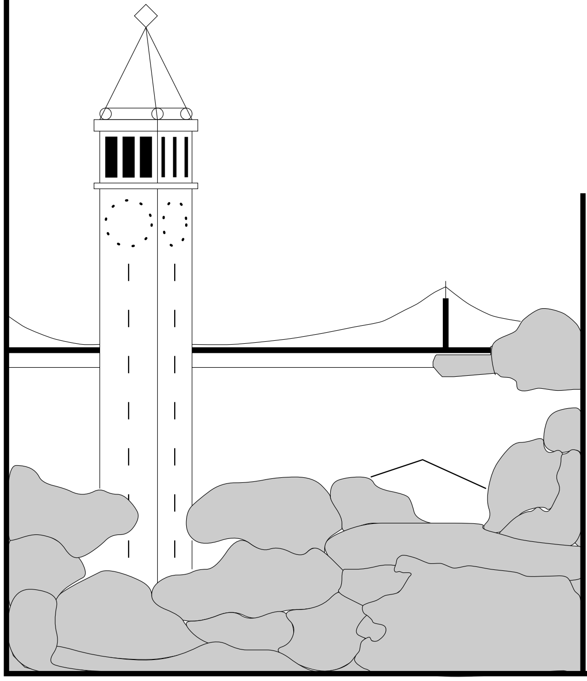
A Recovery-Oriented Approach to Dependable Services: Repairing Past Errors with System-Wide Undo

Aaron Brown
EECS Computer Science Division
University of California, Berkeley



Report No. UCB//CSD-04-1304

December 2003

Computer Science Division (EECS)
University of California
Berkeley, California 94720

A Recovery-Oriented Approach to Dependable Services:
Repairing Past Errors with System-wide Undo

by

Aaron Baeten Brown

A.B. (Harvard University) 1997
M.S. (University of California, Berkeley) 2000

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor David Patterson, Chair
Professor John Chuang
Professor Armando Fox
Professor Katherine Yelick

Fall 2003

A Recovery-Oriented Approach to Dependable Services:
Repairing Past Errors with System-wide Undo

by

Aaron Baeten Brown

Abstract

Motivated by the pressing need for increased dependability in corporate and Internet services and by the perspective that effective recovery can improve dependability as much or more than avoiding failures, we introduce a novel recovery mechanism that gives human system operators the power of system-wide undo. System-wide undo allows operators to roll back erroneous changes to a service's state without losing end-user data or updates, to make retroactive repairs in the historical timeline of the service system, and thereby to quickly recover from catastrophic state corruption, operator error, failed upgrades, and external attacks, even when the root cause of the catastrophe is unknown.

We explore system-wide undo via a framework based on the novel concept of spheres of undo, bubbles of state and time that provide scope to the state recoverable by undo and serve as a structuring tool for implementing undo on standalone services, hierarchically-composed systems, and distributed interacting services. Crucially, spheres of undo allow us to define the concept of paradoxes, inconsistencies that occur when an undo process retroactively alters state that has been exposed outside of its containing sphere of undo. Managing paradoxes is the grand challenge of system-wide undo, and to tackle it we introduce a framework that automatically detects and compensates for paradoxes; our approach exploits the relaxed consistency semantics already present in existing services that interact with human end-users.

We describe an implementation of our system-wide undo framework for standalone services with human end-users. We explore its applicability by assembling and evaluating a prototype undoable e-mail store service, by analyzing what would be necessary to construct an undoable online auction service, and by developing a set of guidelines to help service designers retrofit their services with undo. We find that system-wide undo functionality imposes non-negligible but tolerable overhead in terms of both time and space. Using a novel methodology we develop to benchmark human-assisted recovery processes, we also find that undo-based recovery has a net positive effect on dependability, providing significant improvements in correctness while only slightly degrading availability.

Table of Contents

List of Figures

List of Tables

Acknowledgements

This dissertation owes its existence to the encouragement of my advisor, David Patterson, who gave me the inspiration and confidence both to tackle what others said would be an impossible problem and to carry it through to fruition. My dissertation and qualifying-exam committee members—Armando Fox, James Hamilton, Kathy Yelick, and John Chuang—also deserve my utmost gratitude for their enthusiasm and insights, and for the time and energy they invested into this work.

Equal thanks go to those who provided the moral and technical support that carried this dissertation through to completion: above all, to Randi Thomas for true friendship, constant moral support, and for always being there when there were bumps in the road; to Jim Gray for being an inspiring force, a source of keen insight and sage advice, and an encyclopedia of knowledge about database and transaction systems; to Mike Howard for always finding a rabbit to pull out of his hat whenever I needed more hardware or system support, and for providing a real-life sysadmin perspective on this work; and to Leonard Chung, Billy Kakes, and Calvin Ling for all of their work in helping develop and realize the human-aware benchmarking methodology. Thanks also go to Westley Weimer for an intriguing analysis of the undo code and for his willingness to take on the challenge of being the first outside user of our undo system; to Hua Tang for graciously volunteering statistics advice when it was needed; and to all of the many participants in the user studies and surveys that underpinned our experimental evaluation of undo.

Thanks go as well to the members of the Recovery-Oriented Computing (ROC) Research Groups at U.C. Berkeley and Stanford for their continued interest, advice, feedback, and discussions as the work in this dissertation matured. Finally, a personal thank you to my parents and to all the others who provided encouragement, food, company, and advice, and sympathetic ears over the past six and a half years—you know who you are!

This work was supported in part by an IBM Graduate Fellowship, a Department of Defense National Defence Science and Engineering Graduate Fellowship, and by grants from DARPA under contract DABT63-96-C-0056, the NSF under grant CCR-0085899 and infrastructure grant EIA-9802069, and the California State MICRO Program. Corporate support was provided by Allocity and Microsoft, and Network Appliance graciously provided the filer used in our prototype implementation and experiments. The information presented in this dissertation does not necessarily reflect the position or the policy of the Government or of our corporate sponsors, and no official endorsement should be inferred.

Chapter 1

Introduction

> "To err is human, to forgive divine."
> — *Alexander Pope*

We are in the midst of a sea change in the way that the world's data is stored and maintained. The pendulum that years ago swung toward decentralized, desktop-based storage and manipulation of data has reversed its course, and the computing universe is quickly falling back to a model of centralized processing and data delivery via network services. But we are not simply seeing a revival of the old mainframe model of computing. The pendulum's track is covering new ground, with today's network-delivered services targeting new classes of information users and devices, under new usage models encouraging outsourcing and service composition across corporate boundaries, and on hardware platforms optimized for cost and peak performance at the expense of traditional mainframe virtues like reliability and serviceability. Information services are extending the reach of computation and tying businesses and consumers together through electronic marketplaces, grids, and infomediaries [40] [49]; they are displacing traditional communication technologies [88]; expanding global connectivity by providing back-end support for mobile "post-PC" devices [73] [94]; and even enriching the lives of ordinary home computer users through their popularization of advanced information retrieval, commerce, and media sharing technologies.

In this new service-oriented world, the need for a dependable computing infrastructure has never been more urgent. Services already underpin the viability of many sectors of the economy: studies reveal the impact of service outages as ranging from tens of thousands to millions of dollars of lost revenue per hour of downtime (see Table 1-1) [13] [59] [125], with collateral damage to customer retention and even to the involved service providers' market valuation [9]. Productivity within organizations plummets when communication services are affected: one study reports that e-mail service downtime reduces average worker productivity by up to 35% [88], as nearly half of critical corpo-

Service	Downtime cost
Brokerage	$6,450,000/hr
Credit card authorization	$2,600,000/hr
eBay	$225,000/hr
Amazon.com	$180,000/hr
Package shipping	$150,000/hr
Home shopping	$113,000/hr
Catalog sales	$90,000/hr
Airline reservations	$89,000/hr
Cellular service activation	$41,000/hr
Online network fees	$25,000/hr
ATM service fees	$14,000/hr

Table 1-1: Economic impact of service outages. Economic impact is characterized as lost revenue during periods of service downtime. Data taken from [13] [59] [125].

rate data becomes unavailable during such downtime [82]. And the potentially devastating impact of failure in the networked control services for critical infrastructure systems like national power and water distribution can easily be imagined. Thus, service dependability is critically needed today, and as the trend toward services accelerates even 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.

1.1 Service Dependability and its Influencing Factors

To begin to conceive of how to address the service dependability problem, we must first establish exactly what we mean by "dependability", then begin to explore the factors that influence it. Traditionally, dependability has been defined as a conglomeration of several attributes—availability, reliability, safety, confidentiality, integrity, and maintainability—synthesized into a measure of the system's "ability to deliver service that can justifiably be trusted" [5].

In the service domain, we work with the latter, more intuitive definition of dependability, and declare a system to lack dependability only if it fails to provide an acceptable level of service to a significant fraction of its end-users for a noticeable period of time. The definitions of "acceptable", "significant", and "noticeable" are properties of a given service, since most real-world services accept that perfect dependability is unattainable and thus acknowledge that minor dips in service quality are acceptable and often tolerable or even invisible from the service users' perspective [38]. One key aspect of our definition is that it de-emphasizes pure peak performance—long the sole focus of much computer science systems research[1]—concentrating instead on the more holistic measure of quality of service, which incorporates performance as just one of many factors such as correctness, completeness, accuracy, and consistency.

Another key aspect of our definition is that it speaks of service quality from an end-user perspective, allowing for services that tolerate internal failures and temporary glitches as long as they do not affect the end-user experience. While this definition of dependability therefore includes the influence of such factors as last-mile network quality, for the remainder of this dissertation we will restrict ourselves to dependability as seen by a hypothetical end-user receiving service at the boundary of the service provider's network—that is, we will ignore Internet and last-mile effects, as these can only serve to lower dependability beyond the upper bound of what is seen at the provider edge.

[1] A cursory analysis of the proceedings of three of the top computer systems conferences (SOSP, OSDI, and ASPLOS) reveals that fewer than 10% of the 649 papers published between 1971 and 2002 concern themselves with dependability, focusing instead on performance improvements and novel functionality.

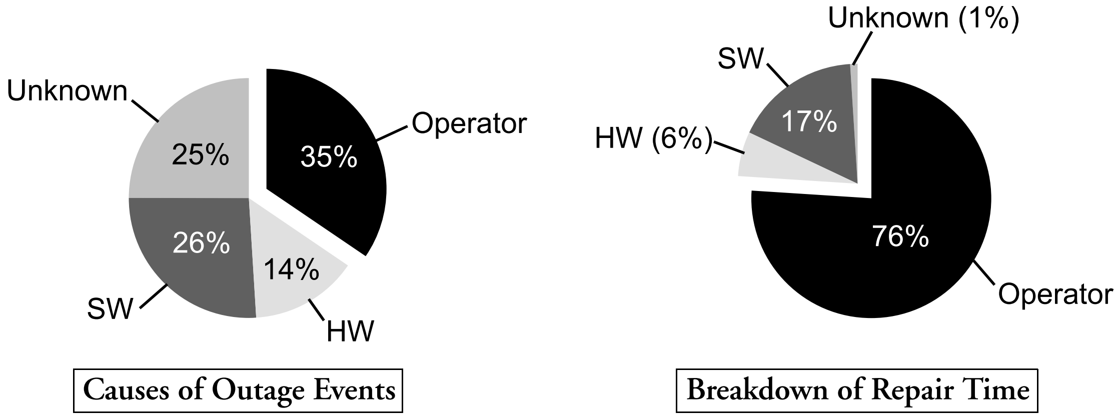


Figure 1-1: Causes of outages and breakdown of repair time for Internet services. The graphs plot data from [85] for two Internet services. The left graph shows a breakdown of outage events by cause. The right graph breaks down the total repair time across all outage events by the cause of the outage that required the repair.

Given our definition of dependability, we can see that any factor that causes a decrease in delivered quality of service is an impediment to dependability. These factors run the gamut from environmental catastrophes to hardware problems to buggy or incorrect software to external attacks to human operator error. Traditional work in fault-tolerant and dependable computing has focused primarily on the first two factors, with increasing work on software and security in recent years. Indeed, this focus has been appropriate, for studies published in the final decades of the last century showed hardware and software to be the key challenges facing computing [45] [46]. But the calculus has shifted as time has passed, hardware has become more reliable, and research on techniques like geographic replication, failover, and fault-tolerant voting has filtered down into practice.

In today's calculus, it is the oft-ignored factor of human operator error that takes center stage as the primary unaddressed impediment to dependability. Two contemporary scientific studies and a raft of older studies and anecdotal statistics highlight its importance. First, consider Enriquez's study of failures in the US public-switched telephone network (PSTN) during 2000 [37]. While not focusing directly on the service domain, the PSTN study is illustrative as an example of how human error can define dependability in a stable system whose hardware and software have been extensively and iteratively refined. Enriquez found that even in the robust and highly fault-tolerant PSTN system, human operator error accounted for 54% of all outages, and 64% of all blocked calls (actual calls that could not be completed due to a system outage). Her study also unearthed the surprising fact that even when human operators did not cause the dependability-affecting problem, they often compounded it through erroneous responses.

More directly relevant is Oppenheimer's 2003 study of Internet service dependability, summarized in Figure 1-1 [85]. Based on an analysis of error logs and failure-tracking databases from several Internet services, Oppenheimer's results indicate that human operator error is the largest single cause of service outages, and furthermore results in outages that take longer to mitigate than those from any other failure source.

Turning to older materials, we find that field data collected over the past several decades bears out the claim that human operators are a key impediment to dependability. Data from the late 1970s reveals that operator error accounted for 50–70% of failures in electronic systems, 20–53% of missile system failures, and 60–70% of aircraft failures [22]. In the mid-1980s, a study of failures in fault-tolerant Tandem systems revealed that 42% were due to system administration errors—again human error [45]. Data collected on the causes of failures in VAX systems reveals that in 1993, human operators were responsible for more than 50% of failures, and that the error rate was rising as hardware and software failures become less frequent [80]. An earlier study of PSTN failures from 1992–1994 reports that human errors resulted in 52% of non-overload outages and were responsible for 50% of non-overload outage minutes, with about half of those due to errors made by telco personnel performing maintenance [65]. And more recently, in 1999 a vice president at Oracle was reported as claiming that one "can have the best technology on the planet, but half the failures we analyze are human error" [77], and in 2001 Hewlett-Packard's High Availability Labs pegged human error as the number one source of unplanned downtime in their systems [110].

1.2 Dealing with Dependability: Avoidance vs. Recovery

Regardless of the source of dependability problems, whether human-related or not, there are two basic approaches to dealing with those problems: **avoidance** and **recovery.**[2] An avoidance approach tries to prevent problems from occurring at all by designing the system so that it is not susceptible to them. Avoidance approaches typically take the form of analysis techniques that identify weaknesses so that they can be corrected before system deployment, or techniques that involve protecting the system with guards so that problems are detected and masked before their effects can be seen externally. Triple-modular redundancy, where the system is replicated three times and a voter used to construct an error-free output, is an illustrative example of the latter technique [112]. Recovery-oriented approaches, in contrast, allow problems to occur (with their results visible externally), and attempt to restore correct operation as soon as possible after the problem has been detected. Deadlock detection and rollback in a transactional database is a good example of a traditional recovery-oriented approach [47].

Both avoidance and recovery are, in theory, equally effective in preserving dependability. One way to see this is to look at the traditional mathematical definition of availability, a restricted measure of dependability constructed as a time average of a binary metric of whether a system is "up" or "down". Availability is defined as the ratio of the mean time to failure (MTTF) to the sum of MTTF and the mean time to repair (MTTR), or:

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

MTTF is a measure of how well the system avoids problems, and MTTR is a measure of how quickly it recovers from them. In this formulation, availability is a fraction between 0 and 1, with 1 representing perfect availability and smaller fractions representing the time-averaged probability of finding the service available at any specific point in time. If we increase the system's avoidance capability by

[2] Our formulation of avoidance and recovery can be seen as a simplification of the traditional breakdown of dependability-improving techniques into fault prevention, fault tolerance, and fault removal [5]. Avoidance covers fault prevention, fault removal, and the masking aspects of fault tolerance; recovery covers the remaining aspects of fault tolerance, including error handling and fault isolation, and further includes external corrective maintenance procedures not usually considered part of fault tolerance.

improving MTTF by a factor *k*, we have the same effect on availability as if we improve MTTR by a factor of 1/*k*:

$$\frac{(k\text{MTTF})}{(k\text{MTTF}) + \text{MTTR}} = \frac{1}{1 + \frac{1}{k}\frac{\text{MTTR}}{\text{MTTF}}} = \frac{\text{MTTF}}{\text{MTTF} + \left(\frac{1}{k}\text{MTTR}\right)}$$

While these formulas show that avoidance and recovery are equally powerful in theory, the realities of practical applications can change the calculus quite significantly. This is particularly true in the service domain, where a combination of rapid service evolution, cost-conscious design, and error-prone human operations tasks make avoidance nearly impossible. Taking these factors one at a time, we first note that services face enormous pressure to keep up-to-date, resulting in the need for rapid evolution in service features and content. This pressure results from aspects of the service model itself. Because many types of services require very little investment on the part of their users—no hardware to buy, software to maintain, or administrators to hire—users of these services can easily switch to different providers of the same or similar service based on the features offered by the other provider. Thus, service providers must keep up in the "feature race", requiring rapid evolution of their services' software platforms. Amazon.com, for example, is reported to roll out software as frequently as three updates per hour [41]. Furthermore, one of the key selling points of the service model is the ability to easily roll out new features at one central point, without having to separately update an installation for each user; this again puts pressure on service providers to evolve their services as rapidly as possible. With rapid evolution comes a corresponding reduction in the amount of effort that can be spent to verify that changes will not cause failure, and well-proven but expensive failure-avoidance techniques like model-based fault tree analysis or systematic fault-injection testing become impractical, leaving recovery as the most effective option. Indeed, at least one major Internet portal service provider explicitly forgoes extensive testing and verification in favor of rapid recovery techniques [18].

All of this rapid service evolution takes place in an environment where cost-conscious design is king. The siren song of commodity hardware is hard for service providers to resist—why buy an expensive fault-tolerant mainframe when the same millions will buy hundreds of commodity 1U rack-mount machines that are easily swapped out and discarded when they break? So services are run on inexpensive machines that do break, and thus hardware failures cannot be avoided and recovery is again essential.

Finally, the human factor provides the most compelling argument for recovery over avoidance. All large services rely on human beings for maintenance and repair. At the very least, humans must perform the physical actions of repairing, replacing, and expanding hardware. Most services require human intervention for software configuration and upgrading, and many require human intervention in the performance tuning loop. The task of diagnosing and fixing failures and other aberrant behavior is also a standard task of the human administrator.

As we all know from personal experience, however, humans make mistakes. Even highly-trained humans working on familiar tasks face non-zero error rates: highly trained, carefully selected, well-paid, and highly scrutinized major league baseball players make errors in 1% to 2% of their fielding chances; bridge watchkeepers, who are supposed to prevent maritime accidents (which could result in the loss of life as well as ships), make serious errors as much as 3.5% of the time [97]; and nuclear power plant operators (who are well-trained and clearly safety-critical) are cited as the most substantial root cause of “significant events” in two surveys of reactor incidents, responsible for 44% to 52% of those events, typically through botched or omitted maintenance activities [54] [55].

