Rewind, Repair, Replay: Three R's to Dependability

Aaron B. Brown and David A. Patterson
University of California at Berkeley, EECS Computer Science Division
387 Soda Hall #1776, Berkeley, CA, 94720-1776, USA
Contact author: Aaron Brown, abrown@cs.berkeley.edu

Abstract

Motivated by the growth of web and infrastructure services and their susceptibility to human operator-related failures, we introduce system-level undo *as a recovery mechanism designed to improve service dependability. Undo enables system operators to recover from their inevitable mistakes and furthermore enables* retroactive repair *of problems that were not fixed quickly enough to prevent detrimental effects. We present the "three R's", a model for undo that matches the needs of human error recovery and retroactive repair; discuss several of the issues raised by this undo model; and describe a prototype implementation of system-level undo in an email service system.*

1 Introduction

The need for a dependable computing infrastructure has never been more urgent. The world is shifting to a model where data is stored and maintained in centralized servers and doled out to clients via network services; we have seen the beginnings of this trend over the last few years with the growth of Internet-based services, portals, and e-commerce. As the trend accelerates further with the deployment of technologies such as pervasive wireless networking, mobile devices, .NET, and J2EE, the social and financial impact of dependability problems in the infrastructure promises to be enormous.

One of the primary impediments to infrastructure dependability today is the human operator (a.k.a. system administrator). Human operator error is the leading cause of outages across a spectrum of systems ranging from Internet services to the US telephone network [2] [4] [7]. When operators don't create outages, they often compound them by not responding quickly enough to fix the problems before damage is done.

What are we to do? One option is to eliminate the human operator from the system. This may work for small embedded devices, but it doesn't apply to the large systems with hard state that make up the network service infrastructure of the future. Furthermore, studies from psychology and system accident theory leave little room for debate: attempts to automate away human operators in large systems invariably fail due the *automation irony*[1] [8].

The only viable alternative, then, is to build infrastructure systems that accept and compensate for the inevitable weaknesses of their human operators. Future systems should recover easily from operator mistakes, give the operator an environment in which trial-and-error reasoning is possible, and harness the unique human capacity for hindsight by allowing *retroactive* repairs once problems have been manifested. There is a recovery mechanism that has these properties, and it is one that we use every day in our word processors and spreadsheets: *undo*. Unfortunately, undo as a recovery model has been limited to the application level, where it is insufficient to tackle the operational problems that plague infrastructure systems: operator errors made during upgrades and reconfiguration, external virus and hacker attacks, and unanticipated problems detected too late for their effects to be contained.

To address these problems, we introduce *system-level undo*, an undo-based recovery model that covers all levels of the system, not just the application. The crux of our undo model is that it disambiguates user intentions from their manifestations in system state, allowing the undo mechanism to repair problems in system state without losing user data.

Although the technological underpinnings of our undo mechanism are simple—a combination of non-overwriting storage and logging of user inputs—the policy choices in an undo implementation are complex. Most challenging is dealing with the problem of *externalized state*, where erroneous changes to system state that have been made visible to the end-user need to be revoked as part of the undo process. Another challenge lies in deciding what state should be made recoverable through undo: a good undo mechanism will preserve user data while allowing arbitrary repairs to system state, and drawing the dividing line between recoverable and non-recoverable changes is non-trivial. A final chal-

[1] The automation irony captures two problems with automation. First, automation shifts the burden of correctness from the operator to the designer, requiring that the designer anticipate and correctly address all possible failure scenarios. Second, as the designer is rarely perfect, automated systems almost always can reach exceptional states that require human intervention. These exceptional states correspond to the most challenging, obscure problems; psychological studies routinely show that humans are most prone to mistakes on these types of problems, especially when automation has eliminated normal day-to-day interaction with the system.

lenge is in constructing an undo system that works at multiple granularities: cluster-wide, per-system, and per-user.

