simply arises from a lack of experience on the operator's part, or worse, to quirks of human cognitive processing that can obstinately steer even an experienced operator toward the wrong conclusion.1 Regardless of the source, however, psychology tells us that mental-model mismatches, and thus human error, are inevitable in the rapidly changing environments characteristic of IT systems.

Particularly disconcerting is that people, with their unique capacity for (often-unintentioned) ingenuity, manage to break even systems designed for dependability and integrity. Take RAID (redundant array of inexpensive [or independent] disks) systems, for example. Hardware vendors have had to go to extreme lengths in high-end RAID products to prevent human operators from removing the wrong disk after a disk failure—often building in hardware interlocks that prevent the removal of working disks. Lower-end and software-implemented RAID systems have no such luxury, and they suffer for it.

In some simple experiments we carried out to investigate the magnitude of this problem, we asked five people to perform a basic repair task: replacing a failed disk in a software RAID system (see sidebar, "A Human Error Case Study"). All five people participating in the experiments were trained on how to perform the repair and were given printed step-by-step instructions. Each person performed several trials of the repair process. We found that, even on this simple maintenance task with full instructions and in a low-stress setting (no alarms, no angry customers or bosses breathing down their necks), the human operators made fatal errors that caused data loss up to 10 percent of the time. What we are left to conclude is that even the best-intentioned reliability technologies (such as RAID) can become impotent in the face of the human capacity for error.

So, is it hopeless? Is there anything that we, as the designers, implementers, and operators of IT systems, can do to prevent human error from permanently damaging data or causing outages?

COPING WITH HUMAN ERROR

In fact, there are several possible approaches for coping with human error, each with its own strengths and weaknesses. We can divide the approaches into four general categories:

- Error prevention
- Spatial replication
- Temporal replication
- Temporal replication with reexecution

The first category attempts to prevent human errors from occurring or at least to reduce their frequency. The remaining three categories handle errors that have already occurred by providing different forms of reversibility, or the ability to remove the effects of an existing human error from system state. The most human-error-resilient systems will implement more than one of these techniques, providing defense in depth against whatever errors challenge their integrity.

Error Prevention

There are two ways to prevent human error from affecting a system: either keep people from making the errors (error avoidance) or stop the errors from reaching the system (error interception). Both techniques require that the types of possible errors be anticipated; as a result, neither is extremely effective—people are simply too good at finding unanticipated ways to make mistakes.

Error avoidance is typically accomplished through user interface design or training. In the former case, the user interface is constructed to block potential errors. For example, wizards can guide a user through predefined tasks, or human input can be removed entirely via automation. These approaches tend not to be very successful in practice, however, since operators often end up bypassing wizards or automated aids to accomplish tasks that went unanticipated by the user interface's designer.

Instead of blocking errors at the interface, an alternative is to train human users not to make errors. Training works by developing the human's mental model of the computer system, thereby preventing the mental-model mismatches that are a major source of error. It is only as effective as it is extensive, however: training must focus on concepts—not just procedures—to help build the broadest mental models and must evolve with the system to ensure that those mental models are kept up to date. The best training programs are extensive, frequent, and designed to force operators out of their comfort zones; technology can help achieve these goals by integrating training periods into a system's normal operation.

When error avoidance fails, an alternative is to let people make mistakes but prevent those mistakes from reaching the system. A good example of this error interception can be seen in the way that many e-mail clients can be configured to batch and delay sending outgoing mail for several minutes, providing a recovery window during which an erroneously or rashly sent message can be recalled, discarded, or edited. This and similar buffering-based strategies are particularly effective because they leverage the human ability to self-detect errors: psychologists report that 70 to 86 percent of errors can be detected immediately after they are committed, even if they cannot be anticipated.2

Unfortunately, error interception has its limitations. It works only when human operations are asynchronous and can be safely delayed to provide a recovery window. It will not work in situations where system state changes quickly, rendering buffered commands obsolete by the time they're executed. Error interception can also create confusion by breaking the immediate-feedback loop that people expect in interactive scenarios—imagine the havoc that a two-minute command execution delay would cause for an operator working at a command line to troubleshoot a system outage.

