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
{abrown,patterson}@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 of undo that matches the needs of human error recovery and retroactive repair; discuss several of the issues raised by this undo model; and introduce an initial architectural framework for undoable systems using the example of an undoable e-mail 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] [7] [14]. 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 entirely. 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] [15].

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 *external inconsistency*, where the undo process alters or revokes erroneous state changes that have already been seen by external end-users. Another challenge lies in identifying and tracking state that should be made recoverable through undo: a good undo mechanism will preserve user data while allowing arbitrary

[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.

repairs to system state, and drawing the dividing line between recoverable and non-recoverable changes is non-trivial. A final challenge is in constructing an undo system that works at multiple granularities: cluster-wide, per-system, and per-user.

In the remainder of this paper, we expand on these challenges, identifying possible solutions and integrating them into an architectural framework for undo-capable systems. We begin in Section 2 by defining the essence of the undo process: the "Three R's" of rewind, repair, and replay. Section 3 puts the Three R's into the context of related work on undo systems. Section 4 delves deeper into the issues and challenges raised by the Three R's model, and Section 5 introduces an architecture for Three R's-undo that addresses these challenges. Finally, we wrap up with conclusions and future work in Section 6.

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 accidental deletion of important data). 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.

3. Related Work

Our 3R undo model draws on a long heritage of work in temporal recovery and undoable systems. Probably the most ground-breaking work to date on sophisticated undo systems can be found in Edwards's Timewarp system [4], a framework for building collaborative productivity applications that supports a rich and malleable view of time and history. In Timewarp, different users work autonomously on shared state, each generating explicitly-visible histories of actions over that state; users can rewind their state, alter their histories, and replay changes at will. Timewarp also defines a framework for detecting and managing undo-induced inconsistencies based on an analysis of the static relationships between user actions [3].

Much of our 3R undo model is similar to Timewarp. In particular, both systems are based on the command-object idiom [9], where history is tracked in terms of user operations representing instances of generic actions. However, 3R undo is targeted at a much lower level of the system and hence has some key differences. First, 3R undo is designed to undo and replay entire systems—from the OS up—and not just application-level events; this requires that 3R rewind be implemented physically whereas Timewarp rollback is done logically. Note that physical rewind is also necessary in 3R undo to cope with buggy systems where user actions can have potentially-arbitrary effects on state.

More fundamentally, 3R undo makes no assumptions about the structure of repair, allowing arbitrary changes to the system itself during the repair phase. In contrast, Timewarp repairs are limited to inserting or deleting well-known actions from the history.

Finally, 3R undo and Timewarp take different approaches to the problem of undo-induced inconsistencies. Timewarp only manages inconsistencies arising as the result of conflicts between two or more operations in the system's history, meaning that it assumes the behavior of operations is repeatable and well-known. Because of its support for unconstrained repair, 3R undo cannot make that assumption, and hence models inconsistencies as arising from conflicts between individual operations and the system state context in which they are executed. The 3R approach permits much more flexibility in repair but sacri-

fices the ability to perform extensive static analysis of inconsistency; an interesting area of future work is to merge the two approaches to provide static analysis of expected behavior while dynamically handling worst-case behavior.

Besides Timewarp, there are several other influential systems that support time-travel-like undo. Freeman and Gelernter's Lifestreams is a system that explicitly manages a user's state repository as a temporal stream of documents [8]; Rekimoto's Time-Machine Computing is a similar system that merges temporal and spatial representations of history [16]. Unlike 3R undo, these systems only support a linear, unchanging view of history without repair: users can observe the past and even add events to the future timeline, but changes to the past are limited to simple annotations that have no side effects on future state. Roxio's GoBack utility is a commercial product providing similar utility to the users of Windows environments [17]. Several graphical-editing environments define models that support editing of past history (with Kurlander's Chimera being one of the most influential [11]), but unlike 3R undo or Timewarp, none of these systems address management of undo-induced inconsistencies.

Finally, in terms of implementation, many systems offer a subset of the 3R properties but none offers full 3R semantics at the system level. For example, backup/restore or checkpointing schemes [1] [6] [12] 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 [13], but again do not offer the ability to interject repair into the recovery cycle; the standard transaction model does not allow committed transactions to be altered or removed. Some extended transaction models do allow altering or undoing of committed transactions (a good example is Korth et al.'s model of compensating transactions, upon which our compensation approach in Section 4.2 is loosely based [10]), but these models are primarily theoretical and, like Timewarp, are based on operation-operation conflicts rather than state-operation conflicts.