2 The Three R's: an Undo Model Akin to Time Travel

To support retroactive repair and recovery from operator error, we propose an undo model based on a 3-step process that we call "the three R's": ***R**ewind*, ***R**epair*, and ***R**eplay*. In the *rewind* step, all system state (including user data 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 is allowed to make any changes to the system he or she wants. Changes could include fixing a latent error, installing a preventative filter or patch, retrying an operation that was unsuccessful the first time around (like a software upgrade of the application or OS), or even simply omitting an action that caused problems (like an accidental "rm -rf *"). Finally, in the *replay* step, the undo system re-executes all *end-user interactions* with the system, allowing them to be reprocessed with the changes made during the repair step.

A convenient metaphor for understanding the 3R undo model is to think of it as time travel. In a common portrayal of time travel in science-fiction, a protagonist travels back through time to right a wrong. By making changes to the timeline in a past time frame, the protagonist fixes problems and averts disaster, and the effects of those changes are instantaneously propagated forward to the present. The 3R undo model offers a similar sequence of events. The rewind step is the equivalent of traveling back in time, in this case to the point in time before an error occurred. The repair step is equivalent to changing the timeline: the course of events is altered such that the error is repaired or avoided. Finally, the replay step propagates the effects of the repair forward to the present by reexecuting—in the context of the repaired system—all events in the timeline between the repairs and the present. Since the events are replayed in the context of the repaired system, they reflect the effects of the repairs and any incorrect behavior resulting from the original error is cancelled out.

All three steps in the 3R model are required to achieve effective retroactive repair and recovery from operator error. Without rewind, recovery would not be possible since the state changes induced by the error could not be revoked. Without repair, the error itself could not be corrected. Without replay, all user interactions and updates between the rewind point and the present would be lost.

There are several existing systems and techniques that offer subsets of the 3R model, but none offers full 3R semantics at the system level. For example, backup/restore or checkpointing schemes [1] [3] [5] offer rewind/repair or rewind/replay, but deny the ability to roll forward once changes have been made. Recovery systems for transactional relational databases use rewind/replay to recover from crashes, deadlocks, and other fatal events, but again do not offer the ability to interject repair into the recovery cycle [6]. Databases are still an interesting example because they have the mechanisms needed for 3R undo: checkpoints and redo logs. However, these mechanism are not automatically combined in such a way as to make 3R undo possible. Even if they were, they do not operate at the system level, disallowing the ability to recover from unsuccessful software upgrades or operator changes to the OS configuration. As will be discussed in more detail in Section 4, we can implement an automatic system-level undo by using similar mechanisms—logging of user interactions and checkpointable/non-overwriting storage—but at a lower level of the system.

3 Challenges in the 3R Undo Model

3.1 Delineating state preserved by replay

When an undo is carried out under the 3R undo model, all state changes made since the undo point are wiped out during the rewind step. It is the responsibility of the replay step to restore all state changes that are important to the end user. Defining exactly what state this encompasses is tricky, especially when repairs could radically change the physical representation of state (*e.g.*, an upgrade of a mail server that rewrites the on-disk mailbox format). Ideally, the replay mechanism should preserve end-user *intent* rather than specific state changes. For example, in an undoable e-mail system, a user's act of deleting a message should be recorded as "delete message with Message-ID *x*", not "alter byte range *m* – *n* in file *z*". By tracking user updates at an intentional level, the replay system has the best hope of preserving the state that the user cares about while leaving as much flexibility for repair as possible.

In the network service environment that we are targeting, users interact with the system through standardized application protocols, so the easiest way to achieve intentional tracking of user updates is to intercept and track user interactions at the protocol level. Most network service protocols are stable, well-defined, and divorced from any particular internal state representation: SMTP and IMAP for email, JDBC/SQL for databases, and XML/SOAP for the emerging online application frameworks, for example. Tracking interactions at the level of these protocols automatically provides an record of user intent that is independent of the details of the application itself; in fact, it should be pos-

sible to completely swap out one server implementation for another during the repair phase and still be able to replay user interactions, as the protocol itself is unlikely to change across implementations. Of course, in those cases where no standard protocol is used or where the protocol is quickly-evolving, user interactions must be tracked at a higher level. Finally, note that protocol-level tracking argues for a implementation of 3R undo as a proxy layer wrapping an existing service application; this is in fact how we are building our undoable email system prototype, described further in Section 4.