Psychology gives us some tools to understand why human mistakes are inevitable. Several researchers have studied the psychological underpinnings of human error [83] [96] [103], but the most succinct synthesis of their work is in Reason’s GEMS framework [97]. GEMS (or Generic Error-Modeling System) describes a model of cognitive task processing and identifies multiple cognitive levels at which humans attack problems. The lowest **skill-based** level is used for common, repetitive tasks. Simple slips and lapses at this level account for roughly 60% of human errors; typos and mistakes in digit recognition are examples of such slips [58]. Up one level is **rule-based** cognitive processing, where reasoning and problem-solving are performed by pattern-matching the situation at hand with previously-seen situations; mistakes caused by misapplying rules account for roughly another 30% of human errors. Finally, the highest cognitive level is **knowledge-based**, where tasks are approached by reasoning from first principles, without the aid of previously-formed rules or skills; mistakes at this level account for the final 10% of human errors. Training and experience move tasks from higher to lower cognitive levels: what might start out as a knowledge-based task gets moved to the rule-based level as it becomes familiar and encoded into mental rules, and similarly tasks may move from rule-based to skill-based if they become so familiar as to be second-nature. While the higher cognitive levels account for fewer errors in absolute terms, they are used less frequently than the lower cognitive levels and thus have higher relative error rates than the lower levels.

The key insight of the GEMS model is that even if all tasks were skill-based (the ultimate effect of long-term training and familiarization), they would still result in human error. And, in service environments where rapid evolution and unexpected failure are common, human operators are more likely to be operating at either the rule- or knowledge-based levels, where relative error rates are higher than at the skill-based level. Furthermore, psychologists have also shown that human error rates can rise to between 10% and 100% in stressful situations, as might be common in service environments, particularly during late-night emergency system maintenance [91].

All of this psychological evidence goes to show that human error is truly unavoidable. Returning to our concern of service dependability, if we were to apply an avoidance-based approach to mitigating the impact of human operators on service dependability, we would have to eliminate the human operators entirely. Completely automating away human operators is an admirable goal, and, if achieved, would indeed represent a giant leap forward in service dependability. But the reality is that the technologies available today for complete automation are not up to snuff; while they may work for small embedded devices, they do not apply to the large, rapidly evolving systems with hard state that make up today’s network service infrastructures.

Furthermore, additional studies from psychology and system accident theory leave little room for debate: practical attempts to automate away human operators in large systems invariably fail due the *ironies of automation* [6] [97]. The ironies of automation expose two problems with automation as an avoidance-based solution to human error. 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.

So we have seen that, in the service context, avoidance and recovery are not equivalent, and that recovery appears to be the most appropriate technique for dealing with service dependability. We have also seen that recovery is most necessary when dealing with human operator error, as avoidance is untenable in that case. Coupled with the earlier observation that human operator error is the most significant current impediment to dependability, we have identified a focus for novel research into service dependability—developing recovery-oriented approaches to human error—and we will take that focus as the subject of this dissertation.

1.3 Undo: A Recovery-Oriented Technique for Human Operators

With a recovery-oriented approach to human operator error as our target, we must devise techniques that will allow service infrastructures to accept and compensate for the inevitable weaknesses of their human operators. Recovery-oriented service 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 productivity applications like word processors and spreadsheets: **undo**. Undo recognizes that humans will make mistakes and offers recovery from those mistakes by providing a way to roll back their effects. It is truly a recovery mechanism, fixing mistakes after the fact, and requiring no advance warning that a mistake is about to happen. It is a method of "designing for error" [84]; it supports the human learning process, allowing trial-and-error reasoning and consequence-free exploration of possible courses of action, identified by Reason and others as critical to effective human-system interaction [97]. And finally, undo matches the way that humans detect their own mistakes: Reason reports that humans can self-detect between 70% and 83% of their own errors (at all cognitive levels), but that they can only self-correct roughly 25%, 50%, or 70% at the knowledge-based, rule-based, and skill-based levels, respectively [97]. Undo is a tool that corrects detected errors, so it dovetails nicely with a human strength (self-detection) while bolstering a human weakness (correction, especially of complex high-cognitive-level mistakes).

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. These problems typically attack **system-level** state, which lives below the application level where undo is likely to be found. System-level state includes operating system and application configuration, the executable binaries for the OS and application software, and software patches, libraries and modules. Compounding the situation is that problems with system-level state can easily percolate up to applications, generating widespread effects beyond the system level; for example, a change to an OS-level packet filter configuration can cause all networked service applications to fail or drop data.

Furthermore, even if we could push it to the system level and resolve the percolation issue, the application-level undo metaphor is not rich enough to solve all the problems we might like it to. Undo at the system level is fundamentally different from the application level because system-level undo potentially affects many more entities—people and computer systems—than just the person invoking it. Also, application-level undo is designed solely to handle human error, but system-level undo can potentially go much farther by allowing the operator to roll back and retroactively repair not only human errors, but also other erroneous state changes resulting from software bugs, faulty patches, hardware-induced corruption, or hacker attacks.

Despite these practical limitations that prevent existing application-level undo systems from solving the problem of human operator error in service contexts, the power of undo as a concept for recovery is extremely compelling. An undo system for service operators could significantly improve both the dependability of service systems as well as making their administrators' and operators' jobs easier. Undo would take the anxiety out of performing system maintenance; it would allow fear-free exploration of new system configurations; it would provide fast recovery from failed or incorrect upgrades, patches, and configuration changes that affect dependability; it would allow retroactive repair of latent problems after they have activated and been discovered, regardless of how long the period of latency; it could be used to clean up from hacker-induced corruption; and it would do all this even if the root cause of the dependability-affecting problem is unknown.

The combination of a compelling need for system-level undo and the concomitant lack of an existing solution for it brings us to three central questions that we will explore in this dissertation:

- What form should an undo facility take for system-wide, human-operator-driven recovery of a service system?
- How and in what service domains can that facility be implemented?
- What are the effects of that undo facility on service dependability?

Through our exploration of these questions, we will show that an undo-based recovery mechanism can be developed for human-operated services, that it is applicable to real-world service applications, that it has the potential to be extended further to capture broad classes of service and non-service systems, and that it can improve service dependability by both attacking the problem of operator error and providing a retroactive repair capability.

Before we proceed, however, it should be noted that undo by itself is neither a perfect nor total solution to service dependability. Undo solves a piece of the problem, perhaps the most important piece today, but there are other significant challenges remaining in the service dependability space. Notably, undo does nothing to address the challenge of *detecting* problems—undo joins the recovery process at the point where a problem has already been detected and the need for recovery has been established. This is not unreasonable given that we saw above how good humans are at self-detecting problems, but a facility to detect anomalies, diagnose them, and suggest when and where to apply undo would be an ideal adjunct to the undo facility we will be describing. Some promising research work in this area is ongoing in the Pinpoint project [20].

Furthermore, undo as we have constructed it only applies to problems affecting *state*. Our undo facility does not provide recovery from problems like accidentally turning off a machine, or restarting after a clean fail-stop failure. In some sense, these non-state problems are the easy ones—no data has been destroyed so recovery is simply a matter of performing a restart or reinitialization—but they are nevertheless problems that could benefit from some recovery-oriented infrastructure support. Candea and Fox's Recursive Micro-Reboots project is taking promising steps toward addressing these non-state problems [18].

1.4 Alternatives to Undo

Nearly all existing techniques for handling human error are, in some sense, forms of undo; we will survey and classify these techniques in Chapter 2 as part of the process of developing our own undo model. Two remaining categories of techniques cannot easily be classified as undo, however. The first is training: preventing human error by improving operators' mental models to the point where no situation is unexpected and error-prone. This is an approach typically taken by safety-critical industries like nuclear power production. While training is certainly effective in reducing human error rates, it is an avoidance technique and cannot completely eradicate error. Indeed, in the words of Reason:

> ". . . errors are an intrinsic part of mental functioning and cannot be eliminated by training, no matter how effective or extensive the programme may be. It is now widely held among human reliability specialists that the most productive strategy for dealing with active errors is to focus on controlling their consequences rather than upon striving for their elimination." ([97], p246)

Training is also difficult on rapidly-evolving information service systems, as the system model may change before operators have had the time to be fully trained on it. Thus training, while a useful complement to undo, cannot replace it.

The other class of techniques targeting human operator error in information services is replication-based fault tolerance, such as in Castro *et al.*'s Byzantine fault tolerance system [19]. Byzantine fault tolerance systems claim to be able to recover from any arbitrary type of failure, human-induced or otherwise, by replicating the operation of the service and comparing the behavior of the replicas to detect and squash misbehaving replicas. While on paper Byzantine fault tolerance may seem to solve all problems in dependability, including human error recovery, it relies on independence assumptions that are rarely met in practice. In particular, human errors must affect only a limited subset of the replicas in the system for their erroneous effects to be detected and squashed by the remaining unaffected replicas. Such an assumption prevents recovery from human errors during common system-wide operations like software upgrades, application deployment, or system configuration changes. These operations typically must be made across all nodes of the system simultaneously, bypassing the protection of Byzantine recovery; if performed node-by-node, the changes would be perceived as misbehavior by unaltered nodes and removed by the Byzantine fault-tolerance engine. Finally, Byzantine replication-based fault tolerance offers no ability to retroactively repair latent or overlooked problems, another key capability of undo-based recovery.

1.5 Contributions and Roadmap

Besides the initial insight of adapting undo as a recovery-oriented mechanism for improving service dependability, the key contributions of this dissertation are as follows, broken down by area:

Concepts and formalism:

- identification of an undo model that meets the recovery needs of human-operated service system, and positioning of that model in the space of existing undo approaches (Chapter 2);
- the concept of a **sphere of undo**, used to delineate the boundaries of an undoable system, to define the interactions between an undoable system and the external world, and to compose undoable systems together (Chapters 2 and 9);
- a solution to the **paradox problem** that arises for external users when undo changes state inside a sphere of undo, based on replaying user interactions and leveraging the ability of humans to understand and tolerate inconsistency (Chapters 3 and 4);
- the notion of **verbs**, a framework for recording and replaying end-user actions on a service in an annotated form that captures the intent and external consequences of those actions, encodes the application's consistency policy with respect to the consequences of the recorded action, and specifies what to do should that consistency policy be violated (Chapters 3 and 4).

Architecture, implementation, and evaluation:

- an architecture for wrapping verb-based undo around existing services, without rewriting them (Chapter 4);
- a practical Java implementation and evaluation of the verb-based undo architecture, and a demonstration of its use for providing undo in an e-mail service (Chapters 5 and 6);
- a study of the issues involved in how the verb-based undo architecture could be used to provide undo in an online auction service (Chapter 7);
- a methodology for evaluating human-centric recovery tools based on **recovery benchmarks** that synthesize aspects of traditional systems benchmarking and human studies, and an application of that methodology to demonstrate the dependability benefits of undo (Chapter 10).

Design guidelines for future undoable services:

- a set of guidelines for developing new service applications with undo in mind, and a corresponding set of criteria for determining whether an existing service application can be made undoable (Chapter 8);
- an approach to transplanting our undo facility from the service environment to the desktop environment, providing new and more powerful forms of recovery for desktop users (Chapter 9).

After discussing each of these contributions in depth, we wrap up in Chapter 11 by discussing directions for future research into undo-based recovery and dependable human-operated service systems, then conclude in Chapter 12.

Chapter 2

An Undo Model for Human Operators

> "O God! O God! that it were possible
> To undo things done; to call back yesterday!
> That Time could turn up his swift sandy glass,
> To untell the days, and to redeem these hours."
> — *Thomas Heywood*

In the previous chapter, we posed the question: "What form should an undo facility take for system-wide recovery of a server/service system, driven by human operators?" In this chapter, we explore the space of possible undo models in search of an answer to that question, and find it by identifying the most practical design point for the demands of system-wide, human-operator-driven server recovery.

2.1 A Survey of Existing Models of Undo

Undo is not a new concept in computer science. Primitive undo facilities were in existence as far back as 1948, in the guise of a checkpoint/restore mechanism on the ENIAC computer [68]. Subsequent research work into undo and models for undo has been spread across several different fields. The bulk of the work on human-driven undo comes from the field of human-computer interaction (HCI), where the focus has primarily been on *interactive productivity applications* such as editors for text and graphics (including CAD systems), word processors, spreadsheets, and graphics design packages. But undo has also had a long and storied history in other fields, such as databases, distributed fault-tolerant system design, and programming language systems and debuggers, where it is used more as an internal recovery tool than a facility for correcting human error. In this section, we will summarize the key contributions of the undo work across these different fields in order to provide context for the undo model that we have developed for human-operated service systems.

2.1.1 Undo models for productivity applications

All of the HCI work on undo for productivity applications presupposes that the undoable application is built to follow the *command object design pattern*, in which users modify a piece of state (*e.g.*, a document) through a well-defined set of commands that act upon it [42]. For example, in a word processor, the document consists of text (words, sentences, and paragraphs), and the end-user's edits consist of adding, altering, deleting, or formatting that text. Under this model, the application maintains a **history** of commands that have been applied to the document. It provides undo by manipulating that history, for example by scanning the history in reverse order and executing the inverse of each command encountered during the scan [10]. In this framework, an undo model is defined by the combination of commands, history, and the undo operator that works to manipulate the history.

The undo models defined in the literature primarily concern themselves with the structure of the recorded history and what undo operators are available to manipulate it. Three key properties distinguish the basic undo models: the representation of the undo operation, the ability to perform selective undo, and the linearity of history. The first property specifies the representation of the undo operation itself, as either a **primitive command** or a **meta-command** [68] [136]. In a system with a primitive-command undo model, invoking undo causes an undo command to be added to the history alongside the operation(s) it is undoing. A subsequent application of undo will "undo the undo" and restore the original command. The undo operation is thus said to be **self-applicable** [44]. There is no explicit redo operation in a primitive-command undo model. Note that some papers refer to this kind of system as having a **history undo/undo model** [131] [136], although that terminology is somewhat of a misnomer as nearly all undo models involve history in some form. The emacs editor is a good modern example of an editor with a primitive undo command [119].

The alternative to a primitive undo command is to have a meta-command-based undo model. In such a model, undo is an operation that manipulates the system's representation of history, but never explicitly appears in it. Systems with an undo meta-command typically also have an explicit redo meta-command. One of the most influential meta-command-based undo models is the Script model introduced by Archer, Conway, and Schneider in the context of the COPE interactive program editor [4], described also in Thimbleby's treatment of undo [127]. The Script model perceives user interaction with the system in the form of a script, divided into two portions: one portion contains the commands that have been executed by the system, and the remaining portion describes commands that are yet to be executed. In the Script model, undo and redo are implemented by manipulating the script, for example by truncating operations off the end of the executed portion of the script for an undo-only model, or, in a more powerful model, by shifting operations between the executed and non-executed parts of the script, allowing them to be undone and redone.

The Script model also highlights the second property of undo models, the ability to perform **selective undo**. An undo model supporting selective undo allows a user to insert, delete, or modify an existing operation in the system's history even when that operation is not the one that has been most recently performed [10]. The Script model accomplishes this by allowing the user to edit the non-executed portion of the script: the user can use undo to transfer operations from the executed to non-executed portions of the script, edit the non-executed portion, then invoke redo to transfer the modified operation sequence back to the executed portion.

A related approach is Yang's Triadic model, which defines an additional meta-command, **rotate**, in addition to undo and redo [136]. A Triadic system maintains the redo portion of the script in a rotatable ring buffer, allowing the user to select the next command to redo by rotating the buffer as desired, for example to skip an existing command or alter execution order. Another approach is evident in Vitter's US&R model, which augments the undo and redo meta-commands with a new **skip** command [131]. The skip command allows the user to skip over recorded commands while redoing a sequence of previously-undone commands. Both the Triadic and US&R models also allow users to insert new commands between undoing and redoing existing commands. Finally, Berlage investigates providing selective undo by copying old commands forward in the history, either in their original form (for redo) or as their inverses (for undo) [10], and Kurlander examines the user interface issues in presenting an editable history to the user in a graphical form that exposes the consequences of selective undo [66].

Of these selective undo models, the US&R model is interesting because it illustrates the third property of undo models, the linearity of their history representation. US&R is a **non-linear undo model**, as it maintains multiple possible versions of the system's history as a tree-like structure, with multiple possible branches at each point in the history. Such a model gives the user selective undo, while preserving the original history and all alternate versions should the user want to go back to them later. In contrast, the Script model is a **linear undo model**, because it only maintains a single, linear view of the system's history at any given time. In the Script model, if the user modifies the history via selective undo, the original history is lost, whereas in the US&R model the original history is retained in a parallel branch to the modified portion of the history.

A generalization of non-linear undo models appears in a body of work examining the challenge of providing undo to collaborative productivity applications, where multiple users interact concurrently to edit the same document or piece of shared state [1] [21] [92]. The key question here is whether an individual user's invocation of undo acts **locally** (on the user's last action, regardless of what others have done) or **globally** (a user's invocation of undo undoes the most recent operation performed by any user) [1]. Local undo provides the better end-user model and is thus more desirable [1], but it requires that one user be able to unilaterally undo his or her own changes (commands) while preserving a sensible view of the shared document to other users. This is a generalization of the selective undo problem with some additional challenges introduced by the fact that the system's global history is distributed and may be difficult to materialize in a consistent form across all the systems participating in the multi-user collaboration.

The problem becomes more difficult with **autonomous collaboration**, where users are operating independently on shared state rather than concurrently. Systems like Timewarp [34] handle autonomous collaboration by enriching the undo model with a notion of **multiple histories**: each user's commands form a private history, and the document's history becomes a compilation of those histories, rather than having a (logically) shared history manipulated by all users. When a user wants to unilaterally alter his or her history, the system must synthesize a new shared document history that reflects the user's history alterations, preserves the histories of other users, and identifies and compensates for semantic conflicts that may arise from the history alterations. For example, in Timewarp-based space planning application, if one user decides to alter history by moving a desk from the center of a room to the corner, that may conflict with a later as-yet-unseen operation from another user who has also put a table in the same corner. In cooperation with the application, Timewarp will flag this conflict and resolve it according to an application-specified policy [32]. Thus, in systems like Timewarp, the basic command/history undo model is enriched with the notions of multiple histories, unilateral undo of a single history, and semantic reconciliation of history conflicts. This same sort of multi-history undo is seen in multi-user source code version control systems systems like CVS [11].

2.1.2 Undo models in other domains

While not traditionally considered to be "undo systems", there are many existing systems that provide undo-like functionality along the lines of the command/history-based undo models discussed above or via their own unique undo models. One area where undo-like functionality is pervasive is the field of database management systems (DBMSs). While DBMSs typically do not provide an undo facility in a form intended to be used by a human user or operator, the notion of undo/redo recovery is heavily embedded in their internal algorithms and procedures [47]. Databases use undo/redo recovery at many levels: individual transactions can abort if they encounter deadlocks or unexpected input conditions, undoing their changes as part of the abort cycle; DBMSs recover from system failure by rolling back the database state to a safe checkpoint (undoing all committed and uncommitted transactions past that safe point) and subsequently redoing the effects of all committed transactions [79]; and DBMSs can even use undo and redo-type operations to manage updates within low-level structures like disk pages [79]. In all these cases, the DBMSs use a command/history-based undo model along the lines of the linear undo/redo meta-command model described in the previous section, with the commands consisting of internal low-level operations like transaction invocations, reads, writes, and lock acquisitions made by transactions, individual page I/Os, and so forth.

