Undo for Operators: Building an Undoable E-mail Store

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

Abstract

System operators play a critical role in maintaining server dependability yet lack powerful tools to help them do so. To help address this unfulfilled need, we describe *Operator Undo*, a tool that provides a forgiving operations environment by allowing operators to recover from their own mistakes, from unanticipated software problems, and from intentional or accidental data corruption. Operator Undo starts by intercepting and logging user interactions with a network service before they enter the system, creating a record of user intent. During an undo cycle, all system hard state is physically rewound, allowing the operator to perform arbitrary repairs; after repairs are complete, lost user data is reintegrated into the repaired system by replaying the logged user interactions while tracking and compensating for any resulting externally-visible inconsistencies. We describe the design and implementation of an application-neutral framework for Operator Undo, and detail the process by which we instantiated the framework in the form of an undo-capable e-mail store supporting SMTP mail delivery and IMAP mail retrieval. Our proof-of-concept e-mail implementation imposes only a small performance overhead, and can store days or weeks of recovery log on a single disk.

1 Introduction

Dependability is one of the greatest challenges facing the designers and implementors of today's enterprise and Internet services, yet even as industry and researchers strive to build more dependable hardware and software [11] [22], dependability today is still largely delivered or lost by the human beings who operate and administer service installations. Human operators are entrusted with the power and responsibility to configure service systems and keep them running despite frequent upgrades and unexpected failure, but, like any humans, they are prone to human error and thus can themselves be a significant impediment to dependability [4].

Despite their critical role in maintaining dependability, system operators are confronted with an unforgiving environment offering little support for carrying out that role. Configuration, upgrades, diagnosis, repair, and recovery at each layer of the system are typically performed with an ad-hoc collection of independent tools. Mistakes can have catastrophic consequences, including loss or corruption of user data, and thus there is little ability to explore and experiment with different potential solutions. Furthermore, in today's complex, tightly-coupled, rapidly-changing systems, operators face precisely those dependability problems that are most likely to result in mistakes: unfamiliar situations with complex interactions and underspecified symptoms [24]. It should come as no surprise, then, that human operator error is pegged as the root cause of roughly 20% to 50% of system outages [10] [18] [20].

Consider, as an example, what problems an operator might face in the day-to-day administration of a corporate or ISP e-mail store. She might be asked to add a new virtual host to the system's configuration; what if, upon doing so, she inadvertently alters the configuration so that mail to existing accounts starts to bounce? Maybe she knows what she did wrong and can go fix it, but even if so, e-mail may be lost in the interim. And if the problem is harder to track down, the system could operate improperly for hours or days, much as happened with Microsoft's DNS servers during a widely-publicized 24-hour outage that was ultimately tracked down to an inadvertent operator configuration mistake [14].

Or what if our administrator is asked to set up a spam filter on the e-mail store, and she configures it incorrectly the first time around? Again, mail could be lost for a lengthy period while the problem is tracked down and resolved. Or consider the case where the operator installs a software upgrade/patch only to find that it performs poorly—or worse, corrupts data—when deployed at full scale. Maybe the system could be restored from backup, but what about the intervening data that are then lost?

Now, imagine that our operator has a tool available to her that provides a system-wide version of the Undo functionality that we have all grown accustomed to in our word processors and productivity applications. In each of the above scenarios, she could use Undo to restore the system back in time to a point before things went wrong. She could then make repairs, retry the pro-

cedure that went wrong the first time, and, with an appropriately-designed Undo system, roll the system forward again, replaying all of the e-mail deliveries and user mailbox operations that were lost or handled incorrectly the first time around.

Unfortunately, this notion of undo, so common in today's productivity applications, is virtually unheard of in the administration and operations environment. We are trying to change that through our research. In this paper, we present the design and implementation of a proof-of-concept Undo system for network-delivered service applications. Our first target application for Undo is an e-mail store system that receives mail via SMTP and provides retrieval access via IMAP. We chose e-mail because it is a widely-deployed, increasingly-mission-critical service; studies report that up to 45% of critical business information is stored in e-mail [19], and that loss of e-mail access can result in up to a 35% decrease in worker productivity [21]. However, despite our initial focus on e-mail, much of our undo system is designed to be service-neutral, and should apply directly to other systems providing network-delivered services.

In the remainder of this paper, we first present an overview of our model for Operator Undo in Section 2. In Section 3, we explore a fleshed-out design for a service-neutral undo manager that implements our undo model. Section 4 describes the integration of the generic undo design with the specific application of an e-mail message store. We analyze the feasibility of providing Operator Undo for e-mail in terms of resource and time overhead in Section 5, then wrap up with related work in Section 6 and future work and conclusions in Section 7.

2 The Three R's Model of Operator Undo

An undo facility is the ideal counterpoint to the dependability problems faced by system operators. It provides a forgiving environment by allowing operators to recover from their mistakes, to handle unexpected situations by exploring and experimenting with alternative solutions to problems, and by reducing the stress and cognitive strain that arise when every action may be catastrophic. A further benefit is that an undo system can be used by operators as a recovery mechanism for non-human-instigated problems. Just as the system can be "undone" to remove the effects of an operator error, it can be wound back to cancel out corruption due to software bugs, to reverse unanticipated effects of a patch or upgrade, and perhaps even to remove the damage done by a malicious hacker or virus attack.

In previous work, we outlined a model for Operator Undo that provides these benefits and sketched the beginnings of a design for a service-neutral undo manager [3]. We recap that work here then proceed to flesh it out into a practical design and a real implementation.

Our model for Operator Undo is based on three fundamental steps that we refer to as the "Three R's": **R**ewind, **R**epair, and **R**eplay. In the Rewind step, all system state (OS through application) is physically rolled back in time to a point before any catastrophic damage occurred. In the Repair step, the operator alters the rolled-back system to prevent the problem from reoccurring. Note that repairs are not constrained by our model and can consist of arbitrary changes to the system or to the rewound part of the timeline. Finally, in the Replay step, the repaired system is rolled forward to the present by replaying portions of the previously-rewound timeline in the context of the repaired system.

The essence of Three-R's Undo, and the property that distinguishes it from more traditional approaches like backup/restore, is that it preserves the system timeline: it restores lost updates and incoming data on replay in a manner that retains their intent and not the (possibly incorrect) results of their original processing. In all the scenarios discussed in Section 1, Three-R's Undo would have restored lost incoming mail and user mailbox updates, re-executing them on the repaired system where they could be processed correctly. It is this restorative ability that gives Three-R's Undo its power as a tool for the system operator.

2.1 Three-R's design decisions

There are a few essential design decisions captured in the Three-R's undo model as we have described it. First is the choice to perform Rewind physically and Replay logically. In this approach, "undo" is implemented by the single operation of restoring a previous snapshot of a system's hard state, and "redo" is implemented by re-executing a recorded sequence of user-level operations. Physical rewind provides the greatest flexibility in recovering from problems because the undo system makes no assumptions about the semantics of state or the possible corruptions it might encounter. Alternate rewind schemes involving logical rollback would require such knowledge, and risk the possibility of corrupt state escaping rewind due to bugs or unanticipated failure modes. Furthermore, by rolling back *all* state, we do not need to worry about corrupt state escaping the rewind roll-back and persisting to cause problems during replay.