Up to now we have defined replay as only affecting user state, but have ignored the issue of whether repairs are tracked. As with user updates, to track and replay repairs the undo system would have to log the intent of the repairs, not their effects on state. While this is feasible for protocol-limited user interactions, it becomes a nightmare when the set of possible changes is limited only by the operator's human ingenuity, not a list of protocol commands. Thus for feasibility reasons we make the choice to not allow replay of repairs in our undo model; we may explore the possibility in future work.

3.2 Externalized state

A favorite device of time-travel fiction is the time-paradox, where alterations to the past timeline effect unexpected changes in the present. In these paradoxes, the time-traveling protagonist, whose memories are typically isolated from the altered timeline, sees the "new" present as inconsistent.

The same problem plagues system-level undo: during the undo cycle, repairs change the past state of the system, and replay propagates those changes forward to produce a new version of the present that is likely inconsistent with the view of the present seen before the undo cycle. For example, in an email system, a retroactive repair could consist of installing a spam- or virus-blocking filter. When replayed forward, formerly-delivered mail messages might be squashed by the new filter. A user who had read, forwarded, or replied to those messages would see the system as inconsistent with his or her expectations once the undo cycle was complete. We call this problem of inconsistencies the "externalized state" problem because inconsistencies arise only when state that has formerly been made visible to an external entity (*i.e.*, the user) is altered by the undo cycle; state that has not been externalized cannot cause inconsistencies.

As with the similar output commit problem discussed in the checkpointing literature [3], there is no complete fix for the externalized state problem; possible solutions involve managing the inconsistency rather than eliminating it. The easiest solution is to simply ignore the inconsistency, assuming that the user will tolerate it. This approach is best suited for minor inconsistencies in applications with relaxed semantics, for example when the inconsistency causes reordering of message delivery in an email system or changes item availability estimates in an e-commerce system. When the inconsistency is too large to ignore, the best solution is to use compensating or explanatory actions to help the user adjust to it. For example, in our email scenario above, we could replace the removed message in the user's mailbox with a new message explaining why the original message was removed; this technique is used effectively today by virus-scanning email gateways.

When the entity that externalizes state is not an end-user but another computer system, there are more powerful solutions available. One, which removes inconsistencies entirely, is to expand the boundary of the undo system to encompass the external system. This can be done by propagating undo requests across system boundaries so that when externalized state is changed the external system is rolled back and replayed with the new version of the externalized state. This approach must be used with care, as the boundary may have to be drawn arbitrarily large to completely tolerate the inconsistency; however, a small increase in boundary size may reduce the inconsistency to the point where it can be tolerated by a human user. Another approach when the externalizer is a computer is to delay the execution of externalizing actions for a given time period; during this undo window, the actions can be rolled back and altered without inconsistency. This approach is limited to cases where the actions are asynchronous and not time-critical, like delivering email to an external system.

Note that any of the solutions that we have discussed for the externalized state problem require that the undo system be aware of any undo-induced inconsistencies and their magnitude. This can be a significant implementation challenge; we will discuss our approach in Section 4.

3.3 Granularity of undo

To be most useful, undo as a recovery mechanism should be available at multiple granularities. A user might want to use undo to recover from a mistake affecting only his or her state; it should not be necessary to rewind/replay the entire system in order for this to happen. Conversely, the system operator must still be able to apply undo across all system state in order to recover from system-wide failure or to carry out low-level repairs that affect all users. An extension of this problem occurs in a clustered system, where it would be useful from an efficiency standpoint to support undo on the per-node level as well as the cross-cluster level.

Exposing undo at multiple granularities raises some challenges, most significantly in managing and coordi-

nating the timelines of state at different levels of the system. For example, if a user in an email system has rewound his or her own mailbox, and the system operator then wants to rewind the entire system, a policy is needed to determine which rewind request takes precedence, and coordination is necessary to ensure that all state ends up at the correct point in time upon replay.