Some database systems offer undo models extending beyond their internal recovery mechanisms. Oracle's Flashback Query system uses sophisticated log and history structures to provide a kind of undo for their human operators: it combines checkpoints and shadow-page-based state logging to provide multiple read-only views of a past state of the system plus the ability to roll the system back to a prior state [86]. This is a restricted variant of linear undo with transactions as the primary commands in the history.

Popping up a level, researchers are also applying database recovery techniques to the database application layer, allowing applications themselves to recover from system crashes via redo recovery. For example, Barga, Lomet *et al.* describe a system that checkpoints application state and logs application execution sequences so that if the system crashes, the interrupted computation can be undone to a safe point (by restoring the checkpoint) and redone up to the point where the system crashed (by replaying the log) [7] [8] [70]. This approach is again a variant on linear undo/redo, with the system failure implicitly performing the undo operation, and an automatically-invoked redo operation after the system recovers.

A similar kind of undo recovery is found in the arena of fault-tolerant system design, with undo/redo in a command/history model used to recover from system failures. One instance of this form of recovery is in the design of fault-tolerant processors like those in the IBM ES/9000 and zSeries (G4, G5, G6) mainframes, where if an error is found during execution of an instruction, the instruction-in-progress is undone and then retried, either on the same processor or a different one [114] [116] [117]. A similar technique is used in Sorin *et al.*'s SafetyNet system, which extends the same rollback/re-execute recovery approach to longer sequences of instructions across nodes in a shared-memory multiprocessor [115].

At a higher level, in message-passing systems, fault tolerance for individual nodes can be achieved by a combination of distributed checkpointing and message logging, where upon a failure all nodes roll back their state to a consistent checkpoint and then redo their execution using the logged messages [36]. In both the CPU-level and message-level cases, we again see a simple history/command-based linear undo/redo model, with the commands being CPU instructions and logged messages, respectively.

Lowell *et al.* have generalized this fault-tolerance approach to generic recovery situations, exploring the limits of undo/redo recovery for fault-tolerant applications and extending the undo model to include the potential of altered execution during replay [72]. While similar to the selective undo models discussed above in terms of its consequences, Lowell's model produces altered execution not through explicit alteration of the history, but as a result of non-determinism in the system's re-execution of logged commands. This inclusion of non-determinism during redo differentiates Lowell's model from most of the undo models we have discussed up to this point, which have all assumed that re-execution during redo is deterministic.

Another area that has explored undo-like behavior is programming languages, where undo can be used to reverse the effects of a program's computation. Early work in this area was done in the Interlisp system (as cited by [68]). The Icon language supports a backtracking operation that allows assignment statements to be reversed, providing a kind of history/command based undo in terms of the values of state variables [48]. Leeman defines a formal model for undo in programming languages based on both the meta-command and primitive command models of undo in a history/command framework, with commands consisting of state variable updates reverted by undo and re-established by redo [68]. The undo models in this domain closely reflect those discussed above for productivity applications, although the target uses (such as recovery from errors and debugging via reverse execution) are quite different.

A niche of the user interface community has been exploring ways to build temporally-centered user interfaces, that make time travel a key part of the user experience. A good example is Rekimoto's Time-Machine Computing, where users can ask their system to "travel in time" to examine past snapshots of their desktop, application, and file system state [98]. Similarly, Freeman and Gelernter's Lifestreams system defines a temporal model of data storage and organization, with files and other data organized into temporal streams that extend into the past and the future [39]. While both Time-Machine Computing and Lifestreams provide access to past states, those past states are essentially read-only, and hence neither system truly provides an undo model.

Finally, there are a few undo-like systems that fall into in our primary domain of concern for this thesis, namely whole-system undo facilities for human system administrators, although as we will see over the course of the next few sections that none provide the type of undo model necessary to fully satisfy the unique demands of that domain. One such system is Pehlivan and Holyer's undo facility for command shells, which claims to offer the ability to undo the execution of arbitrary programs executed under the control of a command shell modified to intercept all file I/O operations [90]. A system operator could use this facility to undo errors resulting from mistyped or incorrect command line invocations. The system works by maintaining versions of files altered by program execution, and performs undo by restoring the original versions of those files. It follows the history/command undo model, treating program executions as commands, and offers a selective undo model with a linear history. It does not, however, consider the undo of application behavior beyond what is written to the file system (such as network I/O or responses to users), nor does it include practical solutions to rerouting all file I/O through the shell or to undoing commands that involve more than one process.

The other prime examples of undo at the level of system administration are systems that use backups or checkpoints to provide desktop users with the ability to roll back their entire system state to a version from a previous time. These include tools such as Roxio's GoBack [102] and IBM/XPoint's Rapid Restore [135], which allow a desktop user to revert his or her disk to a checkpoint of its state as of a previous time; traditional disk-level backup tools like `dump` and `restore` for Unix or Microsoft's `NTBackup` for Windows; Windows XP's built-in System Restore tool, which provides system-level undo by reverting key system files to versions stored in a checkpoint [51]; and VMware's virtual machines with undoable disks, which journal all updates to stable storage and provide the ability to undo recent changes by discarding the journal [132].

These systems define an undo model that is not based on recording a history of commands, but rather on directly tracking a set of system states. They provide undo as a coarse-grained bulk operation that simply restores a prior snapshot of the system's critical state. This approach limits the semantics of the undo model, supporting only a linear history and limiting undo/redo operations to discrete points in time where checkpoints or backups were taken, but is the only feasible approach for situations where it is impossible to define a well-specified set of commands that alter state, as in the case of the desktop environment.

While these system-wide undo tools appear close to what we might want for an operator-undo facility for servers, their undo models have a fatal flaw: they discard all work performed by the system's users when they restore a prior version of system state. While this may be acceptable for a desktop system where the user is likely the person invoking the undo operation, it is unacceptable for a server system where the end-users are unlikely to be aware of the operator's actions, including his or her invocations of operator-undo.

Dunlap *et al.*'s ReVirt system relaxes this limitation slightly, extending the undo model to allow incremental redo of the system's original execution trajectory [30]. ReVirt is designed for postmortem analysis of security attacks: it leverages virtual machine technology to roll a system back following an attack, then uses a logged history of machine execution to step the system forward, redoing the original execution history instruction-by-instruction. A side effect of this redo is that any lost user work is restored, but, because ReVirt is limited to replaying exactly the original execution, it does not provide a way to repair the damage caused by the attack. ReVirt thus is still lacking as an undo model for service operators, as it does not allow the operator to selectively undo problem-causing changes to system state while still preserving end-user work.

2.2 Establishing a Broader Design Space for Undo Models

In order to develop an undo model that works for system-wide recovery of a server or service, we need to take a broader perspective on what constitutes an undo model—in other words, not constraining ourselves to the command/history paradigm that has traditionally underpinned undo models in the research literature. We do this by defining an undoable system as any system that is fundamentally concerned with *state* and *time*, and that allows time to flow logically in reverse, such that past versions of state can be restored or recreated. The undo model for such a system describes how state and time are represented in the system's timeline, what operations can be performed to manipulate that timeline, and, if multiple users are involved in the system, what perceptions each of them has of changes to the system's timeline. By taking a broad definition of what constitutes an undoable system, we can both capture traditional undo models (like command/history and the bulk-state/history model used by the system backup/restore tools discussed above), and define a design space of undo models that will help us locate the optimal design point for our desired system-wide undo facility for services.

2.2.1 Spheres of Undo

To provide some structure to our discussion, we introduce the concept of a **sphere of undo** (**SoU**). Spheres of undo are inspired by Davies and Bjork's Spheres of Control, which were developed to provide a framework for analyzing recovery in database and workflow systems [12] [25] [26]. Spheres of control are dynamic constructs used to represent the boundaries of uncommitted information flow within a multiprocess computation; they are created along with processes, and any information that crosses the boundary of a sphere of control has to be either committed or explicitly revokable. Spheres of control can dynamically grow to encompass other spheres as information flows between them, and can be generated after-the-fact during recovery if a committed sphere is later found to be in error. Spheres of control are used to bound the scope of recovery from a failed or incorrect process by identifying exactly those dependent processes that need to be recovered as a result of the original failure.

In contrast, we define spheres of undo as static constructs induced by the structure of a system. Each sphere of undo is a container representing the boundaries of a particular undoable system. The scope of a sphere of undo is defined by state, and the sphere circumscribes all state in a system that is potentially affected by the system's timeline-manipulation operators (such as undo and redo). Spheres of undo can be thought of as bubbles of state and time: each SoU has its own independent timeline relative to the universe outside its boundary, and due to undo operations, it may logically exist at a different point in time relative to that outside universe.

Some examples of SoU's will be illustrative. Consider a typical word-processing application like Microsoft Word, and for simplicity assume that it only operates on one document at a time. In this case, the SoU is defined by the in-memory copy of the document being edited, as that document comprises the state that can be affected by undo. There is an implicit timeline associated with the document and its history of editing commands: as a user invokes undo to cancel out erroneous actions, the logical time inside the SoU is rolled back along with the document state. Should the user decide to revoke the undo operations by performing a redo, the logical time is rolled forward. Observers outside of the SoU see only the unchanging, saved copy of the document on disk, as they are isolated from the document's timeline by the SoU's boundary. When the user decides to save the current version of the document, a SoU boundary crossing occurs: the current contents of the document are written to disk, becoming visible to the outside observer, and implicitly synchronizing the timeline inside the SoU with that of the outside world. Figure 2-1a illustrates this example SoU.

As another SoU example, consider a message-passing system like those assumed in studies of fault-tolerance based on distributed checkpointing, with all of the processes executing a coordinated distributed checkpointing algorithm that provides output commit upon externally-visible output [36]. Figure 2-1b illustrates how in this case, the SoU is defined by the state that makes up the distributed checkpoint, as that is the state that can be restored should a roll-back (undo) be needed. As before, there is an implicit timeline associated with the sphere (and thus the collection of message-passing processes) that advances as checkpoints are taken. If a faulty process is detected and roll-back recovery begins, then that internal timeline becomes out-of-sync with the timeline of the external world. External observers are protected from the timeline discrepancies by the SoU boundary, unless a message flows across that boundary, perhaps in response to an observer's request for data. Once that message

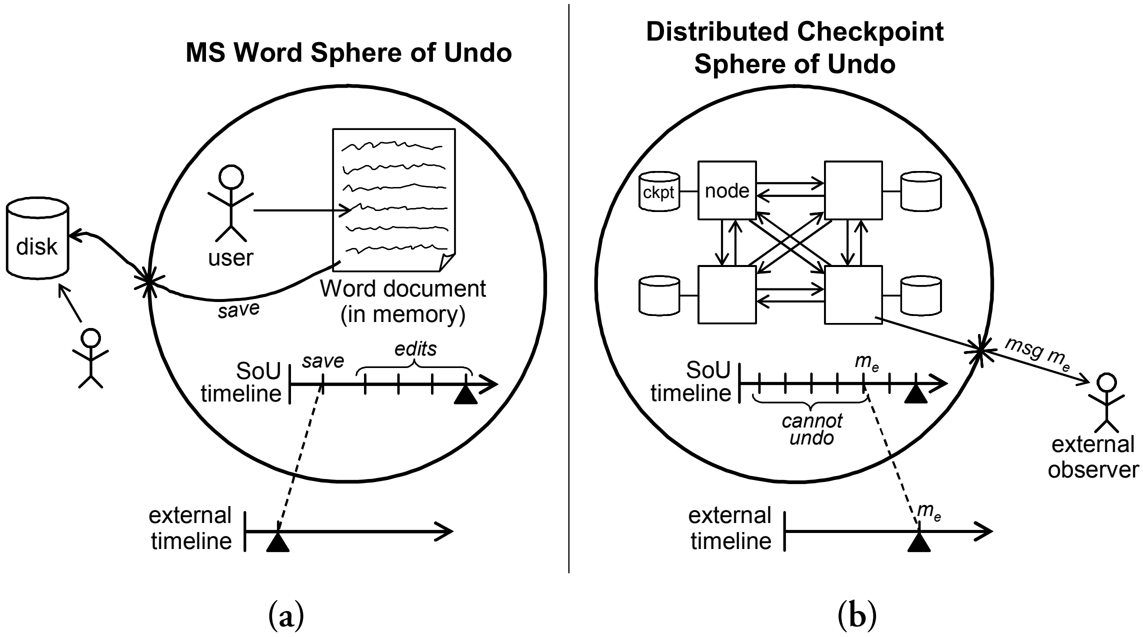


Figure 2-1: Examples of Spheres of Undo. The left-hand example shows a typical sphere of undo for a productivity application, in this case Microsoft Word. The sphere contains the in-memory state of the active document and protects its edits from the view of external observers. When the document is saved to disk, the edits are made visible outside of the sphere, synchronizing the external and internal timelines. The right-hand example shows a typical SoU for a distributed checkpoint system. Here, the sphere protects the checkpoint state and process state from external view. When the system sends a message to an external observer, the timelines inside and outside the sphere are synchronized, and all state changes before that message-send are made permanent and non-undoable (so-called "output commit").

crosses the boundary, the external and internal timelines become synchronized, and, by the output commit guarantee of the distributed checkpointing algorithm, the processes inside the SoU cannot undo (roll-back) past that synchronization point without potentially confusing the observer.

The parallels between spheres of undo and spheres of control may not be immediately obvious, but the key insight is that in both cases the spheres provides *boundaries*. Both spheres of control and spheres of undo represent boundaries of information flow: for spheres of control, the boundaries lie between two different domains of database concurrency- and commitment-control, while for spheres of undo they lie between two different time domains (and likewise, domains of undo control). The information-flow boundary described by a sphere of undo is critical to our understanding of how undo operates in the context of its enclosing environment, since any information that crosses between the interior of the sphere and its outside environment is potentially crossing between two different time domains, and likewise is crossing into or out of the control of the undo system. Being able to draw the correct sphere of undo is therefore a necessary first step in understanding the consequences of adding undo to an existing system.

Just as spheres of control provided a way to bound and understand the propagation of recovery from failed or incorrect database processes, spheres of undo will provide a way to understand propagation of undo-based recovery between time and state domains (such as between a service and its human users, or between independent services that share information). We will return to that discussion later in this chapter, and in even greater depth later in Chapter 9, when we consider undo in the context of

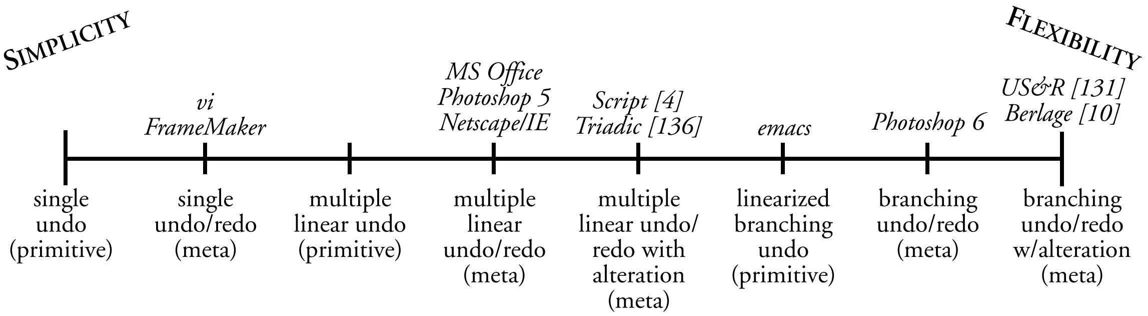

SIMPLICITY
FLEXIBILITY
vi
FrameMaker
MS Office
Photoshop 5
Netscape/IE
Script [4]
Triadic [136]
emacs
Photoshop 6
US&R [131]
Berlage [10]
single undo (primitive)
single undo/redo (meta)
multiple linear undo (primitive)
multiple linear undo/redo (meta)
multiple linear undo/ redo with alteration (meta)
linearized branching undo (primitive)
branching undo/redo (meta)
branching undo/redo w/alteration (meta)

Figure 2-2: Temporal design axis for undo model. The models range from the simplest (single undo) to the most flexible (branching undo/redo with alteration). Points on the axis can be described in terms of the form of the undo operator (primitive/undo-only or meta/undo-redo); whether history is linear, linearized, or branching; and whether the undo model supports alteration, or selective undo. Example systems for the common design points are listed along the top of the axis.

more complex service environments. But first we must understand the design space of undo models within a single monolithic sphere.

2.2.2 Design axes for undo models within a sphere of undo

If we consider just the portion of the overall system that is contained within a single sphere of undo, there are two fundamental design axes that characterize that sphere's undo model. Not surprisingly, these correspond to the two core concepts of any undoable system: time and state. The first axis characterizes the structure of the timeline maintained by the undoable system, that is, the temporal structure of its history. The second axis characterizes how state changes are represented in that history.

Temporal axis. A system's position on the temporal axis captures the richness and flexibility of the history model presented to the users of its undo facility. This axis in our design space covers the entire spectrum of existing undo models described above in Section 2.1.1.

Most undoable systems found in the real world inhabit a point on the temporal design axis somewhere between these two extremes. Figure 2-2 shows some of the intermediate design points along with some example systems. The intermediate points can be described in terms of the same characteristics identified in Section 2.1.1: whether the history model is based on primitive or meta undo commands, whether the history is linear or branching, and whether selective undo is allowed.

At one endpoint of the temporal axis is the simplest possible history model: a system that provides access to a single snapshot of past state, with no ability to return to the present once that past history is restored. In the familiar command/history paradigm, this point on the axis typically manifests as an undo command that can only undo the immediately-prior operation, with no opportunity for redo. In Gordon's formalization of command-based undo, this corresponds to an undo operator with neither the invertibility nor self-applicability properties [44].

On the other extreme end of the temporal axis is a system that offers selective undo and a nonlinear history model, providing access to every point in history previously reached during the operation of the system, plus the ability to delete or alter past events in the history, as in Vitter's US&R model, described earlier [131]. Such as system can be thought of as producing a branching history or timeline: each time the user invokes undo to roll back history, the system creates a new branch of the

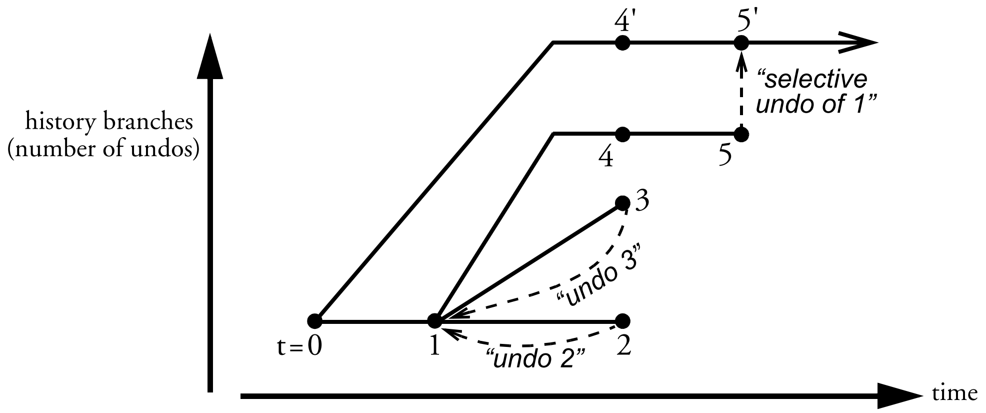


Figure 2-3: Example of non-linear branching history. Time flows from left to right in this example graph. The nodes represent entries in the history—such as commands—and are numbered in the order they are executed. After command #2, undo was invoked to return to the state after command #1. Subsequent command #3 caused a new branch to form. After #3 was then undone, a new branch formed when command #4 was executed. Finally, after executing #5, selective undo was invoked to selectively undo command #1, resulting in the final, current timeline at the top of the graph.

history to hold subsequent operations, and offers the user the ability to jump time to any point along any branch, including abandoned ones. The system would also allow users to clone a branch at any time and alter its contents, then jump to a point along it to make it current. Figure 2-3 depicts a sample history for such a system, with each node representing a command and the node numbers representing the times at which those commands were performed (relative to the monotonic clock of the external world).