In summary, our 3R undo model offers a unique combination of properties not found together in any existing undo system of which we are aware. It operates at a low level of the system, allowing recovery from problems affecting the operating system and higher software levels, supports unconstrained repair with forward-propagation of changes, and allows detection and management of repair-induced inconsistencies.

4. Challenges in the 3R Undo Model

4.1. Tracking recoverable state for 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 track and 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 Msg-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 record user interactions at the protocol level. Most network service protocols define a set of verbs that the client can use to express desired actions, and most are designed so that state is referenced using logical names divorced from any particular internal state representation. Protocols also have the advantage of being reasonably standard and well-defined. Good examples include SMTP and IMAP for email, JDBC/SQL for databases, and XML/SOAP for the emerging online application frameworks. Tracking interactions at the level of protocol verbs and logical state names automatically provides a record of user intent that is independent of the details of the application itself; in fact, it should be possible 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.

Finally, 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 practical reasons we make the choice to not allow replay of repairs in our undo model; we may explore the possibility in future work. Note that this restriction does have implications for the undo history paradigm: the operator can use undo to back up over arbitrary repairs and changes (and in a simple extension can

immediately undo the undo before making any changes), but once changes are made in the repair phase, only user state will be restored on replay.

4.2. External inconsistency

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. Note that this problem of post-undo external inconsistency arises 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 [6], there is no complete fix for the external inconsistency 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 or generating bounce messages upon a delivery failure.

4.3. 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 coordinating 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.

5. An Architecture for 3R-Undo

To explore and address the challenges laid out above, we have been developing an architecture and implementation of a 3R-Undo layer. Our initial application of undo is in an e-mail service system, chosen as it is an essential service in today's Internet environment, but one of our goals is to make the implementation as generic as possible so that it can be easily adapted to other applications.

5.1. Basic structure

Our architecture begins by defining the basic structure of an undoable system. Following the discussion in Section 4.1, we require a verb-based application interface that uses logical state names; this allows the undo system to properly disambiguate recoverable user state changes from other system events. The verb-based interface should cover all interactions with the system that affect user-visible state, including operator interfaces that are used to create, delete, move, and modify user state repositories. If existing protocols satisfy these requirements, then all is well; otherwise, new protocols can be created and layered above the existing interfaces. In our target case of e-mail, the IMAP and SMTP protocols define most of what is needed in a verb-based interface for the end-user, although they need a small extension to the naming semantics in order to provide globally-unique and time-invariant state names. Also required is a new verb-based protocol to standardize and encapsulate the parts of the operator interface affecting user state repositories (such as accounts or maildrops), which we are attempting to develop.

A consequence of the verb-based interface requirement is that undo functionality is best implemented as a wrapper layer surrounding the application itself, interposing on the interface to track and replay state-changing events. Keeping the undo layer outside of the application itself leaves the most flexibility for repair and minimizes the potential for introducing new bugs into the undo system as the application evolves. As a fringe benefit, it also enables straightforward geographic replication of undo-wrapped services, as the high-level intentional undo log can be shipped from one instance of the service to another and applied using the same replay mechanisms used for an undo cycle. We will assume this wrapper-based architecture, shown diagramatically in Figure 1, for the remainder of this paper.

5.2. Managing external inconsistency

Probably the most difficult challenge in architecting undo is in developing a generic mechanism to detect and manage external inconsistency. Our approach is based on an analysis of the history recorded by the undo wrapper: we augment the log with enough information to allow the undo system to track inter-verb dependencies and identify unsafe operations that result in the externalization of inconsistent state. Once identified, these inconsistencies can be addressed using one of the possible solutions introduced in Section 4.2: applying compensating actions, extending the boundary of undo, or ignoring the inconsistencies entirely. A key property of our approach is that it uses a declarative specification of the inter-verb relationships, allowing the system designer to identify and