Spatial Replication

When prevention inevitably fails, other techniques must step in to help cope with the resulting human error. Spatial replication is one such technique that can help with serious, state-damaging errors, as well as simpler operational errors that do not corrupt state, such as an accidental component shutdown. The basic idea behind spatial replication is to create multiple replicas of a system or service, all of which maintain their own (synchronized) copies of the system's key data. RAID is a simple form of spatial replication that can tolerate a single error to one of its replicas (one disk in the RAID set) by reinitializing and reconstructing the erroneous replica from the remaining disks—and note that by extending a RAID system's spatial replication factor to include an additional copy of the data, it could even tolerate the common but fatal human error of removing the wrong (single) disk during repair.

More advanced schemes, such as Byzantine fault tolerance,3 add sophisticated voting algorithms to detect any replicas whose behavior does not match the majority behavior. Under these schemes, if a human error affects one or a minority subset of the replicas, its effects can be detected in comparison with the remaining (majority) set of replicas, and the affected replicas can be discarded and rebuilt from the majority set. So, for example, if an operator accidentally shuts down or even corrupts one of the replica nodes, the remaining replicas can continue to provide service while the failed node is restarted and resynchronized, effectively reversing the effects of the error.

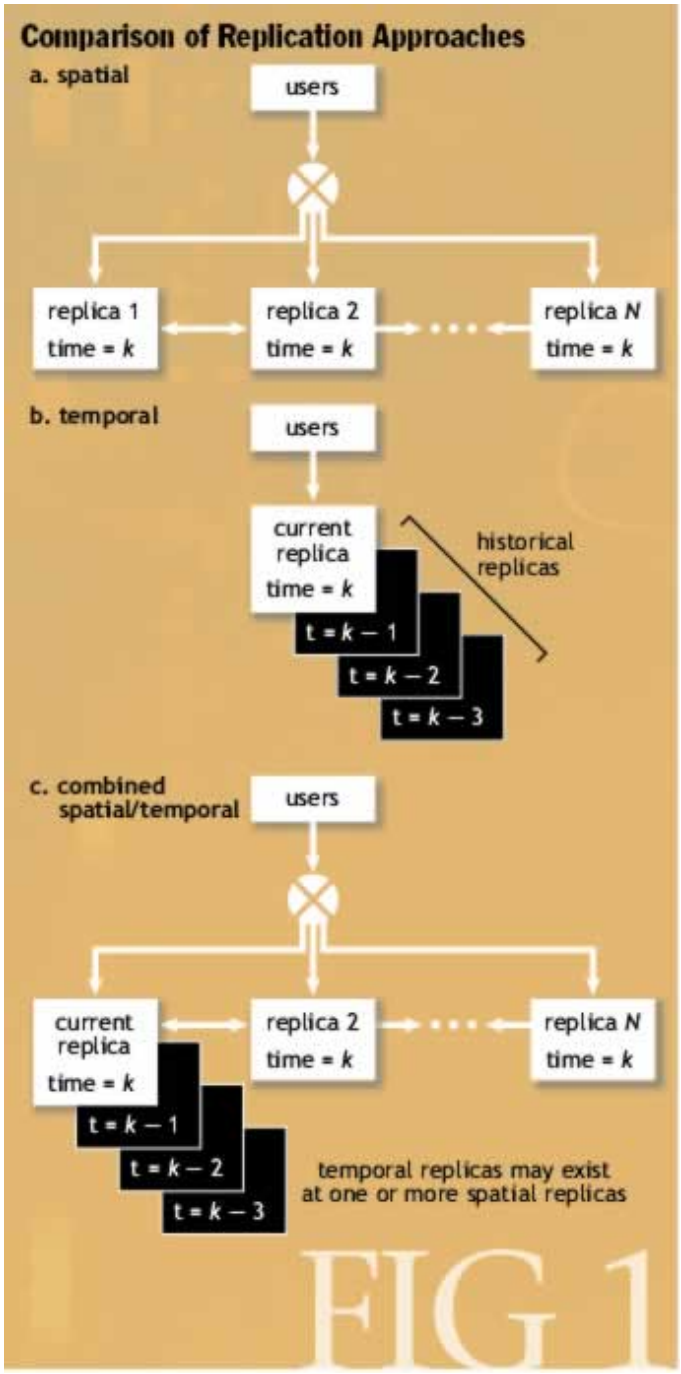
The weakness of spatial replication is that it works only when human errors affect a minority of the replicas; any error that affects a majority becomes accepted as the ostensibly correct state of the system, and the minority replicas (which are actually the error-free copies) are treated as erroneous and ignored. Spatial replication is therefore useless to defend against error in critical systemwide operations such as reconfiguration, application deployment, or software upgrades: these operations must be made to all nodes at once, bypassing the protection of the spatial replication.