A further challenge arises when implementation is considered: to support fine-grained undo at the per-user level, system state must be divided into per-user state and shared state and dependencies between the two types must be respected on rewind and replay; similar issues apply to the logs of user actions used for replay.

4 A 3R-Undo Prototype: Design of an Undoable Email System

We are implementing a prototype of our 3R undo model in the context of Internet and enterprise email. We chose email as our first target application as it 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. For example, retroactive repair is useful in an email system when viruses or spammers attack: with an undo system, the operator can rewind the system back to the point before the virus or spam attack began, retroactively install a filter to block the attack, then replay the system back to the present time. Furthermore, the undo abstraction could be propagated up to the user, allowing the user to recover from inadvertent errors such as accidentally deleting messages or folders without involving the sysadmin.

Figure 1 illustrates how our prototype is implemented as a proxy wrapping an existing IMAP/SMTP-based email server. *Further discussion of the prototype has been omitted from this abstract due to space limitations, but will appear in the final version.*

5 Conclusions and Future Work

Traditional approaches to dependability have not eradicated failure and they do not address the problems of operator-induced and operator-compounded failures. To meet the demand for dependable infrastructure systems, we must consider these unavoidable human factors and develop recovery mechanisms that address them. Our system-level undo mechanism does just that: it provides a tool that compensates for the weaknesses of human operators, allows them to erase the effects of their mistakes, and harnesses hindsight to enable retroactive repair, all while preserving the data that users care about.

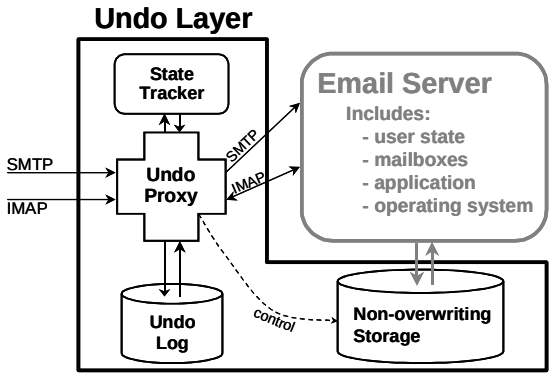

Figure 1: Architecture of Undo layer for email. During normal operation, the proxy snoops traffic destined for the mail server and logs mail delivery and user interactions. The proxy also monitors accesses to messages and folders to track externalized state. Upon an undo request, the non-overwriting storage layer is rolled back to the undo point, the operator is allowed to perform any needed repairs, and then the proxy is used to replay the logged user interactions that were lost during the rollback.

Although we have taken the first steps toward exploring the issues and challenges associated with implementing system-level undo, there is a great deal more to be done, ranging from a further exploration of the issues raised in Section 3 and their solutions, to studying the applicability of the 3R undo model to a broader range of applications, to examining the feasibility of exporting the 3R undo abstraction at a finer granularity to the end-user. We are pursuing these issues, and would welcome company in further developing what we see as an essential mechanism for the dependability of tomorrow's computer systems.

References

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

[2] A. Brown and D. A. Patterson. "To Err is Human." *Proc. 2001 Workshop on Evaluating and Architecting System dependabilitY*, Göteborg, Sweden, July 2001.

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

[4] P. Enriquez. "Failure Analysis of the PSTN." Unpublished talk available at http://roc.cs.berkeley.edu/retreats/spring_02/d1_slides/RocTalk.ppt, January 2002.

[5] D. E. Lowell, S. Chandra, and P. Chen. Exploring Failure Transparency and the Limits of Generic Recovery. *Proc. 4th OSDI*. San Diego, CA, October 2000.

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

[7] D. Oppenheimer and D. A. Patterson. "Architecture, operation, and dependability of large-scale Internet services." Submission to *IEEE Internet Computing*, 2002.

[8] J. Reason. *Human Error*. Cambridge University Press, 1990.