State axis. An undoable system keeps a history in order to track changes to state so that they may be undone later; the system's position on the state design axis characterizes how those state changes are represented in the history.

The state axis spans the design space between logical and physical representations of state changes. Most familiar undoable systems sit at the logical end of the state axis and represent state changes in the history as logical commands or actions. These commands typically encode the operations that affect state, rather than the effects of those operations on the state itself or copies of the affected state. For example, a text editor supporting undo might record the entry of new text as a command specifying the new text and its location in the document (and not the current contents of the document), and might record a formatting change as a command specifying the new format and the location in the document where it is to be applied [1].

Systems using purely command-oriented history representations carry out undo by invoking an inverse operation for each command recorded in the history or by transforming the original command to effectively have the inverse effects [99] [122] [123]. They perform redo by re-executing the original logged commands. This approach has several implications. First, every recorded command must have an inverse that exactly reverses the effects of the original command. Second, the approach makes the critical assumption that the software implementing each command (and its inverses) is *bug-free*, that is, that the original execution of each recorded command was correct and performed exactly the state updates specified by the command. As we will see later, this correctness assumption will limit the usefulness of the command-oriented design point for our system-wide undo for operators.

The other end of the axis is occupied by systems that record their history as a series of snapshots of system state—in other words, they record state changes physically, by storing full copies of the system state at each point in the history or by recording the physical changes ("diffs") to state since the last history record. In the latter case, the difference between these systems and the command-oriented systems discussed in the previous paragraph is that the physical systems simply record the state changes with no understanding of the operation(s) that induced the changes, while the command-oriented systems record only the operations, not the state changes that result. Examples of systems at this physical end of the state axis include backup-restore tools, checkpointing tools (like GoBack [102] or VMware [132]), and versioning file systems (like Elephant [108], Cedar [50], and VMS [27]).

Systems using physical state-oriented history representations perform undo by restoring an earlier recorded snapshot of system state (either directly or by inverting the state diffs). They carry out redo by restoring a later recorded snapshot or by reapplying the recorded state diffs. Unlike the command-oriented approach, this approach does not make the assumption that all software is bug-free: because state-based undo simply restores state without regard to how that state was created or altered, it can undo the effects of operations that behaved incorrectly due to bugs, software/hardware failure, or malicious attack. On the other hand, state-based redo is limited, as it can only restore exactly the versions of state that are recorded in the history. Unlike the command-oriented approach, there is no ability to carry out an "altered" redo, as in selective undo, in cases where the history of commands is altered before redo is invoked, or when the unaltered command history is re-executed on a repaired or altered software platform. As we will see later, this restricted redo will limit the usefulness of the state-oriented design point for our system-wide undo for operators.

Since command- and state-based history approaches offer complimentary advantages and disadvantages, a natural approach for some systems is to adopt a hybrid approach, positioning themselves at an intermediate point on the state design axis. Hybrid systems record history as a combination of physical state snapshots and logical commands, using each for different purposes. The original work on the Script undo model proposed a hybrid system as an option for implementing Script-based undo, with undo implemented by restoring the system to a physical checkpoint of its initial state and redo implemented by executing logged logical commands [4]. Similarly, most relational database systems that offer undo-redo recovery as defined in Mohan *et al.*'s seminal paper [79] use a combination of operation logs and physical checkpoints to represent history. The physical checkpoints are used for undo, negating the need to have inverse operations for all updates and expanding the class of failures that can be recovered, whereas the operation logs are used for redo, allowing recovery even if the original execution suffered state corruption in later portions of its history.

Edwards *et al.* also propose a hybrid approach in a somewhat different context—executing pluggable user interface code in an extensible digital whiteboard system—as a way to address the problem of commands with unknown side effects or side effects that might change between the original generation of that command and the time it is undone/redone [33]. However, they quickly reject the hybrid approach on the grounds that it would make redo too expensive, and instead favor a pure command-based approach that requires all side effects to be encoded as new commands in the history. This approach, while sufficient for a user interface extension environment, is inappropriate for our undo facility for service operators, as it again presumes bug-free commands with error-free specifications, and restricts side effects to well-known, loggable commands.

		REDO	
		command-based	**state-based**
UNDO	**command-based**	pure command • *most productivity apps* • *editors*	hybrid • *not useful*
	state-based	hybrid • *relational DBMSs* • *SnapManager for Exchange*	pure state • *backup/restore* • *snapshot FS's* • *CVS* • *versioning FS's* • *GoBack*

Figure 2-4: Matrix defining positions on the state design axis for undo models. Along the state axis, an undo model is characterized by whether it uses a command- or state-based approach for each of undo and redo. Each box in the chart lists examples of systems using the corresponding state model.

Other examples of hybrid systems include the fault-tolerant message-passing system described above in Section 2.2.1, which uses physical checkpoints for undo and operation (message) logs for redo for much the same reasons as the database; and the Network Appliance SnapManager for Exchange, which, in the context of the Microsoft Exchange e-mail system, again combines physical snapshots with a logical operation log to support recovery from a broad class of problems including state corruption and software bugs [82].

If we step back from these examples, we see that there is essentially a two-by-two matrix that defines the extent of the state design axis within a sphere of undo, as Figure 2-4 illustrates. One dimension of the matrix specifies the type of history representation used for undo: command-based or state-based. The other dimension of the matrix specifies the type of history representation used for redo, again command-based or state-based. Three of the four matrix entries are commonly seen in practice, as detailed above; the combination of command-based undo and state-based redo is the odd one out, as it has the disadvantages of both styles with the advantages of neither.

2.2.3 Considering external actors: a third design axis

So far we have only considered the design space of undo models within a sphere of undo, and have neglected consideration of entities outside that sphere. This omission is reasonable for a large class of undo-enabled applications, such as the productivity applications we use day-to-day, because these applications tend to interact with only one external actor—the user that is operating the application and is therefore the one both generating the history and utilizing the undo facilities. Because this end user is aware of the undo process, he or she can easily be seen to exist logically *inside* the sphere of undo (SoU), along with the application's own state. That is, when the user invokes undo and dissociates the timeline inside the SoU from the external timeline, the user is doing so consciously and can adjust his or her perception of the SoU's timeline accordingly; the user will not be surprised that the system state has been rolled back.

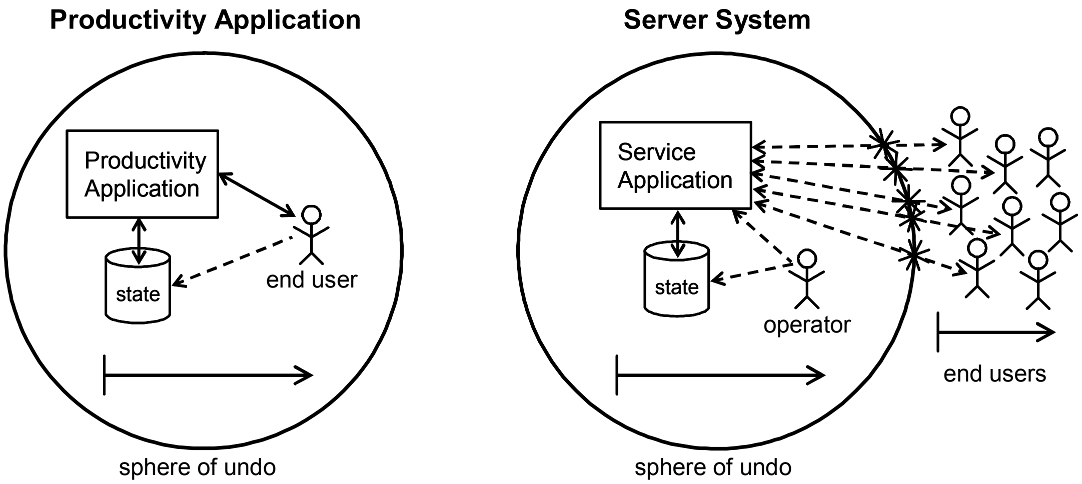


Figure 2-5: Spheres of Undo for productivity applications and server systems. In server systems, the undo model is complicated by the presence of end users external to the sphere of undo. While the administrator or operator can be considered to exist inside the sphere, much as the end user is in the productivity-application sphere, end users in the server case have their own timelines, independent of the timeline inside the sphere, and must therefore be kept outside of the sphere.

Now consider the case that is of interest to us, as established in Chapter 1, of a server system providing a service to multiple end-users, with an undo system for that server's administrator or operator. Such a system can be represented as a sphere of undo containing the server's hard state: the contents of its disk storage, including operating system, application binaries, configuration files, and user data. The administrator can be thought of as logically inside that sphere of undo, just as the end-user of a productivity application is logically inside the SoU of that productivity application. But the end users of the service cannot be assimilated into the SoU of the server, as they are unlikely to be aware of the operator's use of the system-wide undo facility, and thus will not be aware of any timeline divergence within the SoU resulting from an operator undo. Figure 2-5 illustrates this server scenario and compares it to the simpler productivity-application scenario discussed above. Note that we cannot pull the server's end-users into the server SoU simply by notifying them of undo, as they may hold state on the system yet not be actively connected to it when the undo facility is invoked. Such a situation might occur in an auction service, for example, if a user starts up a new auction then disconnects from the service until the auction has completed.

In cases like the server scenario just discussed, invoking undo within the sphere of undo can cause inconsistencies for external actors (like the end users of a server), since the external actors maintain their own view of the system's timeline and will not be aware of any divergence resulting from undo. How the undo system handles these inconsistencies defines a third axis in our design space of undo models: **consideration of external actors**.

An undo model can take one of three positions on this axis. First, it can simply ignore external actors. This is the position taken by most productivity application undo models, since productivity applications usually do not have any external actors—the end user is also the actor who invokes undo. Second, the undo model can attempt to prevent inconsistencies from occurring by blocking any undo operations that would cause inconsistencies to occur. This is the approach we saw earlier in the discussion of the message-passing system in Section 2.2.1: to prevent external actors from ever seeing incon-

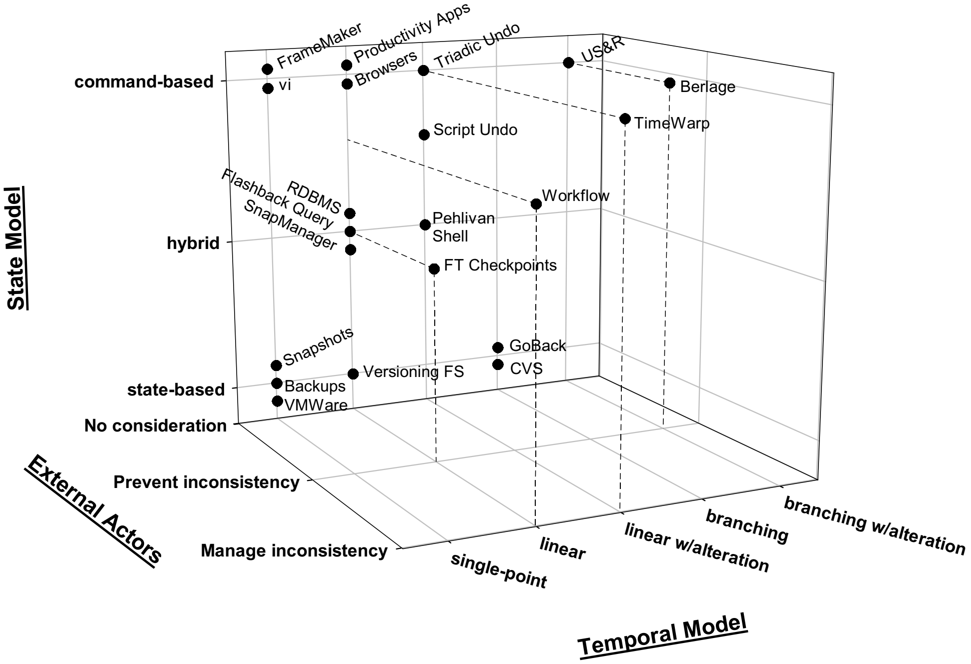


Figure 2-6: Design space of undo models. The design space is spanned by the three axes described in the main text: temporal model, state model, and consideration of external actors. Many of the example systems and literature models discussed in this chapter have been plotted in the design space.

sistent results, the message-passing system prohibits rollback beyond the latest point at which a message was sent to an external entity.

Finally, the undo model can allow inconsistencies to occur, then *manage* them to minimize their effect on external actors. This approach allows undo operations to roll the system back past points where state was externalized outside of the sphere of undo, affecting the timeline seen by external users. But it then requires that the undo system detect any resulting inconsistencies and take steps to mitigate their consequences. This is the kind of approach taken by database workflow systems, which define compensating transactions that update the state of any external actors to match the database's post-rollback timeline.

2.2.4 Locating Existing Undo Models in the Design Space

Figure 2-6 graphically depicts the undo model design space according to the sets of axes developed in the previous section. It also illustrates the points in that space occupied by several existing undo systems and models, including some of those discussed above in Section 2.1.

2.3 Ideal Undo Model for System-Wide Operator Undo for Services

Our goal at the start of this chapter was to identify the optimal form of a system-wide undo facility for the operators of network-delivered services. With the design space of undo models fully fleshed out, we now have the foundation we need to attack that problem. We will do so by considering each of the three axes of the design space in turn, identifying an optimal point on each axis.

Temporal axis. First, we have the temporal axis within the SoU, describing the history structure supported by the undo model. Ideally, an undo for operators should support the richest possible history structure: branched timelines supporting branch cloning and alteration, *i.e.*, a non-linear history model supporting selective undo. The most immediate benefit of such a history structure is that it provides extensive support for trial-and-error problem solving and exploration. Operators and system administrators often find themselves trying to solve unusual problems or correct novel misbehaviors [9] [28], and as such they may try many different solutions, encountering blind alleys along the way until they stumble upon the proper course of action. A branching timeline preserves the different paths that operators have taken (corresponding to different hypotheses), so that if necessary they can return to previous branches should later ones prove unsatisfactory.

We also include support for selective undo in our ideal temporal model, represented either by a US&R-like "skip" operation or more generally by branch cloning and alteration. These capabilities taken together allow an operator to alter a past action in the history while preserving later ones. For example, the operator may have installed three patches, and later discovers that the second patch is causing problems. Given branch cloning and alteration, the operator can clone the history branch corresponding to the three patch installations, edit the new branch to remove the second installation event, and then activate the new branch at the point following the third patch installation. This will leave the system in a state equivalent to one in which only the first and third patches were installed.

State axis. The example above of software patch management suggests that the ideal operator-undo system should represent history using a command-based approach. A command-based approach allows the operator to reason about specific events in the history (such as installing a software patch) and enables the altered redo necessary in a case like the patch management example above (when the operator enables the newly cloned and altered branch).

In fact, a hybrid approach that combines command- and state-based history records is most appropriate. While command-based history records are needed for redo, an operator-undo system must use state-based records (such as checkpoints) for undo. If not, it could not be used to recover from situations where the service behaves incorrectly as a result of software failure, hardware-induced corruption, or human configuration error. For example, consider a situation where the operator installs a software patch or upgrade, creating a command-based history record of that action. It is possible that the patch procedure incorrectly overwrites a key configuration file or application library. If this improper behavior is the unexpected result of a software bug or an unexpected system configuration, the command-based record of the software upgrade will not reflect the actual (erroneous) changes made to the system, and thus will not be properly invertible during the undo process.

Furthermore, relying strictly on a command-based approach limits the undo system to recovery of well-known actions. System administrators and operators typically have full reign over the systems they control, and often can bypass standard control interfaces to modify state directly. For example, in a Unix-like system, the administrator can access the file system underlying most complex applications directly, or can manually edit text-based configuration files maintained by the application; likewise, under Windows, the administrator can use the registry to make changes to the system that bypass the standard control-panel-based administrative interfaces. These changes could not easily be captured in command-based history records, as they are ad-hoc, unanticipated, and ill-scoped. In addition, many administrative interfaces today are so poorly matched to the needs of system operators that they are routinely bypassed, both to solve obscure problems that are not addressed by the controls in the interface, and even for common tasks where the interface impedes the administrator's most efficient workflow [9] [76]. These facts again limit the potential of a command-based history to capture the full spectrum of actions taken by human operators. In contrast, by using a state-based undo model, the undo facility can still recover from damage caused by human error during such unforeseen or untraceable operator actions, even if it can only redo the well-known subset of possible operator actions.

Consideration of external actors. The presence of external actors is unavoidable in an operator-undo model for services: the undo-enabled service will almost certainly have human end users to deal with, and the service itself may potentially interact with other external services. Undo-related inconsistencies from the point of view of these external actors are therefore nearly unavoidable, and cannot be ignored. Given the choice between prohibiting and managing these inconsistencies, we choose to manage them. If we wanted to prohibit inconsistencies, we would only be able to allow undo back to the last point at which an end user (or external service) interacted with the undo-enabled service; this restriction would make the undo system practically useless.

For external actors that are capable of performing their own undo—that is, external actors that are their own spheres of undo—we can manage undo-related inconsistencies by propagating the undo from the initiating service to those external spheres of undo; the subtleties and mechanics of this process are beyond the scope of this dissertation, but are considered in an accompanying technical report [15]. However, for external actors that are unaware of the undo process, notably human users and undo-unaware computer systems, our undo model needs to take steps to manage the external view of the undoable service's sphere of undo, so that work performed by external actors is preserved and so that those external actors see an acceptably consistent view of the service despite the timeline discrepancy across the sphere's boundary. Managing the external view of the sphere of undo is the single most important challenge in developing an operator-undo facility for services, and it will occupy our attention for much of the remainder of this dissertation.

2.3.1 From ideal model to practical target

In order to sharpen our focus on the challenge of managing the external view of an undoable service, we choose to step back from the ideal undo model characterized in the previous section, simplifying it along several of the design axes. Figure 2-7 illustrates the position of the ideal model and a simplified model in the undo model design space. The simplified model developed in this section will form the basis for our implementation and evaluation of an undo facility for service operators. Note that even with its significant simplifying assumptions, we will still face daunting research challenges in defining and realizing our chosen model for operator undo, notably in managing the external view of the undoable system to non-undoable entities; this is a challenge that most existing work on undo has been unwilling to tackle.

Temporal axis. Much research has already been done into history structures and temporal models for undo in more traditional domains like editors and other productivity applications, as surveyed above in Section 2.1.1, and most of that work is orthogonal to the novel challenges of external view that we face in building an undo facility for service operators. Rather than continue treading on well-worn ground, we choose to simplify our undo history model to the fundamental operation of undo, focus on the novel problem of managing external view, and leave as a future extension the complexity of carrying over more sophisticated history models from the literature.