In contrast, logical replay is mandatory if the undo system is to integrate changes made during repair with the system's original timeline. Whereas physical replay would obliterate any fixes made during repair as it rolled the original, corrupted version of state forward over the repairs, logical replay preserves the intent of user operations without reference to the original cor-

rupted state and while still respecting repairs. While logical replay threatens to increase the undo system's complexity and hence the possibility of dependability-affecting bugs, we construct the undo system so that the code used for replay is exercised as part of normal system operation, thereby flushing out any bugs before the replay code must be relied upon during an emergency.

Another key design decision for the Three-R's is that Repair be as unconstrained as possible to allow the operator full flexibility in designing solutions to repair system problems. Often the most confounding and error-prone problems are the ones that have never been seen before. Constraining the Three-R's undo system to a well known set of actions would render it ineffective in exactly those scenarios where it is needed the most.

Finally, note that these design decisions, particularly the choices of physical rewind and unconstrained repair, reflect a fault model that makes minimal assumptions about the correctness of the undoable application service. While this fault model may limit the ability to formally analyze the undo process, it is the key to practical recovery from problems that have altered the proper operation of the system in unknown ways. These are exactly the classes of problems a system operator is likely to encounter when the system is subject to erroneous operator intervention, software bugs, and faulty patches or upgrades.

2.2 Challenges in the Three-R's model

Given the design decisions we have made, there are two key challenges in the Three-R's model. The first is timeline management: to provide the time-travel-like behavior of the Three-R's, an operator-undo system must record the system's timeline so that it can be edited during Repair and re-executed during Replay. In doing so, the undo system must accurately capture the intent of all state changes made by the system's end users in such a way that they can be later replayed while still respecting the alterations made to the system during Repair. Furthermore, the recorded timeline must be causally consistent with the actual execution of the system: all non-commuting operations must be recorded in the same order they were originally executed.

The second key challenge in the Three-R's model is to keep the system consistent from the point of view of an external observer. As in the time travel paradoxes in popular science fiction stories, Three-R's undo can result in a system that appears inconsistent across time to an external observer. This occurs when alterations made during Repair cause state that had already been seen by the observer before Rewind to take on new values during Replay. For example, a repair that affects an e-mail server's spam filter could cause previously viewed e-mail messages to change or be removed, causing the system to appear inconsistent to the observer of those e-mail messages. Note that inconsistencies in state *not* already seen by an external observer are acceptable—even desirable—as they represent the positive effects of repairs; it is only when the inconsistencies are in previously-viewed state that they must be managed.

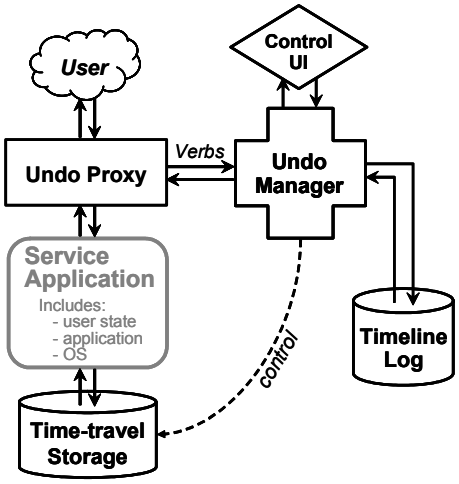

Figure 1: Undo system architecture. The heart of the undo system is the undo manager, which coordinates the system timeline. The proxy and time-travel storage layer wrap the service application, capturing and replaying user requests from above and providing physical rewind from below.

3 Design of a Generic Undo System

While our discussion in this paper focuses on providing Operator Undo for an e-mail environment, a primary goal while developing the architecture and implementation of Operator Undo was to produce a tool that would work with as many enterprise- and Internet-service applications as possible. While some parts of an operator undo implementation are necessarily service-specific (the model of acceptable external consistency, for example), much of the mechanism can be built to be reusable and service-neutral. This is an important consideration for a system targeted at increasing dependability, as any complexity added by the undo system increases the likelihood of dependability problems due to software bugs. If the complicated undo mechanisms are built *once* in a generic manner and then reused as undo is added to each new service, bugs in the undo mechanisms will get flushed out quickly, resulting in a more robust system than if the undo mechanisms were built anew each time.

3.1 Undo system architecture

To this end, our undo system design follows the structure illustrated in Figure 1. The service application—such as an e-mail store server—and its hosting operating system are left virtually unmodified; the undo system interposes itself both above and below the service. This wrapper-based approach supports the fault model dis-

cussed in Section 2.1 by keeping the undo system isolated from problems and changes in the service itself.

Below the service application's operating system, a *time-travel storage* layer provides the ability to physically roll the system's hard state back to a prior point in time. Above the service application, interposed as a proxy between the application and its users, the undo system can intercept the incoming user request stream to record the system timeline and can inject its own requests to effect replay. The proxy and time-travel storage layer are coordinated by the *undo manager*, which maintains a history of user interactions comprising the system timeline. The only interface between the undo manager and the service application itself is a callback used to quiesce the application while taking storage checkpoints or rewinding.

For simplicity, we make a few assumptions about the service application. We assume that it includes internal recovery mechanisms that allow it to reconstruct its internal state from a storage checkpoint, and that it flushes permanent state changes resulting from user interactions to stable storage before responding to the user. These assumptions allow us to coordinate the time-travel storage and timeline log without further hooks into the application; having such hooks would allow us to relax the assumptions at the cost of tighter integration.

The use of a proxy-based approach (rather than an approach where the service and undo manager interact directly) biases our implementation toward services in which users interact with the service via a narrow, well-defined interface, or protocol. Certainly, an e-mail service fits this model, with its use of protocols like IMAP and SMTP. And most Internet services are based on open protocols, while many enterprise services are being developed on middleware that uses XML/SOAP-based protocols for communication. In cases where the user accesses the service via a web front-end and not via a well-defined protocol, the Undo proxy can be inserted at the interface between the web tier and the application/middleware tier.

Despite limiting the range of services that can be easily adapted to support undo, the proxy-based approach has significant benefits. First, for applications with standard protocols like e-mail, a protocol-specific proxy can be developed once and then reused across service implementations, again helping to address the fear that the proxy may introduce extra complexity and hence bugs. Along the same lines, using a protocol-specific proxy rather than integrating undo functionality into the application allows repairs to consist of sweeping changes to the system—such as upgrading or replacing the OS or application—while still allowing replay, as long as the protocols themselves have not changed.

Finally, notice that the service in Figure 1 is depicted as a single monolithic block, with a single entry point for user requests. In this simple version of the undo system design, the entire service is rolled back and forward in time during the Three-R's undo cycle. While this is how we have developed our initial proof-of-concept implementation, we believe the architecture can be extended to support a distributed proxy and clustered service architecture. The extension is straightforward in the case where each service node handles an independent subset of the system's users, but may require the use of more sophisticated techniques from the distributed checkpointing and dependency management domains when shared state is involved.

3.2 Verbs: the undo manager interface

The only service-specific component in the architecture of Figure 1 is the proxy that interposes on the service's user-request stream. Clearly, the proxy itself will be application-specific, as it must understand the protocols it is proxying. But the proxy communicates with the undo manager, a component that itself has no knowledge of the service or its semantics, so it must translate user requests into and out of a form that can be handled generically by the undo manager. At the same time, the undo manager must be able to reason about those translated requests in order to address the challenges of timeline management and external consistency discussed above in Section 2.2.