Temporal Replication

A variant of spatial replication that addresses some of these limitations is temporal replication. As in spatial replication, temporal replication involves maintaining several copies of the system or service, each with a replica of system state. The difference is that in temporal replication, the replicas are not synchronized. Instead, a current replica represents the live state of the system, and a series of historical replicas represent snapshots of different states in the system's history. Only the current replica is actively used to service requests to the system, and human operator intervention is restricted to that current replica. Figures 1a and 1b illustrate the difference between spatial and temporal replication.

With temporal replication, when a human error corrupts or otherwise affects the current replica, the system can be restored from one of the older replicas. Since the operator touches only the current replica—even for systemwide changes such as application upgrades—the older replicas are immune to error and can be used to recover from even severe errors such as destruction of all data in the current replica. The downside, of course, is that recovering to an older replica means that any data created or modified since that replica was created is irretrievably lost. This is the major weakness of temporal replication.

A secondary weakness is that temporal replication copes only with human errors that affect state. If the human operator causes an outage by accidentally shutting down a production server, there is little that temporal replication can do; however, it can be combined with spatial replication as in figure 1c to provide broader-spectrum protection against state-affecting and operational errors.



Despite its limitations, temporal replication is widely used today. For example, traditional offline system backups are a form of temporal replication: the backup tape contains a snapshot of system state at a past time, and that snapshot can be restored if a human error corrupts the current system. File system snapshots, such as those maintained by Network Appliance filers, or split-mirror procedures in RAID-1 arrays, are also forms of temporal replication that can be used for human error recovery.