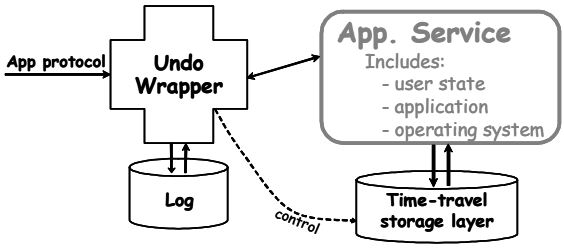

Figure 1: Basic undo architecture. The application service is wrapped with an undo layer that interposes on the verb-based application protocol, logging the intent of user changes for later replay. The undo wrapper also manages a time-travel storage layer used to implement rewind, and tracks state externalization to manage undo-induced inconsistency.

exhaustively test all possible inconsistency conditions; this provides increased confidence in the undo system's dependability, and provides the tools needed to verify *a priori* that all possible undo scenarios can be handled.

The crux of our approach to external inconsistency is an understanding of undo history. We define a *history* as the sequence of *operations* recorded/logged by the undo wrapper, where an operation is an instance of one of the *verbs* in the application interface. In our e-mail application, a verb might be to FETCH a message from the mail store, a corresponding operation would be a particular user's fetch of a particular message at a particular point in time, and the operation would appear alongside other operations in a history of all user interactions with the mail system.

The potential for external inconsistency occurs during the replay stage of undo when an *unsafe* operation appears in the history. Unsafe operations are operations that produce different results during replay than they did during their original execution. To detect unsafe operations, we augment our logging of operations with the following fields (note that these can be specified to the undo system declaratively in template form along with the interface verbs, allowing a generic undo layer to manage the details of operation logging):

- the names of the set of state entities needed to carry out the operation
- a set of preconditions over those state entities that must be satisfied for the verb to be successfully re-executed and to produce acceptably consistent results with the original execution.

The preconditions are generated dynamically as the operation is logged, and incorporate whatever tests are necessary to guarantee acceptable consistency. For example, in our e-mail scenario, the fetch operation might be logged with a precondition that checks a hash of the message body against a hash taken when the original operation

was logged; a discrepancy indicates that the message contents have been altered by repair. Another e-mail example is mail delivery to a folder. The logged operation might include preconditions to check for the existence and read-write status of the mail folder; if during replay the folder was missing or found to be read-only, the operation becomes unsafe.

Careful specification of preconditions is essential if unsafe operations are to be properly identified. If specified too broadly, the undo system will miss unsafe operations and allow inconsistencies to propagate without compensation; if specified too narrowly, there is the risk that perfectly acceptable replay-induced inconsistencies will be flagged as unsafe, hence limiting the transparency of repairs. Specifying the appropriate preconditions is probably the most challenging part of designing an undoable system.

Preconditions and required-state lists give the undo system the information needed to detect unsafe operations during replay, but further mechanism is needed to determine if those unsafe actions produce external inconsistency. Recall that a main purpose of an undo mechanism is to *allow* changes and repairs to the system, since those changes could be converting previously-erroneous results to correct results. Thus unsafe actions are actually desirable, and they only cause the problem of external inconsistency if the state alterations they make are externalized later in the history.

We therefore return to the importance of history: the level of inconsistency is a property of the history, not of individual operations. We define three classes of history with regard to the level of external inconsistency:

- *replay-safe*: the history can be re-executed as is without causing visible external inconsistency
- *replay-acceptable*: re-execution of the history causes visible but acceptable external inconsistency (for example, the history includes compensating actions)
- *replay-unsafe*: the history cannot be re-executed without causing unacceptable external inconsistency.

Without repair, any unedited history recorded by the undo system will be replay-safe. But once we introduce repair, replay-unsafe histories can arise as unsafe operations appear and are externalized later in the history. Furthermore, repairs can consist of alterations to the history itself, with operations deleted, added, or changed; if these changes affect later-externalized state, the history again becomes unsafe.

The undo system must detect when history is replay-unsafe, as these histories need to be converted to at least replay-acceptability for the undo system to work. This is done by tracing dependencies from unsafe operations to externalizing operations, an analysis enabled by again augmenting the undo-logging of operations with the following fields (which also can be generically specified in template form):

- an indication of which of the state entities are expected to be modified by the verb's execution
- an indication of which of the state entities are externalized by the verb's execution
- the amount of time that the results of the verb's execution are delayed before being externalized (to capture scenarios where inconsistency is avoided by delaying notification of asynchronous actions).