The answer to this seemingly contradictory set of requirements lies in *verbs*, the fundamental construct used to represent events in the system timeline. A verb is an encapsulation of a user interaction with the system—a record of an event that causes state in the service to be changed or *externalized* (exposed to an external observer). To achieve the separation of application-specific proxy and application-independent undo manager, verbs are transparent to the proxy while semi-opaque to the undo manager: a verb contains all the application-specific information needed to execute or re-execute its corresponding user interaction, but to the undo manager appears as only a generic data type with interfaces exposing just enough information to manage the verb's recording and execution.

To decouple the record of user interactions from the specific behavior of the application service in processing those interactions, verbs record the *intent* of user interactions as expressed at the protocol level, rather than recording the effects of those interactions on state or the contents of state itself. For example, when a user deletes an e-mail message, a verb is created that specifies the deletion intent and a name that uniquely identifies the target message. As part of recording intent, verbs must capture any system context required to spec-

ify the behavior of the verb; for example, converting times specified relative to the present into absolute times. In this sense, the task of defining verbs involves similar processes as the task of defining conformance wrappers for Byzantine replication, as defined by Rodrigues et al [25]. Note that designing verbs to capture intent achieves the critical goal identified in Section 2.1 of allowing the Undo system to tolerate faulty application behavior during normal (non-Undo) operation, and makes it possible to replay verbs in the context of a repaired system.

Verbs are used in the undo system during all phases of operation. During normal operation of the service, the proxy intercepts end-user interactions that change or externalize state, packages them into verbs, and ships them to the undo manager for processing. The undo manager uses the verb interfaces to generate a causally consistent ordering of the verbs it receives, sends the verbs back to the proxy for execution on the service system, and records the sequence of executed verbs in an on disk log. This verb log forms the recorded timeline of the system. During the Repair phase, the timeline may be edited to remove, replace, or add verbs, or may be left unaltered if repairs are done directly to the service itself. During the Replay phase, the undo manager attempts to re-execute the appropriate portion of the timeline by shipping logged verbs back to the proxy for execution on the service system. As it does this, it uses verb interfaces to determine if external inconsistencies are being created, and if so, invokes other verb interfaces to perform application-specific compensation. Note that the same code is used to re-execute the verbs during replay as to execute them during normal operation, helping to ensure that the replay code is bug-free and dependable. The flow of verbs during normal operation and during Replay are illustrated in Figure 2.

3.2.1 Verb interfaces

Section 2.2 introduced two key challenges in building Three-R's-undo: timeline management and external inconsistency management. To address these challenges, verbs define a set of interfaces that provide the undo manager with a window into the application-specific semantics of verb execution, thus exposing enough information to allow the undo manager to carry out its management tasks. These interfaces fall into two groups, discussed in turn below.

Sequencing interfaces. The first set of verb interfaces is used to generate a recorded timeline that is consistent with the actual execution of the system, thereby addressing the challenge of timeline management. It consists of three procedures that all verbs must define: a commutativity test, an independence test, and a preferred-ordering test. All three tests take another verb as an argument; the first tests if the two verbs produce the same results regardless of execution order, the second tests if the verbs can be safely executed in parallel, and the third returns a preferred execution ordering of the two verbs in the case where they do not commute. Note that these tests are similar to those defined by actions in the IceCube optimistic replication system, although in that case they are used for log merging and reordering rather than execution control [23].

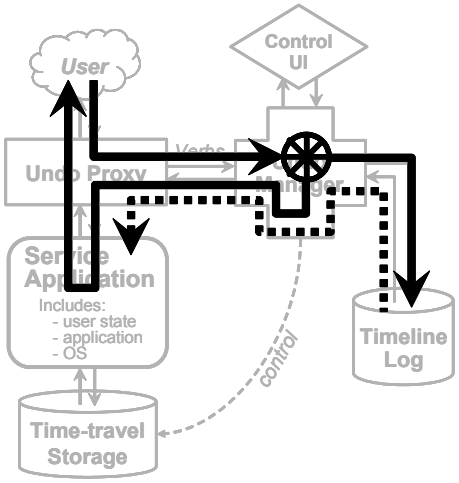

Figure 2: Illustration of verb flow. During normal operation, the verb flow follows the solid black arrows, with verbs created in the proxy and looped through the undo manager for scheduling and logging. During replay, verb flow follows the heavy dashed arrow, with verbs being reconstructed from the timeline log and re-executed via the proxy.

The sequencing tests are used by the undo manager to generate a consistent timeline log when faced with multiple verbs arriving concurrently from multiple users. Because verbs are generated as they arrive at the proxy, whereas their corresponding user interactions are only sequenced for execution in the service application, it is possible that the proxy sees overlapping interactions arrive in a different order than that in which they are eventually executed. Using the sequencing tests, however, the undo manager can guarantee that it sees the same execution ordering as will be chosen by the service application: it can simply stall each incoming verb until all in-flight non-commuting/non-independent verbs have completed execution, using a scoreboard-like data structure to manage the out-of-order execution.

While this approach does involve some serialization of arriving user interactions, it executes as many as possible in parallel, serializing only when there is a non-commutative dependency between concurrently-arriving interactions. It produces a timeline that, when replayed serially, will result in a system state consistent with that produced by the original execution. Furthermore, using the same independence and commutativity

properties, the timeline can be replayed with the same degree of parallelism as the original execution.

Consistency-management interfaces. The second set of verb interfaces is used to manage external inconsistency. This set consists of three procedures that all verbs must define: first, a consistency predicate that compares a record of a verb's original external output to the output produced during replay; second, a compensation function that is invoked with an encoded representation of the inconsistency implied by the failure of the consistency predicate; and finally a squash function used to alter verb execution when it participates in a chain of dependent inconsistent verbs.

The consistency predicate is used to detect externally visible inconsistencies resulting from the undo cycle. Verbs that externalize output record a copy (or hash) of that output when they are originally executed and again during replay. The consistency predicate is applied by the undo manager after the externalizing verb is replayed, and compares the two sets of output to determine if they are acceptably consistent; this test may be simply an equality test, or may be more sophisticated if the application allows relaxed external consistency.

If the consistency predicate fails, the undo manager invokes the second interface, the compensation procedure, which can take whatever application-defined action is necessary to handle the inconsistency. Compensation may consist of ignoring the inconsistency, performing some action to mitigate it (such as creating a missing piece of state), or explaining the inconsistency to the user, among other possibilities.

One final concern involves handling user-induced dependencies between verbs that produce external inconsistencies and later verbs in the timeline. For example, in an e-mail system, a user might choose to delete a message based on reading its content. If during a later undo cycle that (externalized) content is changed, the user's decision to delete the message might be invalid. Given the limited amount of insight into user intent available to the undo system, a conservative approach to handling such scenarios is necessary. The approach we chose is to have the undo manager invoke the third interface, the squash procedure, on all later verbs that do not commute with a verb that externalizes state. Squashing, like compensation, is application-defined, but typically consists of cancelling the verb's original action, informing the user, and leaving it up to them to reconstruct their original intent. Typically, only verbs that destroy or overwrite state will choose to alter their execution when squashed. This policy minimizes the amount of user cleanup needed should a long chain of dependent verbs appear, while ensuring that no potentially-valuable state is lost.