As such, we discard the complexity of branching histories and selective undo, and move to an undo model with a simple linear timeline. While the technology, data structures, and implementation

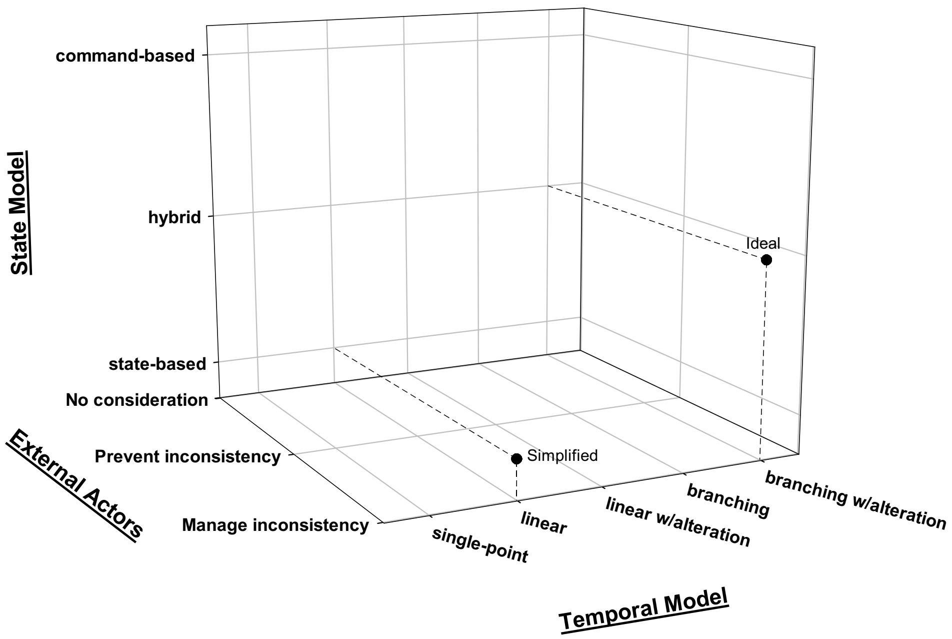


Figure 2-7: Ideal and simplified undo models for a system-wide undo facility. This figure illustrates where the ideal and simplified undo models fall within the three-dimensional undo model design space. The simplified model reduces the complexity of the temporal and state models in order to sharpen the focus on the challenging aspects of dealing with external actors.

techniques needed to support branching histories are not significantly different from those needed for linear histories, this simplification sets aside the challenges of developing sophisticated user interface technology to present the operator with a detailed, manipulatable view of the branching structure of time.

Furthermore, in a more significant simplification, our undo model will not provide a separate redo operation. Instead, it will provide a **committable-cancellable undo** operation, whose operation is illustrated schematically in Figure 2-8. This operator is a variant on the "truncate*" policy in the Script undo model [4]. A committable-cancellable undo operator defines an explicit undo phase in which the system operator can invoke undo to restore a system to prior points in the history, as many times as is desired. Once the system is undone, changes can be made as desired (in some sense defining a new, transient branch of history). However, there is no notion of individually redoing events in the history that have been undone. Instead, the entire undo process can be *cancelled*, discarding any changes and returning the system and the history to the state it was in when the undo phase was started, or *committed*, in which case the history as altered by the undo process is made permanent, and any undone operations are irretrievably lost.

This form of undo operator acts similarly to a linear undo/redo model, with the cancel operation acting like a "bulk redo" to restore everything undone by the undo operation. What is lost with this simplification is the ability to redo individual events in the history, to perform selective undo or modify the history before redo, and to redo any undone history events after the commit operation has been performed. What is gained, however, is a much simpler history model that requires only state-based records, and the potential for a significantly simpler implementation.

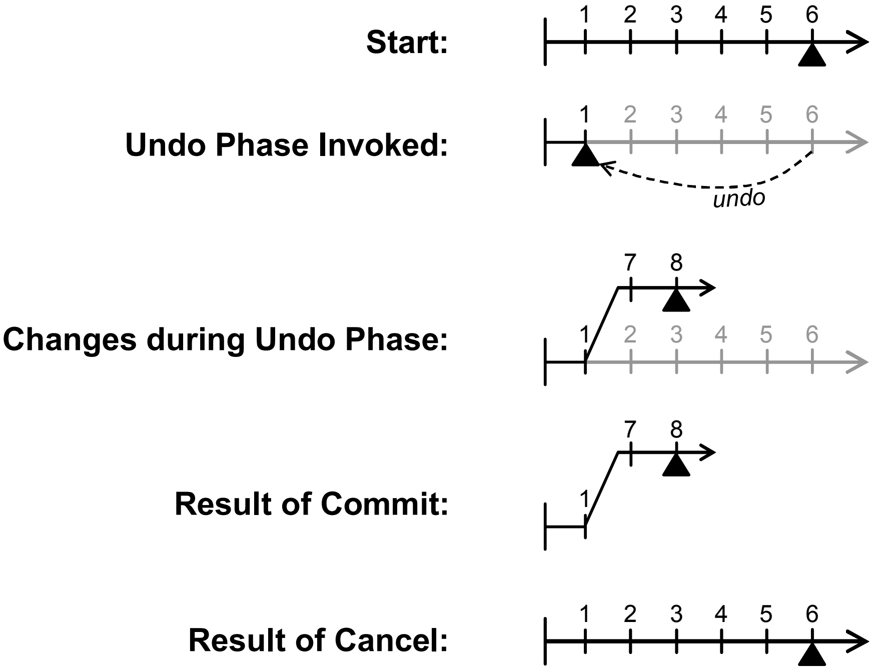


Figure 2-8: Operation of committable-cancellable undo operator. This undo operator works with an explicit *undo phase* that begins when undo is invoked. Changes made during the undo phase are tentative, and can be conceptually thought of as existing in a transient branch of history. The undo phase can be *cancelled*, in which case the tentative changes are discarded and the system is restored to its state before the undo phase began; or it can be *committed*, in which case the original history is discarded and the tentative changes are made permanent.

Despite its limitations, our simplified temporal history model still meets many of the needs of an undo facility for service operators. It still allows operators to carry out hypothesis testing (within the scope of an undo phase), although they will only be able to maintain history on a single hypothesis at once. And, while it does not allow the kind of history editing described in the software patch installation example above, it still is able to recover from state corruption due to a bad administrative operation, leaving the operator with only the need to manually re-execute later operations, a much easier task than having to clean up the mess left by the failed operation. Finally, there are actually advantages to providing a simplified temporal history model: it reduces the complexity of the undo operation and provides a simpler mental model to the person invoking it, thereby helping to avoid confusion, improve predictability, and mitigate the risk of machine-human interaction [29] [75].

State axis. Because our temporal simplifications have left us with the need to only support a cancellable undo for operator actions, we can drop the requirement of capturing the system operator's actions as commands to use during command-oriented redo. A state-oriented history is sufficient: it provides the needed state checkpoints for carrying out undo, and can provide the needed cancel behavior by restoring a state checkpoint added to the history when the undo phase is initiated. Despite its limitations, this simplification also has potential advantages in terms of the usability of the undo facility: since restoring a state checkpoint undoes *all* changes since the checkpoint, checkpoints provide a simpler mental model to the system operator than a system that performs selective undos of individual commands, again reducing confusion and risk.

Note that if our simplifying temporal assumptions are relaxed, we will need to relax this simplifying state assumption as well. Doing so would involve developing a command-oriented framework for operator actions, modifying the service system to generate and replay those commands, and integrating the resulting command history with the undo facility's implementation of redo.

Consideration of external actors. The third axis of the space of undo models is the one where we want to simplify the least, since it defines the most interesting and least well-explored parts of the undo space. Therefore, we will retain the design point of our ideal model, and will manage undo-induced inconsistencies rather than ignoring or preventing them.

To find some traction on the external inconsistency problem, we will make one simplification: we will only consider one class of external actors, human end-users. Our simplification restricts us to *self-contained services*, which we define as services that communicate solely with human users. Equivalently, the boundary of a self-contained service's sphere of undo is only crossed by information that ultimately is delivered to a human user.[1] Note that a self-contained service can actually be composed of several sub-services, but they must all be contained within the same sphere of undo, implying that their combined state is treated as a single entity with respect to undo.

This restriction to self-contained services is not as onerous as might be imagined. A significant number of important services fit into the self-contained service mold, including communication-oriented services like e-mail, messaging, calendaring, and news; some commerce-related services such as online auctions and centralized shopping sites; and most information-retrieval services such as search engines, archives, and news dissemination services. The services that cannot be supported in this restricted undo model are mostly the services that form infrastructure for other services or computer systems: for example, file- or block-storage services, SQL-level database services, network name or directory services, or pure computation services. When these lower-level services are coupled with the applications that use them to deliver services to end-users, however, the combinations can be treated as single spheres of undo providing self-contained services to human users, and then they fit into the undo model we are proposing.

2.3.2 Discussion

Our simplified model for system-wide undo sacrifices the richness of the ideal model along the well-explored temporal and state axes, preserving the minimum of functionality needed to make system-wide undo useful for system operators and administrators. With only a linear temporal model, operators will not have the flexibility to maintain multiple system states in parallel, but will still be able to retroactively repair problems and undo the effects of human error; moreover, the simplified committable-cancellable undo operator is easier to understand and reason about than a fully-general branching undo operator. With only a state-based state model, system operators cannot selectively undo or replace individual system-level commands directly, but they still retain the power to undo erroneous commands via the bulk state-based undo operation.

On the other hand, our simplified system-wide undo model keeps nearly the full complexity of the most challenging and unexplored axis of the undo model design space: consideration of external actors. It is along this axis that the most novel contributions of our work are located, starting with an

[1] More precisely, information from other computer systems may cross *into* the sphere of undo, but no information or responses may flow *out of* the sphere to any entity other than a human user.

approach to handling undo-induced inconsistencies visible to external users of a self-contained service. The next chapter introduces and discusses that approach.

Chapter 3

Managing Undo-Induced Inconsistencies with Re-Execution

> "Twice and thrice over, as they say, good is it to repeat and review what is good."
>
> — *Plato*

In the previous chapter, we established a simplified model for system-wide undo that uses a state-based committable-cancellable undo operator. We identified the problem of undo-induced inconsistencies, where entities outside of a system's sphere of undo can see discrepancies when the undo process detaches the timeline inside the sphere from the timeline of the external entities. We now begin to tackle the problem of how to manage those inconsistencies, defining an approach at the level of spheres of undo, and establishing a vocabulary for our ongoing discussions of undo-induced inconsistencies. Chapter 4 will delve into the details of how this chapter's high-level approach can be translated into a practical architecture for building real undoable systems.

3.1 The Re-Execution Approach and the Three R's

With our scope limited to self-contained services, the greatest challenge in managing external inconsistencies is in making sure that end-user work is not lost as a result of undo. Consider the case of an e-mail service where an operator accidentally botches the installation of a software patch, or accidentally deletes an important configuration file. After some time passes, the operator realizes the problem and decides to use undo to recover, then tells the system to undo to a point before the erroneous action. As the system rolls back its state, it blindly rolls back all of the end-user work that has taken place between the rollback target point and the time when undo was invoked. Without special consideration of the end users, they will lose all of the e-mail they have received in that interim period, and any actions they had performed (such as deleting or refiling messages) will be lost as well.

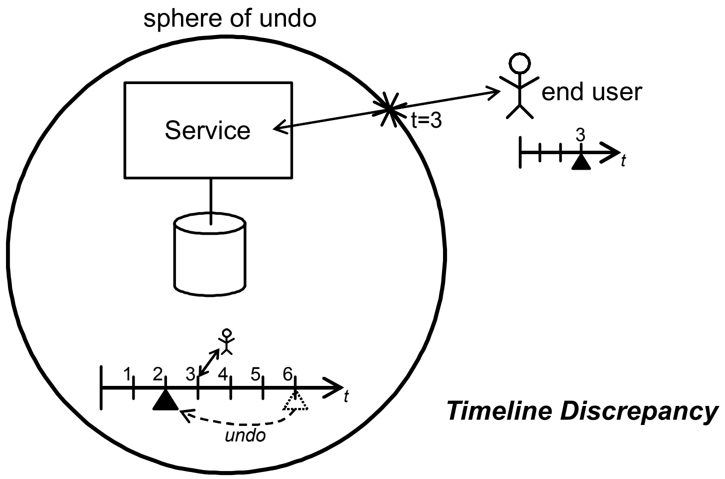


Figure 3-1: Example of undo-induced inconsistency addressed by re-execution. In the depicted scenario, the timeline inside the sphere has been rolled back by undo to *t*=2, discarding the work performed by the user at *t*=3 and causing an inconsistency for the end user (who erroneously thinks their action at *t*=3 is still reflected in the system's state).

A traditional management approach would be to define compensating actions to use whenever an end user lost work as a result of undo. But this approach is unsatisfactory, since the amount of work lost could be significant and no effective compensation might exist for that lost work. Another traditional approach would be to try to isolate end user state from system state. But this again is insufficient, since problems with system state can cause incorrect processing of end-user work and can corrupt user state, particularly when the system state problems being repaired involve configuration errors, software bugs, or failed upgrades. (As an aside, these kinds of propagating failures are serious problems in real-world systems, for example causing a multi-day outage in the eBay auction service when an OS bug corrupted the auction database [35].)

To find an alternative to these limited traditional approaches, we return to the construct of spheres of undo (SoU's). When system operators invoke undo within a sphere of undo, they implicitly break the connection between the timeline within the SoU and the timeline maintained by external entities, like the human end users in our self-contained services design point. In particular, the end users' timelines remain in the present, while the SoU's timeline rolls back, as Figure 3-1 illustrates. It is necessary to roll back the SoU's timeline, since that is how undo recovers from problems affecting the system's state. But this rollback also causes end-user work to be lost: end users think that they have performed operations (reflected in their views of the timeline), but those operations are now no longer part of system state.

To resynchronize the timelines, recover lost end-user state, and thus manage inconsistency, all while still allowing undo of system-level state, we can re-execute, or **replay**, the end-user interactions that were lost as a result of the undo. By replaying interactions rather than restoring state directly, we can be sure that the interactions are reprocessed and take into account the effects of the undo.

This replaying of end-user interactions sounds a lot like a command-based redo, and in fact it has a natural interpretation in terms of undo models and spheres of undo. As Figure 3-2 illustrates, we can recast the single sphere of undo for a self-contained service as a **nesting** of two spheres of undo: an inner sphere that logically holds the portion of system state visible to end users, and an outer sphere

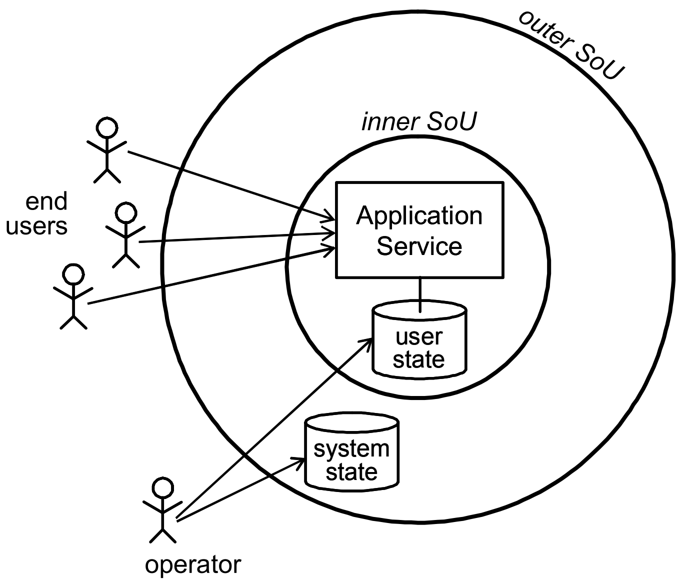


Figure 3-2: Recasting undo as nested spheres of undo. The inner sphere logically contains the portion of system state visible to end users; end users interact directly with this sphere. The outer sphere contains the inner sphere and additionally captures all non-end-user application state and system state; the operator interacts with this sphere. The outer sphere provides the system operator with a state-based committable-cancellable undo operation, while the inner sphere implicitly provides a state-based undo/command-oriented redo model for end-user state.

that contains the inner sphere's state plus all system-level state. End users interact directly with the inner sphere—their operations pass through the outer sphere—and system operators interact with both. When system operators invoke undo, they cause state in both spheres to roll back. When they commit an undo operation, the undo system causes state in the inner sphere to roll forward again to the present, resynchronizing with the timelines of its end users; the roll-forward is accomplished by redoing end-user operations that were lost during the undo.

In this construction, the nested spheres of undo define a layering of undo models. The outer sphere has the undo model we described in the previous chapter (Section 2.3.1): it provides the system operator with a committable-cancellable undo operation using a state-based temporal history. The outer-sphere model achieves its consideration of external actors by way of the undo model in the inner sphere, which is a linear undo/redo model *for end-user state* using state-based undo and command-based redo. It is important to realize that this nested-spheres formulation of our undo model does not magically provide redo functionality for the system operator, nor does it provide command-based undo for the end user: the operator still has only the committable-cancellable undo operation at his or her disposal, and end users have no explicit undo/redo functionality exported to them. The command-based redo of end-user interactions is simply part of the process by which the committable-cancellable undo operation is implemented, and happens implicitly when a system operator's undo operation is committed.

To provide a convenient vocabulary for discussing our nested-spheres layered undo model, we dub it the **Three-R's undo model** based on the three steps that define its undo process (from the system operator's perspective):

1. **Rewind:** all state within both nested spheres of undo is rolled back in its entirety to a prior version, as recorded in the history. This approach implies that the history includes periodic or continuous checkpoints of the entire service system's state, including system-level state like OS and application binaries and configuration, as well as application-level state like end-user data. The rewind operation defines the start of the undo phase, and is the analogue of invoking the system-level undo operation. Multiple rewind steps may be performed, providing multiple-undo functionality.

2. **Repair:** the system operator can optionally make any desired changes to the system. This step provides some of the flexibility lost as a result of abandoning command-oriented redo for system-level state: while system operators cannot edit histories of previously-executed commands, they can choose to manually re-execute them, execute variations, or not execute them at all as part of the repair step. For example, had the operator made an erroneous change to a configuration file, or performed a failed upgrade, he or she could Rewind the system to before the error then Repair it by making the correct configuration change, retrying the upgrade, or simply omitting the erroneous action altogether.

3. **Replay:** all rolled-back end-user interactions with the inner sphere of undo are re-executed against the repaired system. From the system operator's perspective, Replay does not alter the timeline within the outer sphere of undo, as Replay restores only end-user interactions, not operator commands. But for the external end-user, Replay accomplishes the goal of maintaining a consistent external view of the system: all end-user work and state changes are reprocessed following the repair, bringing the timeline within the inner sphere of undo back into sync with the external end-users' timelines.

   The Replay step represents the commit aspect of the undo operator in our prior formulation of the undo model; the cancel aspect is accomplished *without* Replay by simply restoring a checkpoint of the system taken before the initial Rewind. Note that the Replay step implicitly requires that, during normal, non-undo-phase operation, the system must record all end-user interactions with the system so that it can later replay them.