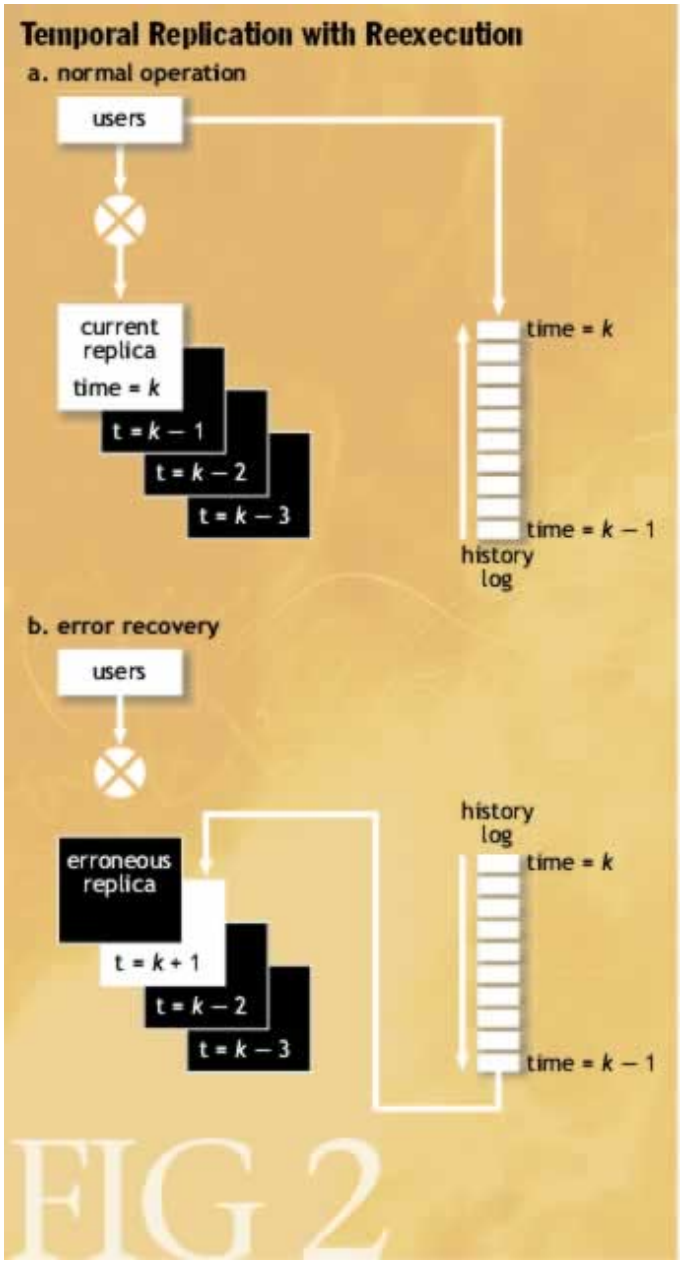
Temporal replication can also be used at the application level. For example, an enterprise application provider might maintain two or more copies of the application's data, with each copy temporally delayed by several hours from the last. An implementation of this strategy

could take the form of queuing copies of all changes to the data during the delay period, then later applying those changes to the delayed copy. For applications built on top of a database, this can be accomplished by using the primary copy's database log to periodically update the temporally delayed replicas.

For applications where exact data consistency is less important, temporal replication can also be used in the large, with entire data centers acting as temporally delayed backups of each other through a similar mechanism of database log-shipping. Finally, VM (virtual machine) technology offers a promising opportunity to make temporal replication available at a finer granularity and lower overhead than these other approaches—essentially, VM snapshots can be periodically taken and quickly reactivated to restore the system after a human error.

Temporal Replication with Reexecution

None of the common implementations of temporal replication truly copes with human errors, since in recovering from errors they discard potentially significant amounts of recent data. A fourth approach removes that limitation by combining temporal replication with reexecution. Essentially, a separate history log is kept containing all changes to the system following the time when the last temporal replica was taken. Then, when a human error occurs, the system copes with it by switching to the old replica and reexecuting the operations in the log to bring that replica up-to-date, effectively reversing—or undoing—the human error, as shown in figure 2. This approach is similar to how recovery is performed in databases,4 but adds the new wrinkle of editing the human error out of history as part of the reexecution.



Of course, manipulating the past history of a system's execution has significant consequences. Notably, editing out the error can cause paradoxes in which the state of the older replica at the end of reexecution is inconsistent with the behavior already seen by users of the erroneous replica. For example, a paradox in an e-mail server application might occur when an erroneously delivered message is read by its (unintended) recipient, then that message later disappears when the human error that caused its misdelivery is undone. A paradox in an auction application might cause the top bidder (or worse, the winner) to change unexpectedly following an undo of an erroneous change to the bidding algorithm.

The choice of how to handle paradoxes is inherently application-specific. Some applications—particularly those without persistent per-user states, such as search engines—may simply not care and choose to leave paradoxes visible. Other applications may choose not to reexecute

operations that induce paradoxes, preferring consistency over lost work; these applications essentially choose between plain temporal replication and temporal replication with reexecution on a per-operation basis. But most applications, particularly those with human end users, can choose a middle ground, where all operations are reexecuted (to minimize lost work) and any visible paradoxes are compensated for in an application-specific manner.