Discussion. The verb interfaces for sequencing and external consistency management bear more than a passing resemblance to similar interfaces used to manage consistency in weakly-connected optimistically-replicated storage systems such as Bayou [30], IceCube [12], and Coda [28]. The similarity is not surprising: the problem of replaying user verbs after the repair phase of Three-R's Undo is somewhat analogous to the task of using an operation log to update an out-of-sync replica in an optimistically-replicated storage system, and our approach is modeled after approaches in that domain.

The key difference in the domain of Three-R's undo is that, unlike in replica systems, not every inconsistency matters—in fact, most inconsistencies that arise are likely due to the positive impact of repairs, representing earlier misbehaviors that are now corrected, and should be silently preserved. This insight motivated the choice of only testing for consistency of external output, rather than using preconditions to test every verb for inconsistency before executing it, as is done in systems like Bayou [30]. We do share Bayou's notion of application-defined compensations; they are just applied in different situations in our system, namely only at the point at which the effects of an inconsistency cross the external boundary of the system.

Another key difference from replica systems is the use of *post-execution* consistency checks in Undo, rather than pre-execution checks. The reason for this is that, in Undo, inconsistencies arise only during replay, not normal operation, and thus can be safely detected after the fact. The built-in rewind functionality can be used to unwind execution to properly compensate for a detected inconsistency, if necessary. Using only post-execution checks simplifies our design, as it is much easier to compare a verb's actual output for consistency than to predict whether its inputs will produce a consistent result, especially given our lack of assumptions about the service's correctness.

3.2.2 Handling failed verbs

Special handling is required when verb execution fails during normal execution. If the verb's corresponding operation reports its status back to the end user synchronously, we do not record the verb as part of the system timeline, and thus the corresponding operation will not be retried upon replay. While this may seem counter to the goals of an Undo system, the problem with recording and later replaying synchronous failed verbs is that the subsequent timeline—the user's choice of future requests—is informed by the failure, and may not make sense if the failure is converted to a success. For example, if the user attempts to create a mail folder with an illegal name, he or she will see the failure and will likely go and try to create another folder using a valid name. If

Verb	Protocol	Changes state?	Externalizes state?	Async?	Description
Deliver	SMTP	✓		✓	Delivers a message to the mail store via SMTP
Append	IMAP	✓			Appends a message to a specific IMAP folder
Fetch	IMAP	✓	✓		Retrieves headers, messages, or flags from a folder
Store	IMAP	✓	✓		Sets flags on a message (*e.g.*, Seen, Deleted)
Copy	IMAP	✓			Copies messages to another IMAP folder
List	IMAP		✓		Lists extant IMAP folders
Status	IMAP				Reports folder status (*e.g.*, message count)
Select	IMAP				Opens an IMAP folder for use by later commands
Expunge	IMAP	✓	✓		Purges all messages with Deleted flag set from a folder
Close	IMAP	✓			Performs a silent expunge then deselects the folder
Create	IMAP	✓			Creates a new IMAP folder or hierarchy
Rename	IMAP	✓			Renames an IMAP folder or hierarchy
Delete	IMAP	✓			Deletes an IMAP folder or hierarchy

Table 1: Verbs defined for undoable e-mail store

during a later undo cycle the system is changed to accept the illegal name, it makes no sense to create the original illegally-named folder, as the user has already reacted to that failure and altered course accordingly.

On the other hand, if the verb that fails during normal operation corresponds to an asynchronous operation, we have a great deal more flexibility. By delaying the reporting of failure to the user, we create a window during which it is possible to invoke undo, fix the problem that caused the verb to be rejected, and then successfully re-execute the verb, without affecting the user's future choice of timeline. This allows us to handle situations such as when an e-mail system is misconfigured to reject e-mail: by delaying bounces or making them tentative, we can provide a window of time during which the configuration can be fixed transparently to the senders of the originally-rejected mail (we discuss this particular scenario further in Section 4.1.2). Of course, once a failed asynchronous verb's results have been reported, we must treat it like a synchronous verb and refuse to replay it. To make this scheme work, we require that verbs identify themselves as synchronous or asynchronous, and, if asynchronous, specify the time window between execution and status visibility.

The other context in which failed verbs become an issue is during replay, when an originally-successful verb fails on re-execution. This case is much simpler than the ones just discussed; we treat the verb's failure status as simply another piece of externalized state, and apply the same mechanisms described in Section 3.2.1 for handling inconsistency in externalized data.

4 Implementing Undo in an E-mail Store

Now we turn to our implementation of the just-described architecture for an e-mail store service, which we define as a service representing a leaf node in the global e-mail network, receiving e-mail for its own local users via SMTP [13] and making it available for reading via IMAP [5]. We focus on what we had to do to adapt the generic architecture for e-mail and the interesting issues we encountered while realizing the implementation. We will not dwell on components whose implementation required just a straightforward translation of the above design into code, like the undo manager itself.

Our implementation, written in Java to leverage its dependability-increasing language features, wraps an unmodified, existing e-mail store. It comprises about 25K lines of Java code, split about evenly between the generic and e-mail-specific components. The e-mail specific part took about two man-months to implement.

4.1 Verbs for e-mail

We defined a set of 13 verbs for our undoable e-mail store that together capture the important state-altering or state-externalizing interactions in the IMAP and SMTP protocols; they are listed in Table 1.

Note that some of the verbs listed, such as Select, do not alter or externalize state but are defined so that they can be properly sequenced by the undo manager as described in Section 3.2.1. Notice also that all of the IMAP verbs are synchronous, as expected given that IMAP is a request-response protocol, whereas the SMTP Deliver verb is asynchronous, reflecting the asynchronous nature of mail transport. Finally, note that while our set of verbs covers only the most commonly-used e-mail functionality for simplicity, it could be extended to encompass some of the more obscure IMAP functionality (such as subscription management) and to capture basic administrative tasks that are performed through interfaces outside of IMAP or SMTP, notably account creation, deletion, and configuration.

Each e-mail verb is implemented as a Java class that implements a common `Verb` interface; the `Verb` interface is defined by the undo manager and declares an API that is a straightforward mapping of the routines described in Section 3.2.1 into Java function declarations. All verbs contain a *tag*, a container data structure that wraps the information needed to execute the verb and to check its external consistency, including a record of whether its execution succeeded or failed. Other than the verb's Java type, the tag is the only part of the verb that is recorded as part of the system's timeline, so it has to be sufficient to reconstruct the verb during replay.

4.1.1 Managing context

One of the challenges in defining verbs for existing protocols like IMAP and SMTP is being able to capture all the necessary context needed to successfully replay the verbs. For SMTP, this is straightforward: we simply capture the parameters passed in to each SMTP command and store them in the corresponding verb's tag.

Applying the same approach to IMAP proved more problematic. In IMAP, operations name state (folders and messages) using names that are only meaningful in a particular system context. In particular, messages are named either by sequence numbers that change any time a message is added to or removed from a folder, or by so-called "unique" IDs that are only unique to a particular instance of a folder and can be unilaterally invalidated at any point by the IMAP server. Similarly, folders are named by hierarchical names that change any time a folder's parent is renamed.

In order to be able to replay IMAP verbs in situations where repairs have changed the system context, we needed to ensure that the verbs specified only absolute names that would still be meaningful after repair. To accomplish this, we defined the notion of an *UndoID*, a time-invariant name independent of system context and capable of being translated into a current IMAP name for verb execution; the proxy is responsible for converting UndoIDs to and from IMAP names based on the current system context.