Given an unsafe operation that "taints" certain state entities (*i.e.*, where preconditions are violated or where the state is modified by the unsafe operation), the undo system can walk the history forward, propagating the "tainted" status to operations that use it as input, and so on. If the undo system finds an operation that externalizes the tainted state, then the potential external inconsistency represented by the original unsafe operation is realized. Knowing this, the undo system can make a decision of how to manage the inconsistency, for example by rewriting the history to insert compensation around the original unsafe operation.

As a concrete illustration of how the detection and compensation process might work, consider the following scenario, which begins when a message containing a virus is delivered to a user's inbox. Before opening the message contents, the user copies the message into a folder, then moves the copy into a second folder. At this time, the system operator realizes that the system is being attacked by a mail virus, and invokes undo. After rolling time back to before the point where our user's message was delivered, the operator installs a filter that discards virus-laden messages, then invokes replay.

During replay, the undo system first executes the deliver operation, which, because of the filter, now discards the message. When replay reaches the copy operation, its preconditions are violated because the message in question was never delivered, and hence it is an unsafe operation. Furthermore, the copy operation externalizes the existence of the message, so the history is replay-unsafe and a compensating action is needed, perhaps inserting a placeholder message into the user's inbox with an explanation for the missing original message. With this compensation, the original replay-unsafe history has been transformed into replay-acceptability, and thus the now-safe copy operation can be executed (on the placeholder message), as can the subsequent move operation. A variant on this scenario would be a system where the operator is willing to accept greater inconsistency by not considering the copy operation as externalizing the message; in this case, the entire history would be replay-safe since there is

no externalizing operation that depends on the unsafe deliver operation.

5.3. Supporting multiple-granularity undo

So far, we have laid out an undo system architecture that addresses two of the three major challenges introduced in Section 4: tracking recoverable state and managing external inconsistency. While at this point we have not yet developed a general solution to the remaining challenge of multiple-granularity undo, we believe that the architecture developed in the previous sections may offer many of the tools needed to analyze and solve the multiple-granularity problem. In particular, the same mechanisms used for detecting unsafe actions, tracing dependencies, and counteracting inconsistency can be used to manage undo dependencies that cross granules of system state.

For example, consider a system supporting per-user undo and a scenario where user *A* wants to replay an operation that updates state from user *B* (or globally-shared state). The replay system can detect this using the dependency information in the augmented undo log, and, depending on the system policy, can either initiate a cascading undo of user *B*'s state or can treat the operation as unsafe and apply the appropriate compensations to *A*'s state. To support such analyses, the architecture needs only the extensions of divvying up system state into independently-undoable per-user state repositories and of logically splitting the system history into per-user histories; the approach to multi-level undo used by Edwards et al. in the Flatland system [5] may prove useful here, although it will need significant adaptation to remove the assumption that operation side-effects are tracked in the history.

5.4. Implications for applications

As we have defined it, our undo architecture imposes some constraints on the applications that can be wrapped with undo functionality. We can treat these constraints as essentially defining a template for undoable applications; besides helping to identify existing applications/services that can be easily extended with undo, the template provides implementors with a guide to constructing new undoable services.

The template for an undo-wrappable service is characterized by a set of properties that must be met for our architecture to be applicable:

- Clients must access the service through well-defined, narrow verb-based interfaces that identify state with logical names; the undo wrapper imposes on this interface to provide 3R functionality.
- All state in the service must be uniquely named by logical identifiers that are assigned when the state is created and never changed. These unique IDs (UIDs) are used to specify the state accessed, modified, or externalized by tracked operations, and are essential in the analysis of external inconsistency.
- Any state affected by an interface verb must be accessible via the interface; this gives the undo system a mechanism for recording the original value of a piece of state, needed to generate the preconditions used to detect inconsistency during replay.
- The interface must offer a complete set of actions on state; it should be possible to generate the inverse of a given verb either with another verb or a sequence of other verbs. This property makes it possible to build compensating actions.
- The service must support relaxed consistency semantics, since external inconsistency is unavoidable with a repair-based undo. Services that demand perfect external consistency and do not allow compensations provide little flexibility for repair during undo.