Remarkably, most applications designed to interact with people already have compensation mechanisms used manually to deal with inappropriate behavior and errors in human-driven processes; these existing mechanisms can be harnessed to enable the recovery benefits of temporal replication with reexecution. For example, the previously mentioned auction paradox can be compensated for by using existing policies for bid retraction and auction result contestation. Typically, a trade-off exists between the cost of the compensation (e.g., absorbing the cost of an incorrect auction) and the cost of the original incorrect behavior. Application designers must assess the expected probability of paradoxes and relative costs and benefits of compensations before settling on an approach to temporal replication with reexecution.

To explore the benefits and consequences of implementing the reexecution approach on a real application, we developed a prototype human-error-undo mechanism for e-mail servers.5 Our implementation logs all incoming IMAP and SMTP traffic, recording e-mail deliveries and changes made to users' mailboxes. When the operator discovers a human error and requests that it be undone, our implementation rolls the system state back to a historical snapshot, then replays the logged mailbox update stream. As it reexecutes updates, it checks for paradoxes by comparing any externally visible output with a record of historical output. When differences are significant, they are compensated for by delivering additional explanatory messages to the effected user.

Our undo implementation for e-mail servers makes it possible for a human operator to quickly reverse erroneous changes made to the e-mail server's operating system, application software, and configuration state—for example, a failed mail server upgrade or a spam filter misconfiguration—without losing user data such as incoming e-mail or mailbox updates. When we evaluated the prototype mechanism in user studies, we found that it made human error recovery easier and resulted in significantly less lost user data than traditional temporal-replication-only schemes (such as backups).6

While temporal replication with reexecution seems to be the best approach we have seen so far—it copes with even systemwide human error without losing data—it does suffer several weaknesses. First, like plain temporal replication, it can do nothing for human errors that do not affect state (such as accidentally shutting down a server). It is also the most challenging of

our recovery approaches to implement. It requires careful, application-specific reasoning about paradoxes, and corresponding implementation of compensation mechanisms. Care is needed to preserve the causal ordering of events when creating and reexecuting the history log—particularly challenging in a distributed context. Finally, reexecution can be expensive in terms of time, particularly on a heavily loaded system, and the history log can consume large amounts of storage. As a result, undo-like capabilities are most useful in conjunction with error prevention techniques, where they can serve as an expensive but powerful and trustworthy second line of defense.

GUIDELINES FOR COPING WITH HUMAN ERROR

Human error is a significant force in IT systems, with the proven potential to affect IT-based business through failures, outages, and damage to data. There are several different techniques for coping with human error, ranging from trying to prevent errors with avoidance and interception to using various forms of replication to deal with errors after the fact. Each of these techniques has its own advantages and disadvantages, summarized in table 1; probably the most powerful is temporal replication with reexecution, but it comes at the cost of implementation complexity and resource overhead.

TABLE 1

Technique	Applicability*	Pros	Cons
Prevention			
Training	OP, SA	Eliminates errors before they occur	Difficult to keep relevant, conceptual, and comprehensive; ineffective on unanticipated situations
Automation	OP, SA	Removes opportunity for error	Useful only for pre-anticipated tasks
Error-aware user interface	OP, SA	Removes opportunity for error	Useful only for pre-anticipated tasks
Buffering operations	OP, SA	Provides recovery window to cancel erroneous operations	Useful only for operations that can tolerate significant extra delay; requires good detection mechanisms
Coping			
Spatial replication	OP, SA	Automatic, transparent mitigation of human error	Effective only when error affects minority of replicas; requires significant extra resources
Temporal replication	SA	Effective on systemwide state-affecting errors	Recent data irretrievably lost when reverting to historical replica
Temporal replication with reexecution	SA	Copes with state-affecting errors without losing recent data	Slow and resource-intensive; difficult to implement correctly

*OP stands for operational errors that do not affect state, like shutting down a machine, and SA stands for state-affecting errors, like deleting data.