In the case of e-mail messages, UndoIDs are allocated and inserted into a reserved message header field whenever a message is injected into the mail store (either by an SMTP Deliver verb or an IMAP Append verb). Then, when creating a verb out of any IMAP command that referred to specific messages, we translate the IMAP names into UndoIDs by fetching the UndoID directly from the message headers. To replay a verb, we translate the UndoIDs back to IMAP names by scanning the folder once to retrieve the UndoID-to-IMAP-ID translations, which are then cached for the duration of the Replay cycle.

The case of folders is more difficult, since there is no place to embed the UndoID in the folder name. To solve this problem, we built a module that we call the `UIDFactory`. It maintains a mapping of UndoIDs to names in a persistent BerkeleyDB database, and is synchronized with the undo manager so that names are invalidated and restored appropriately when the system is rolled back and forward in time. The UIDFactory is designed to be general and reusable, treating names as opaque Java objects. In our e-mail system, the UIDFactory maps UndoIDs to IMAP folder names consisting of an ASCII string for the name of the folder plus the UndoID of the folder's parent.

4.1.2 External consistency model

The first step in implementing the inconsistency management architecture of Section 3.2 for e-mail is to define a model of acceptable external consistency. In doing so, we make a distinction between the transport (SMTP) and retrieval (IMAP) phases of e-mail processing. The transport phase allows for a much more relaxed consistency model than the retrieval phase, since even without an undo system, e-mail transport can result in delayed or out-of-order messages. On the retrieval side, consistency has to be stronger, as users are not used to seeing messages or folders change, appear, or disappear from their Inbox without warning. However, even though users are not used to such inconsistencies, we believe that they will accept them if they are sufficiently explained—there is already evidence for this in the numerous mail filters that delete or alter suspected virus- or spam-containing messages, replacing the original message with an explanatory placeholder.

On the retrieval side, we define externalized state to include the output of message fetches (*i.e.*, the text of e-mail messages, including attachments), the output of message list commands (*i.e.*, the standard e-mail headers, including To, From, Subject, and Cc, but not Date), the output of folder list commands (*i.e.*, the currently extant folders in a user's mail store), and the execution status of any state-altering interactive IMAP commands. We declare this state to be inconsistent upon replay if any objects (messages or folders) that were visible originally are missing or altered on replay or if state-altering commands fail. We discount ordering differences and ignore newly-found objects that were not present during original execution, as such discrepancies are typically masked by sorting in the user's e-mail client.

For the most part, we compensate for detected external inconsistencies on the retrieval side by inserting explanatory messages into the user's mailbox, apologizing for the inconsistencies, explaining what they are, and saying why they were necessary. When new folders or messages are being added to the system, we

can be more clever. For example, when the target folder for a message Append verb is found to be missing, we compensate by creating that folder in a special Lost&Found folder in the user's mail store with the same UndoID as the missing folder. By reusing the UndoID, further replayed operations directed at the missing folder go to its newly created surrogate.

From the undo system's perspective, the transport side of e-mail consists only of the SMTP Deliver verb. As SMTP delivery is asynchronous and only reports back to the sender on failure, the one tricky consistency problem is when a formerly-failed message delivery succeeds on replay. If the failure has already been reported to the sender via a standard bounce message, the undo system must not deliver the message during replay, as it does not know what actions the sender took in response to the bounce. However, by delaying the delivery of the final bounce message (typically, by 4 hours), we create a window in which the operator can use Undo to fix mistakes that would cause mail to bounce erroneously. To avoid aggravating users who are used to getting instant feedback on misaddressed e-mails, a failed SMTP Deliver verb sends an informational "bounce" immediately, informing the original sender that the delivery attempt failed but will be retried for the (typically 4-hour) length of the undo window. Note that any inconsistencies in message content, handling, or recipients are not externalized until message retrieval, so they are handled at that point.

All of the consistency checks and compensations described above are implemented through the `Verb` API in conjunction with the verb tag. When a verb is executed, it updates the tag with a record of the externalized state and the verb's execution status. To reduce the amount of data that must be stored in the timeline, most e-mail verbs record only hashes of their output in the verb tag. Verbs define check routines that compare the tags from the original and replay executions, and compensation routines that perform verb-specific compensations such as generating explanatory messages.

4.1.3 Commutativity and independence

As required by the `Verb` interface, all of our e-mail verbs define methods for determining if one verb commutes with or is independent of another. These determinations are made by examining the verb types and the contents of their tags, and can often be made simply. For example, any two SMTP verbs are independent of and commute with one another, since message ordering is not considered in our consistency model for e-mail. Similarly, any two IMAP verbs belonging to different IMAP users are independent and commutative, since each user's mail store is independent of all others. To facilitate this determination, IMAP verbs store their associated username in the verb tag. SMTP and IMAP verbs commute with one another unless the IMAP verb is a Fetch for a user's Inbox; in this case we conservatively mark the verbs as non-commutative since, due to the existence of aliases and mailing lists, it is impossible to determine from the proxy level who is the actual recipient of an arriving SMTP message.

Given these rules, the only remaining case is of two IMAP verbs for the same user. Here, the tests have to be more extensive, examining the input parameters in the two verbs' tags to determine if they commute. For example, an Expunge and a Fetch do not commute if they share the same target folder, nor do a Store and a Copy if they share the same target messages. However, Append and Store do commute if they have different target message UndoIDs, as do Append and Fetch.

4.2 E-mail proxy

The e-mail proxy is responsible for intercepting all SMTP and IMAP traffic directed at the mail server, converting it into the verbs described above, and interacting with the undo manager. The proxy is one of the simpler components of the undo system. It accepts connections on the IMAP and SMTP ports and dispatches threads to handle each incoming connection. Each connection is handled by a separate thread, which runs in a loop, decoding each incoming SMTP or IMAP interaction, packaging it into a verb, and invoking the undo manager to sequence, execute, and record the verb. For IMAP connections, the proxy never interacts directly with the server; it merely opens connections that are used by the verbs themselves during original or replay execution.

The SMTP case is more complicated, however. Because we want to be able to use Undo to recover from configuration errors that cause mail to be erroneously rejected, we must create verbs for all delivered messages even if they would normally be rejected. Thus the SMTP proxy acts more as a server than a proxy, completing a transaction with the client before packaging it into a verb and sending it to the real server. By doing so, however, the proxy opens itself up to denial-of-service attacks: an external attacker can generate streams of invalid SMTP requests (such as relaying requests), cluttering up the timeline log and burning extra resources to handle the later failures when those requests are executed against the real server. In our prototype, we err on the side of caution by validating recipient addresses against the real server before accepting a transaction from the client, rejecting any *syntactically*-invalid recipients or relay requests, while allowing otherwise-rejected recipients. This decision reflects a tradeoff between attack vulnerability and the system's ability to recover from configuration errors affecting address validation, and is probably a decision best left to site policy.

4.3 Time-travel storage layer

At the base of the undoable e-mail system is the time-travel storage layer, which provides stable storage for the e-mail store's hard state as well as the ability to physically restore previous versions of that state. The storage layer is designed to be application-neutral, and has neither knowledge of the e-mail store nor any customizations to e-mail semantics.

Ideally, a time-travel storage layer would provide the ability to restore state backward to any arbitrary point in time; to restore state *forward* in time to cancel the effects of a previous rollback (essential in providing the ability to undo an undo); and to accomplish these time-travel operations instantaneously. Unfortunately, we could find no storage layer offering all of these properties, so we were forced to improvise. We started out with a Network Appliance filer whose WAFL file system and SnapRestore feature provide snapshots that can be created and restored almost instantaneously. Two limitations had to be addressed: its 31-snapshot limit, and the fact that restoring an old snapshot annihilates any later ones, preventing forward time travel.