While these properties may not capture the full set of important service applications today, in many cases only small changes are required to make popular services undo-wrappable. Our target e-mail service is one such example: it already supports relaxed consistency and uses verb-based interfaces (IMAP and SMTP), and with simple extensions (adding an IMAP command to modify message bodies and defining globally-unique message IDs) would fit our undo-wrapping criteria.

5.5. Status

As of this writing, we have just begun a new implementation of the 3R undo architecture described above; we have already built and discarded a partial throw-away implementation of a 3R-undoable e-mail system whose failings inspired many of the design decisions in the architecture presented herein. One of the most significant lessons learned from that early prototype was that it was difficult to have confidence in the implementation because of the mental gymnastics required to anticipate potential external inconsistency and track the state needed to compensate for it. In response, an important goal of our new implementation is to explore the possibility of producing a generic undo wrapper that implements the logging, external consistency management, and 3R functionality in an application-independent manner; it may still be complex, but it will only have to be developed and debugged once. To achieve this goal, we are investigating ways to separate application policy from generic undo mechanism, perhaps by allowing the application to supply specifications of its

verbs and their properties along with callbacks to evaluate preconditions and invoke compensating actions.

6. Conclusions and Future Directions

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.

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 4 and their solutions, to extending and formalizing the framework introduced in Section 5, 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, to developing a generic implementation of 3R undo as a pluggable framework for services that want undo-based recovery.

There is also great potential for progress in other domains related to undo. One example is problem detection: while psychology shows that humans can quickly self-detect about 70% of problems that they create [15], undo-based recovery would become even more powerful given some mechanism for detecting the remaining 30%. Another example is in techniques to provide virtualized, isolated clones of active systems, allowing the operator to experiment with undo and carry out "what-if" scenarios without affecting the live system. A final area is in benchmarking: new types of *recovery benchmarks* are needed to evaluate the utility of techniques like undo. We are pursuing many of these areas as we develop an implementation of our undo architecture, and would welcome company in further advancing what we see as an essential mechanism for the dependability of tomorrow's computer systems.

Acknowledgements

The ideas in this paper would not have developed without the input of the members of the UC Berkeley/Stanford ROC research group. Special thanks go to Jim Gray for insightful feedback and inspiration for the framework of Section 5, and to the anonymous reviewers for their comments. This work was supported in part by DARPA under contract DABT63-96-C-0056, the NSF under grant CCR-0085899 and infrastructure grant EIA-9802069, and the California State MICRO Program.

References

[1] A. Borg, W. Blau 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] W. K. Edwards. Flexible Conflict Detection and Management in Collaborative Applications. *Proc. 10th ACM Symp. on User Interface Software and Technology*. Banff, Canada, October 1997.

[4] W. K. Edwards and E. D. Mynatt. Timewarp: Techniques for Autonomous Collaboration. *Proc ACM Conf. on Human Factors in Computing Systems*. Atlanta, GA, March 1997.

[5] W. K. Edwards, T. Igarashi, et al. A Temporal Model for Multi-Level Undo and Redo. *Proc 13th ACM Symp. on User Interface Software and Technology*. San Diego, CA, November 2000.

[6] 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.

[7] P. Enriquez, A. Brown, and D. A. Patterson. Lessons from the PSTN for Dependable Computing. *Proc. 2002 Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN)*, New York, June 2001.

[8] E. Freeman and D. Gelernter. Lifestreams: A Storage Model for Personal Data. *ACM SIGMOD Bulletin* 25(1):80–86, March 1996.

[9] E. Gamma, R. Helm, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] H. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. *Proc 16th VLDB Conference*, Brisbane, Australia, 1990.

[11] D. Kurlander and S. Feiner. Editable Graphical Histories. *Proc 1988 IEEE Workshop on Visual Languages*, Pittsburgh, PA, October 1988.

[12] 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.

[13] 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.

[14] D. Oppenheimer and D. A. Patterson. Why do Internet services fail, and what can be done about it? *Proc. 10th ACM SIGOPS European Workshop*. Saint-Emilion, France, September 2002.

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

[16] J. Rekimoto. Time-Machine Computing: A Time-Centric Approach for the Information Environment. *Proc 12th ACM Symp. on User Interface Software and Technology*, 1999.

[17] Roxio, Inc. GoBack3. http://www.roxio.com/en/products/goback/index.jhtml.