For the architect trying to design or implement a human-error-tolerant computer system, the best advice when looking at the different mechanisms in table 1 is to think in terms of multiple lines of defense: a system can cope effectively with the broad spectrum of possible human errors only by drawing on more than one of the techniques discussed here.

The first line of defense is avoidance: using automation, error-aware interface design and comprehensive, ongoing training where possible to keep errors from happening in the first place. To catch the errors that inevitably will slip through, the ideal system adds a layer of

interception, buffering those operations that can tolerate the extra latency to provide a recovery window from error, or even executing them on a virtual copy of the system before committing their effects to the real iron.

When these prevention techniques fail, the ideal system next draws on the replication-based approaches, using spatial replication to handle operational errors and ultimately relying on temporal replication with reexecution as a heavyweight, but trustworthy, last line of defense against state-affecting errors.

Unfortunately, the reality of today's IT is that we have a long way to go before human error "coping skills" become commonplace: most of the techniques discussed here exist in some form, but they are far from ubiquitous, and they tend to be difficult to implement properly from scratch. Since human operator error is not likely to disappear anytime soon as a determining factor in the dependability of IT systems, however, it is crucial that we continue to advance the state of the art in human error tolerance. This should be done both by pursuing and enhancing the approaches presented here and by developing innovative new approaches that can cope with human error effectively, efficiently, and at low implementation cost.

REFERENCES

1. Reason, J. 1990. Human error. Cambridge: Cambridge University Press.

2. See reference 1.

3. Castro, M., and B. Liskov. 2002. Practical Byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems, 20(4): 398-461.

4. Gray, J., and A. Reuter. 1993. Transaction processing: Concepts and techniques. San Francisco: Morgan Kaufmann.

5. Brown, A. B., and D. A. Patterson. 2003. Undo for operators: Building an undoable e-mail store. Proceedings of the 2003 Usenix Annual Technical Conference. San Antonio, TX (June).

6. Brown, A.B. 2003. A Recovery-oriented approach to dependable services: Repairing past errors with system-wide undo. Ph.D. Dissertation, University of California, Berkeley, Computer Science Division Technical Report UCB//CSD-04-1304 (December).

A Human Error Case Study

To investigate the potential for human error in an IT environment, we carried out a simple experiment using RAID storage. We set up software RAID-5 volumes on three different test systems, each with a different operating system: Windows 2000, Red Hat Linux 6.0, and Solaris 7 (Intel) with Solstice DiskSuite 4.2. We then recruited and trained five technically savvy people to play the role of system administrator. Each person was randomly assigned to one of the test systems and given the charge of repairing any disk errors that occurred.

The experiment consisted of a series of trials. In each trial we simulated a simple stop failure on one disk in the RAID volume and observed whether the person participating was able to repair the volume by replacing the failed disk with a spare. Once the person completed the repair (or gave up), the trial ended and we moved on to the next one after restoring the test system to its initial operating state. Each person completed between six and nine trials on each test system, depending on how quickly the failures were handled.

The aggregated results from the experiment are shown in table 1.

TABLE 1

		Trials with Human Errors		Human Error Rate	
RAID System	Total Trials	fatal errors	any errors	fatal errors	any errors
Windows	35	1	3	2.9%	8.6%
Solaris	33	0	6	0.0%	17.1%
Linux	31	3	7	9.7%	22.6%

On the Linux system, our users made fatal errors on 3 of 31 trials—nearly a 10 percent fatal error rate! The other systems had smaller fatal error rates, but all three experienced significant overall error rates—between 8 and 23 percent, counting fatal and nonfatal errors. Particularly interesting was the fact that error rates remained nonzero on two of the systems even as the people using them gained experience and familiarity in later trials, as shown in figure 1.