We began by building a Java wrapper that hides the telnet/console-based command-line interface to the filer's snapshot management tools. The wrapper tracks the filer's active snapshots and provides an API for creating, deleting, restoring, and listing them. Also, during normal operation, the wrapper periodically takes snapshots at multiple configurable granularities (*e.g.*, every 10 minutes, every hour, every day, every week), aging out old ones according to an algorithm that preserves a specified minimum number of snapshots at each granularity while maximizing the number of snapshots in the most recent past. With our default granularities, the 31 available snapshots span up to a month of time, with up to 20 snapshots concentrated in the past day.

To address the lack of forward time-travel, we added a routine to the wrapper that copies an old snapshot forward to the present, effectively overwriting the current state of the system with an older snapshot of state, but without destroying any intervening snapshots. By leveraging the filer's ability to forward-restore a single file from a snapshot in constant time, this copy-forward routine runs in time proportional to the number of files in the file system, independent of their size. Given this ability, we implement reverse time-travel by first taking a recovery snapshot, then copying the desired old snapshot to the present. To do forward time-travel after that, we need merely restore the recovery snapshot, which takes the system back to the point before the old snapshot was made current.

Finally, to address the limited number of snapshots, we designed the undo manager to implement Rewind by first restoring to the nearest snapshot prior to the rewind target, then using the existing Replay code to roll the system forward to the exact target time point. Given this approach, extra snapshots become a performance optimization rather than a functionality issue.

4.4 Undo manager

Our implementation of the undo manager is a reasonably straightforward translation of its description in Section 3 into Java code. The undo manager stores the system timeline as a linear append-only log of verbs. The log is implemented as a BerkeleyDB 'recno'-style database, with each verb assigned a sequential log sequence number (LSN). The LSN is the fundamental internal representation of time to the undo manager, and all "time-travel" operations like rewinding and replaying operate in terms of LSNs, although versions of all external interfaces are provided that take real dates and times rather than LSNs.

The undo manager mediates execution of verbs during normal operation much as described in Section 3.2. One special case bears mention: when verbs arrive for execution during an in-progress undo cycle, the execution process has to proceed differently. The undo manager cannot allow the verb's operation to modify the state of the service system, since the verb is effectively in a different timeline than the system. However, we do not want the undo system to lose delivered e-mail during an undo cycle. Similarly, we want to retain the ability for users to at least inspect their mailbox state, even if it is temporarily inconsistent and immutable (although this is again likely a site policy choice). Our solution is to defer execution of asynchronous verbs (like SMTP deliveries) until the undo cycle completes—being asynchronous, they can tolerate the delay—and to execute synchronous verbs in a read-only mode. If a synchronous verb cannot be executed read-only, the execution fails and ideally reports an explanatory message back to the user. Synchronous verbs executed read-only are still added to the end of the timeline log, as they can externalize state even if they cannot change it.

5 Analysis of Overhead and Performance

With an implementation in place, we set up some simple experiments to gauge the overhead of adding our proof-of-concept Undo implementation to an existing e-mail store and to evaluate its performance. Since Three-R's undo is targeted at reducing human operator stress and improving overall system dependability, a true evaluation of Undo would require a dependability benchmark incorporating human subjects as described in [2]; such a study is beyond the scope of this paper, although we are in the process of performing one as future work.

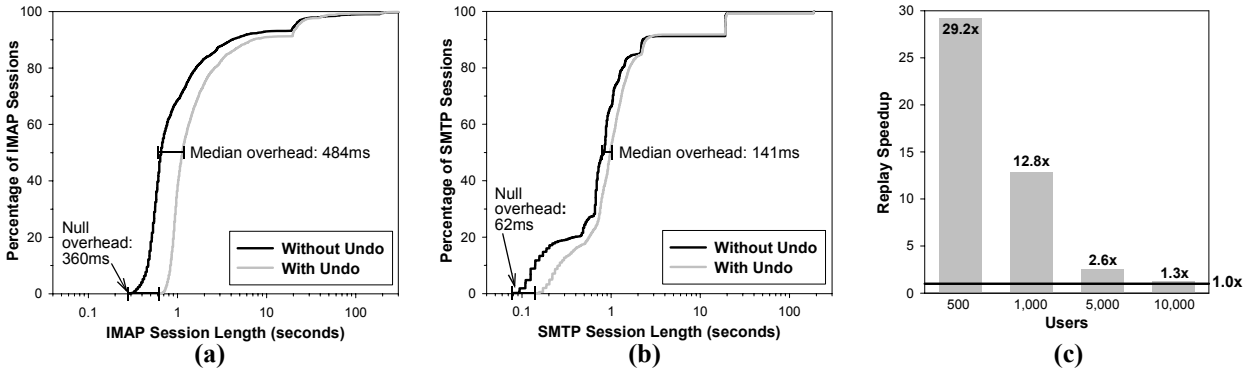

Figure 3: Overhead and Performance of Undo. The leftmost two graphs show cumulative distributions of session length for mail retrieval via IMAP (a) and mail delivery via SMTP (b), with and without the undo system in place. The rightmost graph (c) shows the performance of replay, represented as speedup over the original execution of the benchmark run.

5.1 Setup

We deployed a setup consisting of four machines: a mail store server, the undo proxy, a workload generator, and a time-travel storage server. Details of the machine configurations are given in Table 2. All machines were connected by switched gigabit Ethernet. The filer was configured with two volumes, a 250GB time-travel volume with a 40% snapshot reserve, and a 203GB log volume with the standard 15% reserve. The proxy was configured to store its timeline logs on the filer's log volume, accessed via NFS. The mail server was configured with 10,000 user accounts, with all of their storage (home directories and mailspools) placed on the filer's time-travel storage volume and accessed via NFS.

Our measurement workload was provided by a modified version of the SPECmail2001 e-mail benchmark [29]. SPECmail simulates the workload seen by a typical ISP mail server with a mixture of well-connected and dialup users. We modified the SPECmail benchmark to use IMAP instead of POP for retrieving mail and added code to export detailed timings for each e-mail session along with the benchmark's usual summary statistics; we also modified the benchmark to direct all mail to the mail store rather than to remote users, as we were only interested in mail store behavior. The benchmark was set up with its standard workload for 10,000 users at 100% load, a configuration that is intended to generate a workload equivalent to what a 10,000-user ISP would see during its daily load peak. In our experiments, this translated to an average of 95 SMTP connections and 102 IMAP connections per minute. Each benchmark run consisted of a 30 minute measurement interval preceded by a 3-minute warm-up.