The key step in this process is Replay, as it is through Replay that we tackle the problem of maintaining an acceptably consistent view of the service's sphere of undo to external users. The inclusion of a replay step in our undo model, combined with the willingness to replay external interactions even after the system has changed due to Repair, sets our undo model and process apart from the existing research work on undo and defines its key novel contribution.

A convenient metaphor for understanding the Three-R's undo process 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. Three-R's undo offers a similar sequence of events. The rewind step is the equivalent of traveling back in time within the sphere of undo, in this case to the point in time before an error occurred. The repair step is equivalent to changing the timeline inside the sphere: 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 re-executing—in the context of the repaired system—all events in the external 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.

Thus, the Three-R's process enables a capability that we denote **retroactive repair**—the ability to change the system in the past to fix an already-present problem, such that those repairs propagate forward and cancel out the effects of the problem on state within the sphere of undo. In other words, as the undo facility replays end-user operations, it restores the user-initiated state changes and incoming data that were lost during Rewind in a manner that retains their intent and not the (possibly incorrect) results of their original processing. Retroactive repair provides service operators with a powerful recovery capability, allowing them to fix problems after they have already manifested themselves, even to end-users. For example, an e-mail service administrator could retroactively repair a problem with a spam or virus filter configuration that has erroneously discarded or corrupted user e-mail; and could recover from a problem like the aforementioned eBay outage by retroactively repairing a software bug that, once triggered, corrupted critical user and application state.

One of the properties of retroactive repair is that, like Three-R's undo in general, it is primarily effective for fixing problems that affect *state*. It is hard to imagine how to retroactively repair a problem that, say, transiently caused user interactions to fail, since those failures would have been immediately visible to their corresponding users. Of course, Three-R's undo could still be used to rewind and repair any state changes that provoked such failures; it is the manifestation of the failures themselves that cannot easily be retroactively repaired. As we will discuss later in Chapter 8, there are certain cases where it is possible to retroactively repair failed interactions, but only in specific cases where the failure notification is asynchronous to the interaction's actual execution.

3.2 Verbs

In order to perform the Replay stage of the Three-R's, the system-wide undo facility must maintain a record, or **log**, of all end-user interactions that have crossed the boundary of the system's sphere of undo. To do so, it encapsulates each incoming user interaction into a construct we denote a **verb**; verbs are then collected into the log and used later for replay. The actual format of verbs and their collective log can vary widely (one specific example is described in the next chapter), but all verbs share some common properties:

- each verb encapsulates a single end-user interaction;
- each verb contains enough detail about the interaction's operation and parameters to be able to replay that verb at a later time;
- each verb contains a record of any output that crosses from inside the sphere-of-undo boundary out to the external end user.

Verbs and verb logs bear more than a passing resemblance to database transactions and database logs, perhaps not surprising given the parallels between database spheres of control and our spheres of undo. There are two key differences, however. The first is that verbs are defined at the granularity of user requests, which can be much higher-level than traditional transactions, and in fact may generate multiple transactions in underlying service databases as they are processed. Consequently, verbs do not necessarily possess the properties typically presumed of transactions, such as the ACID properties [47].

Second, and more important, is that verbs contain a record of the external output associated with the end-user's request; we will see shortly how this record is used to make it possible to complete our inconsistency management framework.

Verbs serve several purposes in our Three-R's formulation of system-wide undo. First, they make Replay possible by constituting a permanent record of the end-user interactions that need to be restored to avoid externally-visible inconsistencies. Second, they provide a convenient way to predict the effects of a Three-R's undo cycle by exposing exactly what state will be preserved or discarded during that cycle. Since verbs are defined for all end-user interactions, and only for end-user interactions, they map exactly to the set of state changes that will be restored (redone) by Replay.[1] Thus, by looking at the set of verbs that is defined for a particular service, the service's operator can understand exactly what state changes will be preserved by the Three R's. Any state changes *not* represented as verbs, such as system-level configuration changes, patches, and upgrades, will be discarded by the Three R's. Furthermore, by looking at the particular instances of verbs in the timeline log, the service's operator can tell exactly what specific end-user interactions and state changes will be restored for a given undo cycle.

Finally, verbs play one more crucial role: they provide a locus for annotating end-user interactions with additional information about how users perceive the consistency of the output they see. This feature will be the key to completing our approach to managing undo-induced external inconsistency, as we will see in the next section.

3.3 Paradoxes

There is an important subtlety with Replay that we have been dodging up until now: what happens if the system has changed enough (as the result of Repair, or simply of Rewind) that the replayed end-user interactions fail or produce different results? In such a case, the Replay process would *not* completely restore the end user's original view of the timeline, leaving unmanaged external inconsistencies. These unmanaged inconsistencies are **paradoxes**, and are a fundamental problem in a system-level undo system.

A paradox occurs when the Replay process does not generate the same results as the original execution history of end-user interactions with the service. In terms of the spheres of undo framework, a paradox occurs when information that has crossed the sphere of undo boundaries from the inner sphere to the outside world becomes invalid. Or, in terms of verbs, a paradox occurs when the output produced by a replayed verb does not match the output generated during the verb's original execution. Given that we have Replay, this situation can only happen when Rewind and/or Repair alter the system enough that, during Replay, the timeline inside the inner sphere of undo moves forward along a different path than during the system's original execution. Figure 3-3 illustrates one such scenario, showing how retroactive repair can cause previously-exposed information to be altered before being re-exposed during Replay.

We call these situations paradoxes because of their similarity to the "time paradox" device that is a staple of science fiction stories. In such stories, a time-travelling protagonist goes back in time and makes alterations to the past timeline. Upon returning to the present, the protagonist, whose memories are typically isolated from the altered timeline, sees the "new" present as inconsistent. This is simi-

[1] In the nested spheres of undo interpretation of the Three-R's undo model (Section 3.1), the state changes defined by verbs map exactly to the state changes occurring in the inner, end-user sphere of undo.

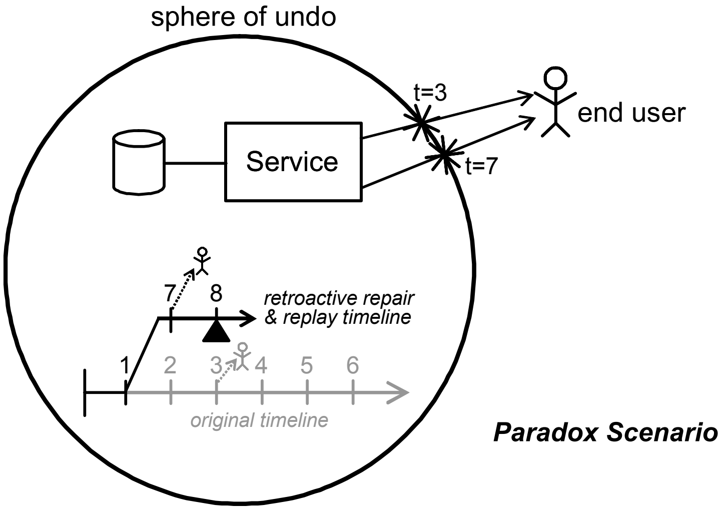


Figure 3-3: Scenarios that cause paradoxes. This diagram shows how paradoxes can be caused by retroactive repair. Only the inner sphere of undo in the nested model is shown here. After the end user was sent a message at *t*=3, the sphere was rewound to *t*=1, a repair was performed, and then replay was invoked, replacing the original timeline and causing alternate data to be exposed at *t*=7. The paradox in this case results from the fact that the end user should not have seen the original message set at *t*=3.

lar to what happens with our spheres of undo: changes to the timeline within the sphere of undo cause external observers (who are isolated from those timeline changes) to see the contents of the sphere as inconsistent. 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; inconsistencies become paradoxes only when they are in state that has been previously exposed in the output of a verb.

3.4 Coping with Paradoxes

In general, paradoxes are unavoidable in our undo model. To be useful to the operator, our model permits permanently rolling back administrative changes to the system, which is what happens when the operator invokes and immediately commits an undo. This capability is mandatory to allow the operator to recover from human error and system failure. Furthermore, the undo model allows the operator to change the system arbitrarily before committing the undo phase. This capability is also mandatory, as otherwise the operator would be denied the ability to retroactively repair latent problems that surface unexpectedly, like software bugs, hardware-failure-induced corruption, and latent configuration errors. Given that both of these scenarios, rollback of administrative changes and retroactive repair, can affect the way that end-user interactions are processed, paradoxes will be a natural consequence of providing undo to service operators.

Most existing approaches to undo avoid the paradox problem by constraining the undo model to make them impossible. In the terminology of Berlage [10], these approaches attempt to preserve the *stable execution property*, namely:

> "A command is always redone in the same state that it was originally executed in, and is always undone in the state that was reached after the original execution" [10].

For example, most editors and single-user productivity applications achieve stable execution by prohibiting redo of undone actions once a change to the active document has been made. Fault-tolerant checkpoint/message-logging systems prohibit undo across points where information has been made externally visible (the notion of "output commit" [121]); while this approach prevents paradoxes, it also limits recovery—in our case, it would deny the operator the ability to undo as soon as any user interacted with the system, making the undo facility essentially useless. Relational databases using undo/redo recovery provide stable execution by not allowing committed transactions to be altered during redo/replay: they avoid paradoxes by disallowing retroactive repair and insisting that replay follow the same (or equivalent) trajectory as the original execution.

An alternative to preventing paradoxes is to detect them as they are about to occur and resolve them before they are manifested to external entities. Detecting paradoxes before-the-fact generally requires a sophisticated understanding of the differences between the two or more timelines in question. Typically, this approach is taken by systems that try to resolve conflicts between multiple histories consisting of well-understood commands. For example, the Timewarp system allows multiple collaborating users to maintain their own autonomous timelines on a shared document; paradoxes in Timewarp arise when one user invokes undo or edits his history unilaterally, causing conflicts between those autonomous timelines. Timewarp detects paradoxes by analyzing the commands in those timelines using an understanding of the semantic compatibility of the different commands [32]. Once a paradox is detected, it is resolved either by manipulating the command history to semantically remove the conflict, or by interactively presenting the conflict to the user for resolution.

A Timewarp-style approach is inappropriate for our operator undo model for services, though, since it requires more semantic understanding of the system history than we can provide while still maintaining a useful level of recovery power. To apply the Timewarp approach, we would have to know ahead of time how the Rewind and Repair processes might affect the user interactions re-executed during Replay. Given that our undo model does not record operator actions and does not constrain what can be changed during Rewind and Repair, it is virtually impossible to understand the system changes well enough to predict the effects of Replay; if we do not assume bug-free software and perfect hardware, the prediction problem becomes even more intractable.

An alternate approach taken by some replicated file systems designed for weakly-connected operation, like Bayou [126] and IceCube [60], is to resolve similar paradoxes by reconciling one command history against a snapshot of system state. This approach too is inappropriate for our needs, as it again presupposes a semantic understanding of all the consequences of a command (verb) on system state, and how those consequences are affected by any operator actions rolled back or initiated during Rewind and Repair.

Thus, we are left only with the option of allowing paradoxes to occur, managing them after the fact to try to mitigate their effects on external observers. This option implies an approach based on **compensation**: taking corrective action to resolve paradoxes or to mitigate their significance in the perception of the external end-user. The compensation approach is best demonstrated today in database workflow systems (a workflow is a sequence of transactions designed to accomplish a higher-level goal). Paradoxes arise in workflow systems when an earlier transaction in a workflow is identified as problematic and needs to be undone, yet that or subsequent transactions have already had externally-visible effects. Workflow systems handle these paradoxes by requiring that every transaction with external effects define a compensating transaction that can cancel out those effects. When a problematic transaction is detected, the workflow system rolls back the entire workflow transaction-by-transaction, using the compensations to reverse external effects at each step, until the problematic transaction is itself undone and compensated for [43] [133]. Korth *et al.* present an alternative approach where the compensation is performed only for the problematic transaction, allowing the effects of later transactions to remain, subject to commutativity constraints [64]. Workflow systems thus define a form of undo that provides Rewind within the context of a workflow execution, and that can handle paradoxes resulting from Rewind when compensations exist.

Our undo model has somewhat different requirements, however, and thus the workflow approach is not directly applicable to the problem at hand. Our undo model concerns itself with paradoxes that are visible to human end-users, an entirely different type of entity than the reservation systems, banks, and other database-based services typically assumed by workflow systems. We cannot run compensating transactions on users' brains, erasing the information that has previously been exposed to them. So we must develop a novel approach, one that recognizes that human end-users are not transactional yet are still capable of processing the inconsistencies underlying paradoxes. We will do this by leveraging the inherent tolerance for less-than-perfect consistency exhibited by nearly all applications that involve human end users, and by encoding the specifics of that tolerance and consistency model into verbs. The key role of verbs in paradox management is thus revealed: they provide a framework that allows our undo facility to reason about the significance of paradoxes and, if necessary, take the appropriate compensating steps to help end users reconcile their mental timeline with their post-undo view of the system.

In taking this compensation-based approach, we will have to accept that paradox handling will necessarily be imperfect, but that the recovery benefits of undo make such imperfection tolerable. Without full knowledge of end users' thought processes, there is no way to know precisely what end users will see as paradoxes and what they will ignore. Likewise, there is no way to perfectly compensate for a paradox that has occurred, as the user may have taken actions—potentially actions invisible to the system, like making phone calls—based on the data involved in the paradox. Our approach will therefore be a conservative one: we will declare paradoxes whenever there is a significant discrepancy in the external view of the sphere of undo, leaving end users with the responsibility of filtering out the paradoxes that they care about; similarly, we will compensate for paradoxes as completely as possible given the information available to the undo system, but will preserve and present enough information to the end user that they may take whatever steps are necessary to compensate for their own external actions affected by the paradox.

The next chapter will concern itself with an undo architecture that uses and extends the verb construct to accomplish these goals; we will then go on to describe case studies of how verbs and paradox management can be combined to provide system-wide undo for e-mail and auction service applications.

Chapter 4

An Architecture for Three-R's Undo for Self-Contained Services

> "Time will run back and fetch the Age of Gold."
>
> — *John Milton*

Having established the Three-R's undo model as an appropriate design point for an operator-targeted undo facility, we now turn to the practical considerations of how to actually design and build a Three-R's system. In this chapter, we limit our scope to self-contained services as defined in Section 2.3.1; that is, services contained within single spheres of undo where the only information flow out of the SoU is to human end-users. Chapter 9 will revisit the question of designing undo functionality for services with more complex structures.

As identified in the previous chapter, our greatest challenge in creating an undo architecture lies in detecting and managing paradoxes. A secondary but related challenge is in generating a record of time that accurately reflects the history of end-user interactions with the service, that captures the intent of all state changes made by the service's end-users, and that can be replayed later while still respecting the alterations made to the system during Repair. We will address these two challenges in depth below, after first laying out some basic design assumptions and establishing the overall architecture of the undo system.

4.1 Design Points and Assumptions

The goal of our undo facility is to support the operators of service systems, allowing them to recover from their own errors and to retroactively repair other system-level problems that affect the service's dependability. Our focus is on recovering from problems that affect a system's **hard state**; that is, persistent, on-disk state encompassing user data, application and system configuration records, and system binaries. Such problems might include erroneous configuration changes, accidental deletion of data due to operator error (*e.g.*, "`rm -rf *`"), data corruption resulting from a failure during storage reorganization, system corruption due to a virus or external hacker attack, failed upgrades or patches to the service application or OS, software bugs (which can be seen as errors in the hard state holding software binaries), and so on.

As we have defined it, the Three-R's model of undo is not an appropriate recovery tool for problems that only affect soft state or simply cause failstop crashes; there are a plethora of orthogonal techniques for recovering from such problems, typically based on restarting the affected system. Furthermore, Three-R's undo is not targeted at solving the problem of corruption or destruction of hard state caused by the correct processing of misguided end-user actions. For example, consider the case where a user of an e-mail service explicitly but accidentally deletes her mailbox. To use Three-R's undo to recover from this, the system operator would have to remove the user's `delete` verb from the record of end-user events before invoking Replay, as otherwise the deletion would be re-executed and its effects restored. While it is possible to make such changes to the recorded history of end-user operations in our architecture, it is not an operation for which the system is optimized; rewinding and replaying the entire service is likely to be too heavyweight for such cases. Chapter 9 will discuss an approach to hierarchical undo that makes the Three-R's approach more tractable for repairing mistakes made by end users.

4.1.1 Self-contained services

Our undo facility is targeted at a design point of self-contained services, or single, standalone spheres of undo. The self-contained service model implies several further design points. First is the assumption that, given a service, we can identify all points at which the boundary of its sphere of undo is crossed, where requests enter the sphere and data leaves it. We must be able to intercept those boundary crossings both to record the verbs that define a history of interaction with the service and to later replay that history. Typically, this implies that the service communicates with its end-users via some sort of narrow, well-defined interface or **protocol**. We will assume that this protocol is a request-response protocol, with information flowing out of the sphere of undo only in response to user requests. This assumption matches the user interaction model of most services, with the exception of "push" services that stream data, unsolicited, to the end-user. Note that unsolicited push *input* to the service, like a live stock quote feed, can be treated as a sequence of response-free requests; it is only push *output* that does not fit our assumptions.

Our assumption of a protocol-based service interface also implies the assumption that the protocol is stable across an undo cycle. If the protocol is changed or extended, the undo system needs to be extended as well to understand the new protocol. Furthermore, if the protocol changes, the undo system may not be able to successfully replay historical end-user interactions, and hence may lose end-user work. For example, if a service upgrade is performed to add a new type of interaction, and subsequently the upgrade is deemed faulty and rewound using undo, it will be impossible to replay any of the recorded instances of the new interaction, unless they can be re-expressed in terms of the original interaction set.

Finally, the self-contained service design point encodes the assumption that the service's hard state (storage) is centralized and self-contained, and not intermingled with the state of any other service. In other words, it must be possible to identify the boundaries of the service's state, or more precisely, the state affected by end-user requests to the service. These boundaries correspond to the boundaries of the service's corresponding sphere of undo; if it is not possible to identify them, then it is likewise impossible to define a sphere of undo and hence impossible to provide consistent rewind and replay functionality.

4.1.2 Black-box services

Our undo system architecture treats the service to which we are adding undo as a **black box**: the architecture does not require any knowledge of the internal implementation details of the service, and the service itself is not modified to provide undo functionality. Instead, we wrap the undo system around the service by interposing on its protocols and storage traffic. The primary advantage of taking a black box approach is that we can use existing, unmodified service implementations; it is not necessary to rewrite a service or even understand all of its internal implementation details in order to enhance it with undo functionality. On the other hand, we do assume that the service's *protocols* are transparent and well-understood—indeed, we will see that the black-box approach requires an intimate understanding of the service's protocol specifications and behavior.

Choosing a black-box design point rather than integrating undo functionality into the service itself also has the advantage of allowing 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. The ability to support such unconstrained repairs is a unique advantage of our approach to undo, and is a crucial factor in enabling recovery from system-level dependability problems.

Another advantage of treating services as black boxes is that it makes the undo facility reusable. The undo facility for a particular service protocol can be implemented once and reused without changes for different implementations of that protocol—for example, allowing an undo system for e-mail to work with any mail server supporting the IMAP and SMTP protocols. The undo facility's reusability can be increased further by building it as a generic core with protocol-specific knowledge “plugged in” based on the specific type of service being used. 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.