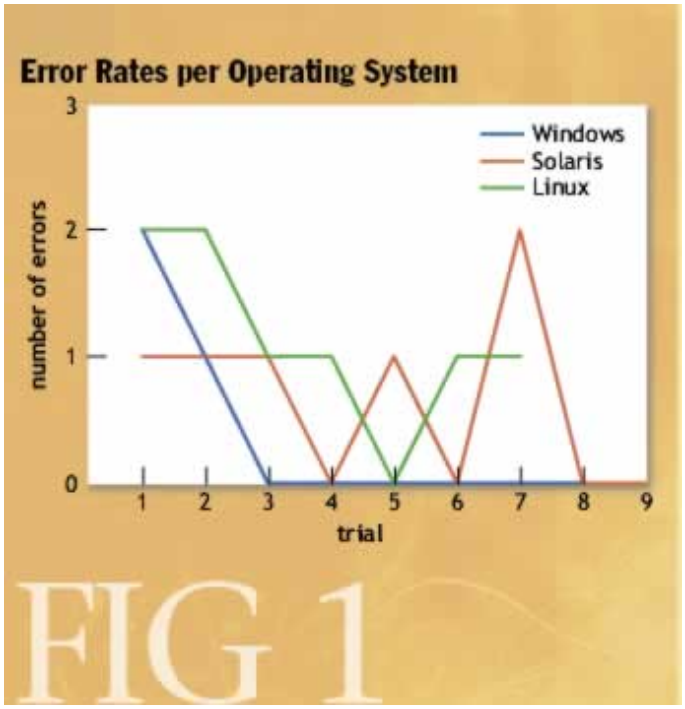


FIG 1

The obvious conclusions from our experimental data are that human error is a key impediment to reliability, that error does not go away with training or familiarity, and that system design can impact error rate (as seen in the differences among the three tested systems).

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

AARON B. BROWN is a research staff member in the Adaptive Systems department at IBM's T.J. Watson Research Center. His research interests include understanding the role and impact of human system managers in large-scale IT infrastructures, quantifying and reducing IT management complexity, and benchmarking nontraditional aspects of IT systems. He is also one of the architects of IBM's Autonomic Computing effort. Before joining IBM, he co-founded the Recovery-oriented Computing (ROC) project at the University of California, Berkeley, with his Ph.D. advisor (and now ACM President) David A. Patterson.

Originally published in Queue vol. 2, no. 8—
Comment on this article in the ACM Digital Library

More related articles:

Phil Vachon - **The Keys to the Kingdom**
An unlucky fat-fingering precipitated the current crisis: The client had accidentally deleted the private key needed to sign new firmware updates. They had some exciting new features to ship, along with the usual host of reliability improvements. Their customers were growing impatient, but my client had to stall when asked for a release date. How could they come up with a meaningful date? They had lost the ability to sign a new firmware release.

Peter Alvaro, Severine Tymon - **Abstracting the Geniuses Away from Failure Testing**
This article presents a call to arms for the distributed systems research community to improve the state of the art in fault tolerance testing. Ordinary users need tools that automate the selection of custom-tailored faults to inject. We conjecture that the process by which superusers select experiments can be effectively modeled in software. The article describes a prototype validating this conjecture, presents early results from the lab and the field, and identifies new research directions that can make this vision a reality.

Pat Helland, Simon Weaver, Ed Harris - **Too Big NOT to Fail**
Web-scale infrastructure implies LOTS of servers working together, often tens or hundreds of thousands of servers all working toward the same goal. How can the complexity of these environments be managed? How can commonality and simplicity be introduced?

Steve Chessin - **Injecting Errors for Fun and Profit**
It is an unfortunate fact of life that anything with moving parts eventually wears out and

malfunctions, and electronic circuitry is no exception. In this case, of course, the moving parts are electrons. In addition to the wear-out mechanisms of electromigration (the moving electrons gradually push the metal atoms out of position, causing wires to thin, thus increasing their resistance and eventually producing open circuits) and dendritic growth (the voltage difference between adjacent wires causes the displaced metal atoms to migrate toward each other, just as magnets will attract each other, eventually causing shorts), electronic circuits are also vulnerable to background radiation.