Machine	Configuration
Mail store server	**Type:** IBM Netfinity 5500 M20 **CPU:** 4x500MHz Pentium-III Xeon **DRAM:** 2GB **OS:** Debian 3.0 Linux, 2.4.18SMP kernel **Software:** Sendmail 8.12.3 SMTP server, UW-IMAP 2001.315 IMAP server
Undo proxy	**Type:** Dell OptiPlex GX400 **CPU:** 1x1.3GHz Pentium-IV **DRAM:** 512MB **OS:** Debian 3.0 Linux, 2.4.18SMP kernel **Software:** Sun Java2 SDK version 1.4.0_01
Workload generator	**Type:** IBM Intellistation E Pro **CPU:** 1x667MHz Pentium-III **DRAM:** 256MB **OS:** Windows 2000 SP3 **Software:** Sun Java2 SDK version 1.4.1
Time-travel storage server	**Type:** Network Appliance Filer F760 **DRAM:** 1GB **OS:** Data OnTAP 6.2.1 **Disk:** 14x72GB 10kRPM FC-AL, 1TB total

Table 2: Machine configurations for overhead experiments

5.2 Results: overhead

We begin by comparing the user-visible latency with and without the undo system in place. Figures 3(a) and (b) plot the cumulative distributions of the IMAP and SMTP session lengths measured both from the unmodified e-mail store server and from the undo-enabled version of the same. In the latter case, the undo system was actively proxying connections and recording the system timeline. Note that here, a session is defined as a single complete set of client interactions with the mail server, from login to logout.

Looking at the session-length distributions, we see that the Undo system does not significantly alter the shapes of the distributions, essentially just shifting them to the right. This shift represents the overhead added by the proxying, verb-generation, and verb-scheduling code. While the overhead imposed by Undo is not negligible, ranging from 62ms for a null SMTP session to 484ms for the median IMAP session, it is still relatively small compared to the threshold at which human users begin to perceive unacceptable sluggishness, typically pegged at about one second [16]. Furthermore, this latency is spread across an entire session and not a single interaction. For longer sessions (typically, those that

retrieve large amounts of data), the undo-induced overhead is essentially insignificant.

Next, we consider the storage overhead of providing Undo, which consists of the timeline log and the database of mail folder name mappings (as discussed in Section 4.1.1). We measured the amount of log data accumulated during the measurement interval of the undo-enabled benchmark run described above, which consisted of 30 minutes of peak-load traffic for 10,000 users. In that time, the undo system generated 206.5MB of timeline log. Closer analysis showed that a bug in the Java serialization code was contributing an enormous amount of overhead by writing large swaths of garbage data to the log. With this overhead factored out, the undo system generates an estimated 96.2MB of uncompressed timeline log over the 30 minute interval, 71% of which consists of copies of incoming e-mail. This result extrapolates to 0.45GB of timeline log per 1,000 simulated users per day. Translating to a more concrete reference, a single 120GB disk could hold just under 250,000 user-days of log data, enough to record 3½ weeks of timeline for a 10,000-user ISP. Adding log compression may help further reduce the storage overhead of undo.

The name database size is relatively static and proportional to the number of total mail folders in the system. For our 10,000 users each of whom only had an Inbox, the corresponding name database required 12.3MB of disk space, indicating that a 120GB disk could hold the names for over 93 million e-mail folders.

5.3 Results: performance

We next look at the performance of the Three-R's cycle itself. We measured this by starting with the system at the end-state of a 30-minute SPECmail benchmark run for various numbers of simulated users, and recorded the time it took to rewind the system back to a storage checkpoint taken at the start of the benchmark run, then to replay it forward to the end of the run.

With the forward-time-travel workaround of Section 4.3 in place, it took on average 590 seconds, or 9m50s, to rewind the 10,000-user system (average of three runs, standard deviation <1%). The bulk of this time was spent copying files from the old snapshot into the active system, and so this time is heavily dependent on the number of files in the file system; our experiments show it to scale roughly linearly with the number of simulated SPECmail users. In contrast, using the Network Appliance filer's built-in snapshot restore capabilities, an old snapshot can be (non-undoably) restored in a constant 8 seconds on average (10% std. deviation over 12 runs), independent of the number of simulated users. This is the order of magnitude rewind time that would be achievable in practice, given the proper interfaces into the filer to support undoable snapshot restore.

Turning to replay, Figure 3(c) plots the time to replay the logged verbs from a SPECmail benchmark run represented as a speedup over the original 30-minute run-length, for several different numbers of simulated users. Across all experiments, the system was able to sustain an average replay rate of approximately 8.8 verbs/sec, enough to surpass by a factor of 1.3x the maximum original verb arrival rate of 6.7 verbs/sec for 10,000 simulated users, and by much larger factors for lighter workloads with fewer users. While this replay performance brings the possibility of Operator Undo into the realm of feasibility, it is still somewhat disappointing. Analysis shows that the measured replay performance is primarily due to the overhead of establishing, authenticating, and tearing down SMTP and IMAP connections for each replayed verb. Significant improvements in replay speed could be realized through more optimized connection management, and likewise if these protocols provided a "batch mode" that allowed a trusted entity (like the undo system) to reuse a single authenticated connection to replay the interleaved interactions of multiple users.

6 Related Work

Our Three-R's Undo approach draws on a host of well-studied techniques—service proxying, operation logging and replay, replica consistency management, timeline history management, and checkpoint recovery, to name a few—and creates a novel synthesis of them in the form of a tool for creating a forgiving environment for system operators. In particular, our system uniquely combines the ability to integrate repairs into a logged operation history, common in collaborative productivity application frameworks, with the system-wide applicability of traditional system checkpointing or backup/restore techniques.

It is this ability to restore history after repairs that differentiates our undo system from other log/replay-based recovery environments like transactional database systems [17] and transparent log-based rollback-recovery systems [1] [9] [15]. In these other systems, replay is only possible if the system context has not changed since the log was recorded, for they provide no mechanism to handle replayed events that fail or produce different externally-visible results than during their original execution. In transaction systems, for example, transactions are assumed to be permanent once committed, and cannot change their results or commit status as part of recovery. In log-based rollback recovery, once state escapes to the external world, rollback beyond that escape point is simply disallowed. In contrast, our Three-R's approach allows the trajectory of replay to differ from the original execution, detecting significant differences and compensating for externally-visible

inconsistencies. This detection and compensation is made possible by our verbs: not only do they provide a framework for specifying consistency and compensation, but they provide a higher-level record of user intent than transactions or message logs, making intelligent compensation more feasible.

As we have already discussed in Section 3.2.1, the challenge of retroactively integrating repairs into a logged operation history bears a great deal of similarity to the problem of reconciliation in optimistic replication systems such as Bayou, IceCube, or Coda [12] [28] [30]. But it also has an even more direct counterpart in work on timeline management for collaborative productivity applications, an area which has explored sophisticated undo models supporting highly-malleable views of time. Probably the best example of work in this area is Edwards and Mynatt's Timewarp system [8], a framework for collaborative productivity applications that maintains histories of all user actions over shared state. In Timewarp, users can rewind their state, alter their histories, and replay changes at will. Timewarp defines a framework for detecting and managing inconsistencies that arise from retroactively-inserted changes to the timeline, much as we do to handle inconsistencies resulting from Repair, but Timewarp's approach requires that all alterations to the past timeline consist of insertions or deletions of predefined actions in the operation log, as it identifies inconsistencies by detecting conflicts between the well-known actions [7]. In contrast, our design point of allowing unconstrained repair limits the applicability of the Timewarp approach, and hence we detect inconsistency by directly examining externalized state. Along similar lines, Timewarp performs undo (Rewind) logically, whereas we must perform it physically as we cannot trust that operations were processed correctly during original execution.