There are drawbacks to a black-box service design point, however. It puts a lot of pressure on the protocol to expose enough information to handle paradoxes, as we will see below. As a result, some service protocols may have to be modified to make them suitable for undo; Chapter 8 presents guidelines for making these modifications. A black-box design also requires extra machinery to reverse-engineer information that would be easily visible inside the service, such as request execution order. And it adds performance overhead, since all recording and replaying of user interactions must be done through proxies outside the service.

4.1.3 Fault model

Despite the drawbacks of the black-box service design point, there is one key reason why we chose it, an advantage we have not yet discussed: treating the service as a black box means that we need not make assumptions about its correctness, and can therefore recover from situations where the service is explicitly *not* behaving properly. Anything goes within the black box; from its vantage point of a proxy outside the service, the undo facility can record what end-users are trying to do with the service (their intent, as expressed by the protocol commands they generate), regardless of what actually happens to those protocol commands once they cross into the black box. Were the undo system built into the service itself, then it would be difficult to recover from situations where the service was corrupted—be it by a failed upgrade, a human operator error that trashes state, a software bug, and so on—because the corruption at the service level could affect the undo system as well.

Thus, the black-box service model buys us a measure of fault isolation between the undo system and the service being protected by it. Our fault model for the service itself is very relaxed: it can process requests in essentially arbitrary ways, as long as it does not generate external output via non-protocol channels and as long as it is able to at least accept well-formed protocol commands after it has been repaired as part of the Three-R's undo process. The former restriction ensures that we can intercept all external output to use for paradox detection, and the latter restriction guarantees that we can always attempt to replay end-user interactions after the Repair phase of the Three R's. While this relaxed 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.

In contrast, our fault model for the undo system itself and the infrastructure supporting it is very restricted. For simplicity, we assume that the undo system and supporting infrastructure (including the storage layer) are free of bugs and behave correctly at all times. This is a significant but unavoidable assumption, since any recovery mechanism eventually has to be either trusted or surrounded with a further layer of recovery; and it is hard to see how a such a system-wide recovery mechanism as Three-R's undo could be further protected by a broader-scale recovery system. We can, of course, use defensive techniques to construct the undo system implementation so that it behaves predictably when it fails (and in fact we have done so for the prototype described in the next chapter).

Furthermore, while not likely true of our experimental prototypes, there are several reasons why the fault-free assumption could be supported for a widely-used or commercial undo system. First, the undo system is reusable as discussed in the previous section. Second, assuming the protocols are reasonably stable, the undo system itself will evolve at a slower pace than the service itself. Third, a well-implemented undo system (like our experimental prototype) uses the same code paths for normal-case execution and for replay and recovery, and makes it possible to explicitly test all possible paradox cases and compensations. These three reasons taken together suggest that the code for a widely-used undo system will be well-exercised and inspected, reducing the possibility for damaging bugs and misbehavior.

4.2 Undo System Architecture

Figure 4-1 depicts the overall structure of our undo system design. 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 discussed above by keeping the undo system isolated from problems and changes in the service itself.

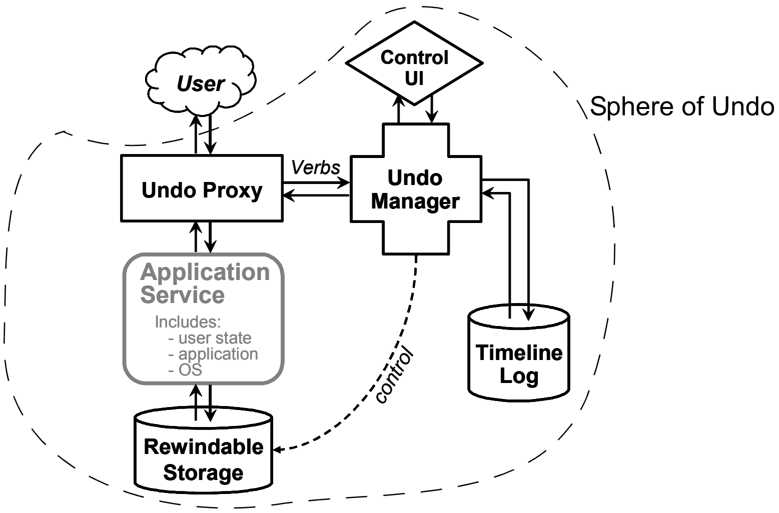


Figure 4-1: Undo system architecture. The heart of the undo system is the undo manager, which coordinates the system timeline. The proxy and rewindable storage layer wrap the service application, capturing and replaying user requests from above and providing physical rewind from below.

Below the service application's operating system, a rewindable storage layer provides the ability to physically roll the system's hard state back to a prior point in time; this layer could be implemented by a snapshot-based file system like Network Appliance's WAFL file system [53], a versioning system like Elephant [108], or a suitably modified non-overwriting, log-based file system like LFS [101] or GoBack [102] [111].

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 more 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.

Note that the service in Figure 4-1 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. This matches our sphere of undo model for self-contained services; we will examine how the undo system architecture can be extended to support finer-grained rollback in Chapter 9.

4.3 Verbs

The only service-specific component in the architecture of Figure 3-1 is the proxy that interposes on the service's user-request stream. Clearly, the proxy itself will be service-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 manage paradoxes and to generate an accurate record of the aggregate history of all end-user requests.

The answer to this seemingly contradictory set of requirements lies in the same **verb** construct that we introduced at a high level in Chapter 3. Recall that a verb is the 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). Typically, verbs correspond to operations in the service's protocol, although they can also represent an agglomeration of protocol operations or subsets of an operation; the basic rule is that one indivisible operation or interaction from the end-user's point of view defines a single verb.

In our undo architecture, a verb's encapsulation of an interaction consists of several parts:

- a record of the type of operation being performed
- a record of the parameters to the operation
- a record of **externalized output**, or any output generated by the operation that goes to external entities
- a record of whether the operation succeeded or failed.

Verbs record the operation type and parameters so that the operation can be replayed later as part of an undo cycle; they record externalized output and failure status in order to make it possible to detect paradoxes, as described below in Section 4.5.1.

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, and to detect and manage paradoxes.

Verbs are used in the undo system during all phases of operation, as Figure 4-2 illustrates. 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 generates 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 verb log is typically left unaltered as repairs are made to the service itself, but it may be edited to remove, replace, or add verbs if desired. 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 detects paradoxes on a verb-by-verb basis and handles them by invoking verb-supplied compensation routines. 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.

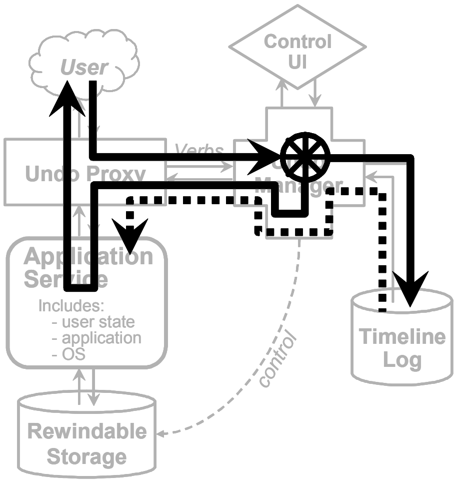


Figure 4-2: Illustration of verb flow. During normal operation, the verb flow follows the solid black arrows, with verbs created in the proxy and looped through the undo manager for scheduling and logging. During replay, verb flow follows the heavy dashed arrow, with verbs being reconstructed from the timeline log and re-executed via the proxy.

There are two significant challenges to implementing the verb flow of Figure 4-2. The first is the challenge of actually generating the verb log—the recorded sequence of verbs—so that it records the intent of end-user interactions rather than any erroneous effects of those actions, and so that it represents a causally-correct serialized version of the history of end-user interactions. The second challenge is in using the framework provided by verbs to properly detect and manage the paradoxes that may occur when the verb log is replayed in the context of a repaired system. We discuss these challenges and our solutions to them in the next two sections.

4.4 Generating the Timeline Log

For a timeline log to be usable for replay, it must provide an accurate record of the operations that end users have performed on the service. In other words, replaying the log should produce the same end result as the original execution, and the entries in the log should be decoupled from any misbehaviors of the service itself so that the log is still meaningful when replayed after repairs.

4.4.1 Decoupling verbs from service behavior

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. Designing verbs to capture intent achieves the critical goal identified in Section 4.1.3 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. For example, in an e-mail service, an incoming e-mail message generates a verb that records the contents of the message and the intended recipient's address. What happens to that message when the verb is actually executed is irrelevant from the verb's point of view: the message could be processed correctly, but it could also be dropped, corrupted, duplicated, filtered, or misdelivered. Likewise, when a user of an e-mail service deletes an e-mail message via an IMAP command, a verb is created that specifies the deletion intent and a name that uniquely identifies the target message, regardless of what (if any) messages actually get deleted.

As part of recording intent, verbs must be meaningful outside of the particular system context in which they are generated. Any recorded parameters cannot depend on the current state of the system for their interpretation, as there is no guarantee that such state will exist during replay. Because we are targeting our undo facility at existing services, we do not have the latitude to redesign protocols to guarantee that their operations take context-free parameters; as a result, the service-protocol-specific proxy must strip context dependencies from protocol operations as it turns them into verbs. 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.* [100].

In some cases, stripping context is easy. For example, any operation parameters that specify relative times must be converted to specify absolute times instead. Consider a `SEARCH` operation that takes a parameter specifying the time horizon of the search. Instead of recording that parameter as an interval relative to the present (*e.g.*, "the past day"), it should be recorded as an absolute time (*e.g.*, "the 24 hours preceding time *t*", where *t* is the time that the verb is generated).

A more challenging case of stripping context arises when protocol operations specify their targets via context-dependent names or identifiers. An example of this is evident in the IMAP e-mail retrieval protocol, where operations specify their target messages as sequence numbers referenced to the current contents of the currently-selected folder. Were verbs to simply record those sequence numbers, they could not be replayed in any situation where the contents of the folder had changed as a result of the undo operation. More generally, any protocol operations that refer to state using identifiers that are not globally unique and immutable are context-dependent and need to be stripped of that context before being encoded into verbs.

Our undo architecture provides a generic facility for helping handle the case of context-dependent names. It defines the notion of an **UndoID**, a globally- and temporally-unique, immutable, and time-invariant name that can be mapped to the current context-dependent name for the piece of state to which it refers. In some cases, UndoIDs can be stamped directly into the service's state itself using hidden fields: for example, in our undoable e-mail service described later in Chapter 6, we encode message UndoIDs into an unused message header field. For other cases, the undo architecture provides a temporally-aware database that maps UndoIDs into context-dependent names; the service-specific proxy updates the database each time name mappings change based on incoming protocol commands. The database is also synchronized with the undo manager so that names are invalidated and restored appropriately when the system is rolled back and forward in time. The database and its implementation are discussed in more detail below, in Section 5.5.

4.4.2 Sequencing verbs

At first blush it seems a simple task to ensure that the recorded log of verbs matches the sequence of operations performed by end users. The reality is more complicated, since the undo system is limited by only seeing end-user operations from its vantage point of a proxy outside of the service itself. If only one user interacted with the service at a time, there would be no problem. But if multiple users interact with the service simultaneously, the proxy may see those interactions arrive in a different order than they are eventually executed once they reach the service. And, because verbs are generated as they arrive at the proxy—not when their corresponding user interactions are actually executed by the service application—a naïve undo manager that simply recorded verbs in arrival order would not produce an accurate timeline log.

Sequencing interfaces. Our solution to this problem is to require that verbs define a set of interfaces, used by the undo manager, that expose enough information to help the undo manager determine and control the execution order of arriving verbs. These interfaces consist of three procedures that all verbs must define: a commutativity test, an independence test (which implies commutativity), and a preferred-ordering test. All three tests take another verb as an argument; the first tests if the two verbs produce the same externally-visible 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 [93].

Our definition of commutativity is slightly unorthodox, as it speaks only of the *externally-visible results* of the execution of two verbs. In other words, commutativity is defined from the point of view of an end user outside the service's sphere of undo, or, equivalently, at the protocol level. If the user sees the same output and the same system state (as visible via the protocol) regardless of the execution order of two verbs, then those two verbs are considered to commute, regardless of any internal service state differences not visible via the protocol.

More rigorously, let A and B be verbs, and let $Output(V)$ represent the external output of verb V. Likewise, let $Output(A|B)$ represent the external output of verb A given that verb B has already been executed. Let S be the represent the service state as is visible through the protocol. Let $Apply(S,V) \rightarrow S'$ represent the result of executing verb V on state S, that is, the protocol-visible state S' that results from V's execution on S. Then A and B commute if and only if both of the following hold:

- $Output(A|B) = Output(B|A)$
- $Apply(Apply(S,A),B) = Apply(Apply(S,B),A)$

The first criteria says that the external output of the system is identical regardless of the execution order of A and B, while the second says that the protocol-visible service state is identical regardless of the execution order of A and B.

Our definition of commutativity is necessitated by our black-box treatment of services, and shows one of the limitations of the black-box model. It is conceivable that two verbs commute externally but not internally, and that the internal state differences resulting from a different execution order could affect the external behavior of later verbs. However, a system that allowed such conditions would be undesirable from a user interface point of view, regardless of the presence or absence of an undo system—users should be able to assess the consequences of their actions from the visible state of the system, and should not be confused by hidden state. This same point motivates one of the properties we require of services for them to be made undoable, that they expose all pertinent state via their external protocols (see Chapter 8).

We can use the sequencing tests defined by verbs—commutativity, independence, and preferred ordering—to address the problem of creating an accurate timeline log. The undo manager simply uses the sequencing tests to *enforce* an ambiguity-free ordering of verbs before sending them to the service for execution. Using a scoreboard-like data structure and a scheduling discipline similar to the scoreboarding algorithms developed for dynamic instruction scheduling in early out-of-order CPUs [52], the undo manager stalls each incoming verb until all in-flight non-commuting/non-independent verbs have completed execution. Once all of its dependencies clear and the new verb commutes with and is independent of any remaining verbs, it is allowed to continue to the service to be executed. While a verb is stalled, other incoming verbs that do not depend on it are allowed to bypass it in the scoreboard if allowed by the verbs' preferred execution ordering.

The scoreboard-based sequencing approach produces a timeline that, when replayed serially, will result in an externally-visible system state consistent with that produced by the original execution. Furthermore, a similar algorithm and the same sequencing tests allow the timeline log to be replayed out of order with at least the same degree of parallelism as the original execution, helping to speed up the replay phase of undo. We will discuss our implementation of sequencing and parallelized replay in the next chapter (Section 5.4).

Performance. Sequencing verb execution does raise several issues, however. First is performance: sequencing verbs at the proxy can result in partial serialization of arriving user interactions, increasing latency and decreasing potential throughput. Our approach of serializing only non-commuting verbs reduces this penalty by executing as many arriving verbs as possible in parallel, serializing only when there is a non-commutative dependency between concurrently-arriving interactions. However, our commutativity tests are limited by the information available within the verbs, and that information is in turn constrained by the service's protocol. Protocols that do not expose enough information to accurately determine commutativity and independence at the verb level require conservative commutativity/independence tests, and can significantly impede performance by imposing too much serialization at the proxy. Put another way, there is significant pressure on the verb implementor to make the commutativity/independence tests as tight and accurate as possible, which requires a deep understanding of the protocol to understand where verbs truly do and do not depend on each other. Indeed, most of the difficult work in implementing the undoable e-mail prototype system described in Chapter 6 was in tightly specifying and implementing the commutativity/independence tests.

Long-running verbs. Another issue raised by the sequencing of verb execution is how to handle long-running verbs. For example, an e-mail service likely defines verbs for fetching and delivering messages. If the client (end user) is connected to the service via a slow dialup link, these verbs could take seconds, minutes, or even hours to run. Long-running verbs like these throw a wrench into the sequencing mechanism we have described, since they can potentially stall other dependent verbs for unacceptably long periods of time—as long as it takes those long-running verbs to execute.

There are several ways to handle long-running verbs, and our undo architecture supports all of them. The first, and most effective, is to split long-running verbs into multiple phases, ideally limiting reads and writes of shared state (and hence dependencies) to the shortest phase or phases. Sometimes this technique can be applied at the proxy without altering the protocol or service itself. For example, in the e-mail case described above, the fetch verb can be divided into two phases, with the proxy executing the fetch and queuing its resulting response as the first fast phase, and the proxy then dribbling out the response to the slow client as a second slow phase that is independent of any other executing verbs; an analogous technique works for mail delivery. If a service requires long-running interactions that cannot be divided at the proxy—for example, an interaction corresponding to a long-running computation—then either the service must be modified to split the operation, or one of the following less-effective techniques must be used.

A second approach to handling long-running verbs is to allocate more resources to the verb scheduler so that it can extract parallelism from farther ahead in the request stream, allowing the system to maintain acceptable throughput even if long-running verbs impose additional latency. This is an approach similar to what has been done in the historical battle to extract greater levels of instruction-level parallelism (ILP) in superscalar processor core design [52]. Analogously to adding more execution units and increasing reservation station sizes in a processor core, we can increase the size of our scoreboard queue of pending verbs, and add more execution threads to service it. While this approach can help overcome the effects of an occasional long-running verb with few dependents, it has diminishing returns as the degree of dependency and frequency of long-running verbs increase.

Finally, long-running verbs can be tackled by attempting to remove dependencies. Long-running verbs by themselves are not a problem; it is only when they block dependent verbs that they become an issue. Some dependencies are mandatory—for example, a verb that deletes a piece of state must depend on the verb that creates that state. But sometimes dependencies visible at the proxy level are **false**, and only appear to exist because of a lack of detailed information resulting from the abstraction of the service's protocol. For example, an incoming message in an e-mail service may be delivered to zero, one, or many mailboxes, depending on its address and routing tables inside the mail service. The proxy cannot determine which mailbox will receive the message, so it must assume a dependency between the delivery verb and other verbs that examine the contents of a particular mailbox; typically this dependency will be false.

Just as in other situations where false dependencies occur—like in the handling of addressing-induced memory dependencies during CPU instruction scheduling [52]—the apparent dependencies can often be removed by obtaining more information about the dependent operations. Specifically, the proxy can sometimes eliminate false dependencies between verbs by executing additional protocol commands that extract the information needed to identify the dependency as false; in our above e-mail example, the dependency could be resolved by asking the mail service to translate the address to a set of destination mailboxes. Fully resolving false dependencies may require extending the protocol or modifying the service, however.

Handling failed verbs. One last issue in sequencing verbs into a recorded timeline of history involves what to do when verb execution fails, either due to incorrect parameters supplied by the end user, or because the operation requested by the end user is inappropriate given the current system state. Our approach to failed verbs depends on whether the verb reports its failure back to the end user *synchronously* or *asynchronously*.

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 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 service 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 6.3.1). 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.

Discussion. The need to sequence verbs is the Achilles' heel of our black-box proxy-based approach to services. The extra mechanism needed to delay and scoreboard incoming verbs, the resulting performance impact, and the complexity of dealing with long-running verbs, could all be avoided if it were simply possible to determine the actual order in which verbs (or their corresponding operations) were executed by the service. One strategy might be to move the undo facility into the service itself and use the service's own log of operations as a timeline log. This is, in fact, the approach taken by one undo system, Network Appliance's SnapManager for Exchange [82]. Unfortunately, this approach gives up one of the most significant benefits of the black-box approach, and one crucial to our undo model: the ability to decouple the timeline log from the actual behavior of the service, and hence the ability to replay the log on a repaired system.