Our Three-R's Undo approach supports recovery from system-wide problems, not just errors within an application. Again, this property on its own is available in many other systems. Users of desktop PCs can purchase software tools such as Roxio's GoBack [26] or IBM/XPoint's Rapid Restore [32] that provide the ability to examine past system states, physically roll-back an entire machine, including the OS, to a past state, and even roll-forward again later. Virtual machine systems such as VMware [31] provide the ability to log system operation so it can be rolled-back and replayed; Dunlap et al.'s ReVirt system demonstrates a particularly clever use of the technique for intrusion analysis [6]. But unlike our Operator Undo model, these systems provide either Repair or Replay, but never both—if changes or repairs are made to a rolled-back system, replay either wipes out those changes or is prohibited altogether.

In terms of our actual implementation of an undoable e-mail system, probably the most relevant prior work is a commercial product, the Network Appliance SnapManager for Exchange, which is a system that integrates the snapshot capabilities of Network Appliance filers with Exchange's built-in mail logging [19]; similar functionality is provided by other systems that build e-mail atop a database system. In the case of a system failure, the SnapManager system allows an operator to restore a previously-archived snapshot of the mail system, then replay forward using the Exchange transaction logs. While this system is similar in approach to our Three-R's-undoable e-mail store, there are two fundamental differences. First, the SnapManager system does not detect and compensate for external inconsistencies. Second, operation logs in the SnapManager/Exchange system are recorded deep within the Exchange system, long after the user's protocol interactions have been processed. If the Exchange server is misconfigured or buggy, these logs may be incomplete or corrupted, and will almost certainly not contain a record of mail deliveries incorrectly rejected by the system. In contrast, our external proxy-based logging takes place before the e-mail server even interprets the mail protocols, allowing recovery from failures or configuration problems at all levels of the mail server, and not binding the logs to a specific server implementation (hence allowing server upgrades as part of Repair). While our approach is still susceptible to bugs in the proxy, the likelihood of those bugs is reduced by the relative simplicity of the proxy and the fact that the proxy executes operations directly from the same verb objects as are stored in the timeline log, quickly flushing out bugs during normal operation.

Finally, while we have focused entirely on the mail store and its operator, there has been work on developing undo-like functionality for the user side of e-mail. Here, the challenge is not recovery from system operator error or software bugs, but dealing with user errors like accidental mailbox deletion. We think that many of our Three-R's undo techniques would make sense at this level of the system, and are thinking about ways to extend our approach to provide undo to both operators and end-users, but unlike some brave researchers [27], we do not intend to tackle the most challenging of end-user problems: unsending e-mail on the Internet.

7 Conclusions and Future Work

Dependable systems will not be achieved until we address the challenges facing the human operators who exert such a crucial influence on dependability. Our model of an operator-targeted system-wide undo facility is a first step toward creating a forgiving environment for system operators so that they may better respond to system problems. It reduces the impact of mistakes,

allowing for inevitable human error and making trial-and-error solutions feasible, and provides a last-resort tool for guaranteed recovery from developing software problems or data corruption.

Our proof-of-concept implementation of undo for e-mail proves the concept feasible and demonstrates that even an unoptimized implementation imposes reasonable overheads in terms of space and time. That said, there are several future directions we are pursuing to increase the functionality of Operator Undo and to better understand its influence on dependability. First, we are considering extensions to the basic undo model that would allow for more complex undo timelines supporting multiple branches of history. We are also working to extend the model to support multiple hierarchical levels of undo, allowing undo to be simultaneously exposed at per-user, per-machine, and per-cluster granularities. We would like to see implementations of Three-R's-style undo in more applications than just e-mail; we are considering the design of an undoable auction service, and would welcome company in exploring other applications. Finally, we feel it is important to understand how a tool like Undo affects the behavior of system operators and how those behavioral changes impact dependability; to that end, we are developing dependability benchmarks that incorporate human operators as participants.

Source code

Source code for our undo framework and e-mail proxy is available at http://roc.cs.berkeley.edu/undo/.

Acknowledgements

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. The first author was supported by an IBM Graduate Fellowship. Special thanks go to Network Appliance for their generous donation of the filer used in our experiments, to Mike Howard for always having an experimental testbed or two up his sleeve, and to our shepherd, Brian Noble, and our anonymous reviewers for their feedback and insight.

References

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

[2] A. Brown, L. C. Chung, and D. A. Patterson. Including the Human Factor in Dependability Benchmarks. *Proc. 2002 DSN Workshop on Dependability Benchmarking*. Washington, D.C., June 2002.

[3] A. Brown and D. A. Patterson. Rewind, Repair, Replay: Three R's to Dependability. *Proc. 10th ACM SIGOPS European Workshop*. St. Emilion, France, 2002.

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

[5] M. Crispin. *RFC2060: Internet Message Access Protocol Version 4rev1*. December 1996.

[6] G. Dunlap, S. King, et al. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. *Proc. 5th OSDI*. Boston, MA, December 2002.

[7] 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.

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

[9] 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.

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

[11] IBM. *Autonomic Computing: IBM's Perspective on the State of Information Technology.* http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.

[12] A. Kermarrec, A. Rowstron, et al. The IceCube approach to the reconciliation of divergent replicas. *Proc 20th ACM Symp. on Principles of Distributed Computing (PODC 2001)*. Newport, RI, August 2001.

[13] J. Klensin, ed. *RFC2821: Simple Mail Transfer Protocol*. April 2001.

[14] R. Lemos and M. Farmer. Microsoft fingers technicians for crippling site outages. *ZDNet News*, 25 January 2001.

[15] 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.

[16] R. B. Miller. Response time in man-computer conversational transactions. *Proc. AFIPS Fall Joint Computer Conference*, 33:267–277, 1968.

[17] 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.

[18] B. Murphy and T. Gent. Measuring System and Software Reliability using an Automated Data Collection Process. *Quality and Reliability Engineering International*, 11:341–353, 1995.

[19] Network Appliance. SnapManager Software. http://www.netapp.com/products/filer/snapmanager2000.html.

[20] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? *Proc. 4th USENIX Symp. on Internet Technologies and Systems*. March, 2003.

[21] Osterman Research. Survey on Messaging System Downtime from a user perspective. http://www.ostermanresearch.com/results/surveyresults_dt0801.htm.

[22] D. A. Patterson, A. Brown, et al. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. *UC Berkeley TR UCB//CSD-02-1175*. Berkeley, CA, March 2002.

[23] N. Preguiça, M. Shapiro, and C. Matheson. Efficient semantics-aware reconciliation for optimistic write sharing. *Microsoft TR MSR-TR-2002-52*. May 2002.

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

[25] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using Abstraction to Improve Fault Tolerance. *Proc. 18th SOSP.* Banff, Alberta, Canada, October 2001.

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

[27] A. Rubin, D. Boneh, and K. Fu. Revocation of Unread E-mail in an Untrusted Network. *Proc. 1997 Australasian Conf. on Information Security and Privacy.* Sydney, Australia, July 1997.

[28] M. Satyanarayanan. The Evolution of Coda. *ACM Transactions on Computer Systems*, 20(2):85–124, May 2002.

[29] SPEC. SPECmail2001. http://www.spec.org/osg/mail2001.

[30] D. B. Terry, M. M. Theimer, et al. Managing Update Conflicts in a Bayou, a Weakly Connected Replicated Storage System. *Proc 15th SOSP.* Copper Mountain, CO, Dec. 1995.

[31] VMware. http://www.vmware.com.

[32] Xpoint Technologies. XPoint Rapid Restore Server. http://www.xpointdirect.com/en/IBMRRPC/RRServer.asp.