A more promising direction would be to modify the service to expose the actual execution ordering of interactions. This would require that interactions be uniquely identifiable, perhaps by stamping them with unique identifiers or by passing verbs directly to the service for execution. It would also require that the service be able to explicitly predetermine its own execution ordering, and hence would be easiest to implement for services that already support a notion of partially-serialized execution, for example those based on a transactional database or lock manager. Exploring this possibility is an interesting and promising avenue for future work.

4.5 Paradox Management

Having addressed the challenge of generating an accurate timeline log with the sequencing mechanisms and intent-based verbs described above, we can now turn to the fundamental question of managing paradoxes. As we discussed in Chapter 3, a Three-R's-based undo system for self-contained services must confront paradoxes head-on, as they are bound to occur given that services typically have large numbers of external end users who are unaware of the operator-driven undo process. We also laid out the requirement for a compensation-based approach to handling paradoxes, and further argued for a framework that can detect inconsistencies visible to human end users, reason about them relative to a service-specific consistency policy, and take appropriate compensating or explanatory action should any inconsistency be deemed a paradox by that policy.

We create this paradox-management framework by again requiring that verbs define a set of standard interfaces. In this case, the interfaces expose the service's paradox-detection policies and compensation framework to the undo manager. The interface set consists of three procedures that all verbs must define:

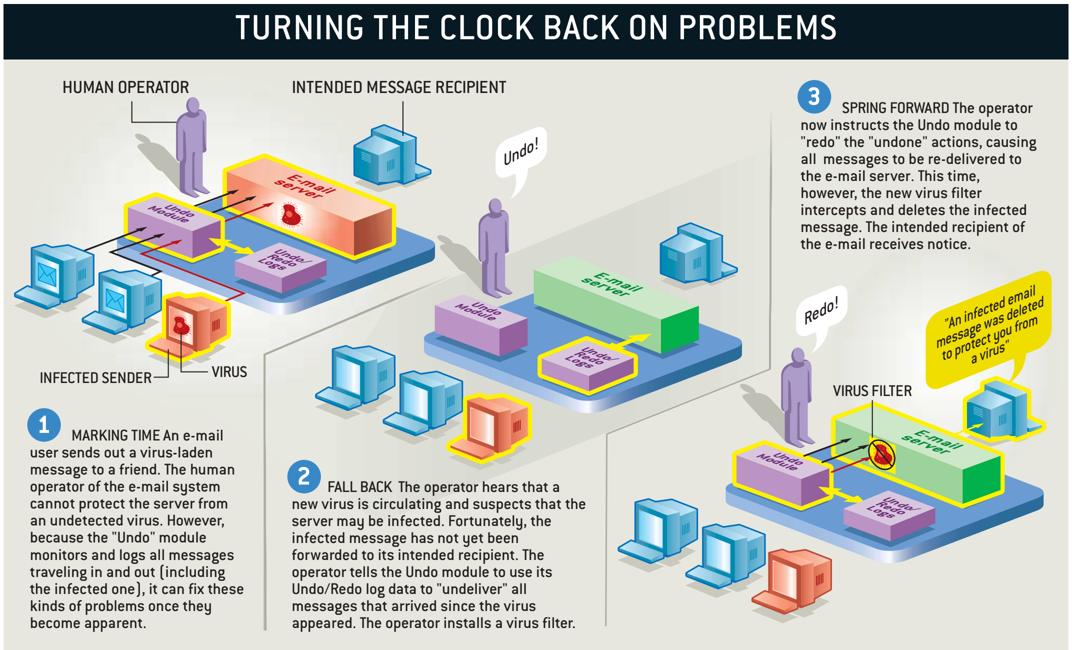
- a **consistency predicate** that compares a record of a verb's original external output to the output produced during replay
- a **compensation function** that is invoked with an encoded representation of the inconsistency implied by the failure of the consistency predicate
- a **squash function** used to alter verb execution when it participates in a chain of dependent inconsistent verbs.

4.5.1 Detecting paradoxes: per-verb consistency predicates

Recall that paradoxes (as defined in Section 3.3) occur only when the Three-R's undo cycle causes inconsistencies to appear in state that has *previously been viewed* by an external observer outside of the service's sphere of undo. To detect these paradox situations, we start by making the assumptions that service state can only be viewed externally via the proxied service protocol, and that verbs are defined for every protocol operation that **externalizes** (exposes) service state. These assumptions allow us to detect paradoxes on a per-verb basis: a paradox occurs any time a verb externalizes different output during Replay than it externalized when it was originally created. In other words, "consistency" from the point of view of detecting paradoxes is defined over a verb's **externalized output**—any response returned to the end user, plus the success/failure status of the verb's execution.

Note that verbs can behave differently during Replay than during their original execution and *not* cause paradoxes, as long as they either produce no external output or produce the same external output in both cases. This observation is a cornerstone of our approach to paradoxes, as it means that the service can behave differently during Replay, only causing paradoxes when the different behavior affects state that has already been seen by an actual, live end (and hence recorded in some verb's record of external output). By ignoring non-externalized behavioral differences during Replay, we increase tolerance for repairs and avoid burdening end users with compensations and explanations for inconsistencies that they would never notice.

For an illustrative example of our paradox detection policy, we turn again to the e-mail service domain, with a scenario depicted in Figure 4-3. Typically, mail delivery to an e-mail service results in no externalized output: the incoming message is silently accepted and later routed to one or more mailboxes. Imagine that our e-mail service has built-in virus filtering capabilities, but that the virus definition files are not up-to-date. If the e-mail service is now attacked by a new e-mail virus, the operator may wish to use undo to recover from that attack by retroactively updating the virus definition files to block the virus-laden messages. After rewinding the system to before the point when the virus attack began, and repairing the system by updating the virus definitions, the operator invokes replay. When the undo manager replays the verbs representing incoming e-mail, those delivery verbs corresponding to virus-laden mail will behave differently, as their messages will be filtered out rather than delivered to users' mailboxes. No paradoxes will be detected at this point, since it may well be that some users had not checked their e-mail and had never seen the originally-delivered virus-laden messages, and as such will be unaware of any inconsistency caused by the absence of those messages. Only when the undo manager encounters a verb that actually attempts to retrieve one of the virus-laden messages will a paradox be declared, since such a verb will produce different external output than it did



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

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

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

Illustration courtesy of Samuel Velasco. Used with permission.

Figure 4-3: Example of paradox detection and handling. This example shows how paradoxes might occur and be handled in an e-mail service. The paradox here is that the previously-virus-laden message disappears as a result of an undo cycle; the paradox is detected and compensated for by presenting an explanatory message to the end user (shown in the bubble on the right).

originally: originally, it returned the virus-laden message; during replay, it returns an error as the message does not exist.

To implement our externally-based per-verb paradox detection policy, each verb must define a consistency predicate that detects externally visible inconsistencies in that verb's output and failure status. To do this, verbs that externalize output record a copy (or hash) of their 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 external output to determine if they are acceptably consistent. The comparison test may be simply an equality test, but can be more sophisticated if the service allows relaxed external consistency.

This latter ability to define a relaxed consistency policy for external output on a verb-by-verb basis is a key factor in making a Three-R's undo system practical. It provides a way to encode any natural tolerance for inconsistency that a service's users might have, making that flexibility visible to the undo manager and thereby reducing the number of paradoxes generated as a result of repairs made during an undo cycle. For example, human users of an e-mail service may not care about the order in which messages are listed in a mailbox, since most mail clients impose their own sort order on top of that returned via the e-mail protocol; similarly, they may not see newly-delivered messages as a paradox, as e-mail messages arrive asynchronously anyway. If an undo cycle causes these sorts of inconsistencies, they should be ignored rather than treated as paradoxes, since end-users will accept them without complaint. The verb-specific consistency predicates provide a locus for encoding such policies; in our example case, the verb responsible for listing a mailbox's contents simply ignores ordering and newly-arrived messages when performing its comparison of external output from the original and replay executions of the verb.

4.5.2 Handling detected paradoxes: per-verb compensation methods

Once a paradox is detected by the failure of some verb's consistency predicate, the next step in the paradox management process is to perform some action to mitigate the paradox from the point of view of any end users who might observe it. These mitigating or compensating actions are coordinated by the undo manager, but their particular behaviors are inherently service-specific. Again, verbs provide a convenient locus for encoding service-specific compensating actions so that they may be used by the generic undo manager; the compensation methods define the second prong of our three-prong framework for injecting service-specific paradox management policies into the undo system.

Each verb that can produce a paradox must define an associated compensation method, which takes as an argument an encoded representation of the detected paradox and carries out whatever action is necessary to handle that paradox. Compensation may consist of ignoring the paradox, performing some action to mitigate it (such as creating a missing piece of state), or explaining the inconsistency to the user either directly or by embedding a message into the affected piece of service state, among other possibilities. Our architecture allows compensation only on a verb-by-verb basis, corresponding to our per-verb approach to paradox detection. We will not address the questions of how to define compensations and when compensations are even possible here, but will return to these issues below, in Section 4.6.2. In the meantime, we assume that every verb that can produce a paradox also defines a corresponding compensation for that paradox.

4.5.3 Handling propagating paradoxes: the squash interface

Our final concern involves handling user-induced dependencies between verbs that produce paradoxes and later verbs in the timeline. For example, in an e-mail service, a user might choose to delete a message based on reading its content. If during a later undo cycle that (externalized) content is changed, causing a paradox, 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, we have little choice but to take a conservative approach to handling such scenarios.

To achieve this conservative approach, we attempt to detect all possible verbs that might be affected by a paradox. We then give the service the option of modifying the execution of those verbs, for example to suppress their execution or to alter them to provide the end user with information about the long-term impact of the original paradox. These possibilities are achieved by requiring that verbs define a third key interface to complement the paradox detection and compensation interfaces. This interface, the **squash** procedure, is invoked on all verbs that depend on earlier paradox-producing verbs as the undo manager replays the timeline log. Here, dependencies are determined according to verb commutativity, tested using the same commutativity predicate defined for verb sequencing. If a verb is found to depend on an earlier paradox-producing verb, its squash method is invoked regardless of how (or if) the original paradox was compensated for.

Since verbs define their own squash procedures, squashing, like compensation, is service-specific, and the squash procedure represents the third component of our framework for embedding service-specific paradox management policies into the undo system. Typically, squashing will consist of cancelling the verb's original action, informing the user, and leaving it up to them to reconstruct their original intent, but can also involve more complex recovery behavior like performing a different set of operations. In the common case, only verbs that destroy or overwrite state will choose to alter their execution when squashed; others will define a null or no-op squash procedure. 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.

Note that because paradoxes are detected in terms of verb commutativity, and because our commutativity test is based only on externally-visible state, there may be situations where a later verb appears to (externally) commute with a paradox-producing verb, yet still fails or changes behavior as result of internal state changes not detected by the commutativity test. Rather than attempt to extend our commutativity tests to detect such situations, which would require a deeper understanding of the underlying application implementation than we can achieve in a protocol-based proxy, we simply allow later ostensibly-commuting verbs to execute against the current system state. If they fail or generate altered external output, they will be picked up as paradoxes in their own right and managed just as the original paradox-producing verb was.

4.6 Discussion

Managing undo-induced paradoxes is the central concept in our Three-R's approach to undo. In this section, we will delve deeper into the implications of a managed-paradox approach, examining its consequences for defining recovery guarantees, its implications for the applicability of Three-R's undo to different services, and its relationship to the treatment of paradox-like problems arising in other fields.

4.6.1 Recovery guarantees for undo

Traditional recovery-oriented systems, like transactional databases, have typically defined **recovery guarantees** (or invariants) based on the ability to return the system to a known, pre-existing state following a failure; in a transactional database system, the recovery guarantee typically states that the effects of all committed transactions will be preserved, whereas it will be as if any uncommitted transactions have never been executed [71]. Propagating paradoxes and our conservative approach to handling them raise the question of whether Three-R's undo can provide similar guarantees to either the end user or the operator about the state of the system after an undo cycle completes.

The most basic recovery guarantee that we can offer in a Three-R's system is **transparency**: that *in the absence of paradoxes*, external observers' view of the service system after the undo cycle will be consistent with their view of the service before the undo cycle was invoked. Here, consistency is defined by the service and expressed in the service's verbs' paradox-detection predicates. If paradoxes do occur, then the situation is more complicated. However, in this case we can turn to work in the database community for inspiration on how to proceed.

Korth *et al.* extend the transactional database recovery guarantee to a situation more like what we face with paradoxes in Three-R's undo by defining the notion of **soundness**. Soundness describes a recovery guarantee for situations where committed transactions can be later revoked and replaced with compensating transactions [64]; Korth's model of compensating for altered committed transactions mirrors our technique of compensating for paradox-producing verbs. In Korth's terminology, a system is sound if any transaction can be compensated for without affecting the history produced by later transactions; so-called *R*-soundness relaxes the model slightly by allowing compensations to alter later transactions in the history, as long as the altered transactions are equivalent to their original versions with respect to some non-equality relation *R*.

The recovery guarantee for a sound or *R*-sound system depends on the compensations that are defined. In the absence of knowledge about the compensations, the only guarantee that can be provided is one of **local compensation**: namely, that for every altered transaction, the appropriate compensating transaction will be applied, and that there will be no effects on later transactions in the history (or no effects that violate the relation *R* for *R*-soundness). From the end user's point of view, local compensation means that inconsistencies in state will be visible after the undo cycle, but that each inconsistency will either be explained or compensated for individually.

Translating back to the vocabulary of undo, a Three-R's system with paradoxes that do not propagate is sound; it is *R*-sound if paradoxes propagate but their propagation has effects that are insignificant relative to *R*. *R* is implicitly defined by the service's verbs' squash methods: if all the squash methods are null, even when paradoxes propagate, then the system is *R*-sound. If any verb's squash method is non-null, then the system is not *R*-sound, since a non-null squash method implies a situation where an altered verb (a paradox) results in effects on the later history that cannot be silently tolerated.

An undoable system with no propagating paradoxes (or verbs with all-null squash routines) therefore follows the (*R*-)sound recovery guarantee of local compensation. But propagating paradoxes are a fact of life in our Three-R's undo facility, however, and it is not always the case that the service has a sufficiently relaxed consistency model to subsume the visible effects of propagation, as in *R*-soundness. This situation arises because verbs can represent elements of longer streams of user interaction, where the result of one interaction informs later ones, and the information needed to fully compensate for a change in an early interaction is available only in the user's mind. Indeed, while it may be *possible* to re-execute a later verb with the same results as during the original execution (satisfying *R*-soundness), it may not be *desirable* to do so given the earlier paradoxes that have occurred. To discuss a recovery guarantee for undo, we therefore need a new version of soundness that speaks of an end user's intent in carrying out an entire sequence of operations—that can understand the intent of a user who is reading e-mail and deleting spam, for example, as opposed to just seeing a sequence of mail fetches and deletes. Such an **intent-soundness** property would guarantee that compensations preserve the higher level user intent expressed by operation sequences, not just the semantics of individual operations, and would make an ideal recovery guarantee for an undo system.

However, the information needed to reason about this level of soundness is simply not available at the service protocol level, and typically is not available anywhere in the system except in users' minds. Thus, we cannot even attempt to provide an intent-soundness guarantee in our generic undo architecture. Instead, we leverage the fact that we are targeting services with human end users: we trust those users to be able to reason about their own intent and to carry out the necessary higher-level compensating steps to restore their intent should a paradox disrupt a sequence of operations spanning more than one verb.

Thus, even with propagating paradoxes, we keep our compensations local, and only extend the local compensation recovery guarantee with a promise that squashing will be performed conservatively on any verbs dependent on the paradox-producing verb. What this guarantee means from the end user's perspective is unclear; it depends entirely on how the service defines its verbs' squash methods. Ideally, the service will use squashing to block the execution only of verbs that destroy state, and in other cases will simply add more information to the explanation presented to the user for the original paradox. In this case, the offered recovery guarantee is the local compensation guarantee plus the promise that any changes dependent on an individual state inconsistency will be explained along with that inconsistency, allowing the user to manually re-establish their original intent taking the inconsistency (or paradox) into account. We call this the **lose-no-state** recovery guarantee, since it promises to squash any verbs that might undesirably destroy state following a paradox, and it is the type of recovery guarantee we provide for the undoable e-mail store service described in Chapter 6.

To summarize, the recovery guarantee offered by a Three-R's undo system is a only local one, and is inherently entangled with the particular undoable service's semantics. The generic undo architecture can guarantee that:

- every verb recorded during normal execution will be re-executed during replay
- after each such re-execution the verb's paradox-detection predicates will be invoked
- if a paradox is detected, the verb's compensation methods will be invoked, providing local compensation for the individual paradox
- the squash methods for later verbs that do not commute with the paradox-producing verb will be invoked.

It is possible to achieve transparency, soundness, *R*-soundness, intent-soundness, lose-no-state, or practically any other recovery guarantee in a Three-R's system following the architecture we have laid out. But doing so is possible only with consideration of the service-specific paradox policies that are injected into the generic undo system via verbs and their three-prong framework of paradox detection predicates, compensation methods, and commutativity tests and squash methods.

4.6.2 Potential applicability of Three-R's undo

By this point we have established that the Three-R's undo architecture can capture a service's policies for detection and compensation of undo-related paradoxes, and can provide a corresponding recovery guarantee. However, our case is built upon the assumption that it is possible to define the needed paradox-management policies and guarantees for real-world services. Beginning in this section and continuing through the following two chapters, we will examine that assumption through analysis and case studies, and will arrive at a deeper understanding of the cases where paradox management is or is not applicable. Since Three-R's undo rests firmly upon the principle of managed paradoxes, our analysis will consequently shed light on the question of when and where it is possible to apply the techniques of Three-R's undo.

Fundamentally, successful paradox management requires a degree of flexibility in the way a service is viewed by its external users. To manage a paradox, it must be possible, meaningful, and cost-effective to *retroactively alter* incorrect information that has been exposed from the service to the outside world—information that has already flowed across the boundary defined by the service's sphere of undo.[1] The compensation routines defined by verbs encode the policy and mechanism by which these retroactive alterations are made.

[1] By only looking at boundary crossings, we are following our model that paradoxes only occur when information is passed between different "bubbles" of time. If the service wants to change the definition of paradoxes to include undo-induced changes to state that are not exposed across the boundary of a sphere of undo, our undo architecture cannot be used as is. Instead, it must be modified so that every verb that alters unexposed state is treated as if it immediately exposes that state, creating pseudo boundary crossings that can be treated by our paradox-management architecture.

There are two ways in which a service can tolerate retroactive alteration of information that has crossed its sphere-of-undo boundary. The first is through relaxed consistency: if the service application's external consistency model can absorb the retroactive alterations, then compensations are unneeded and paradox management is trivial. We saw an example of this case earlier, in the example of an e-mail service where retroactive changes to message ordering and new message arrivals could be ignored by the paradox management framework. We will see another example later in the auction system described in Chapter 7, where retroactive changes to bids other than the final or winning bid are absorbed by the external consistency model and can be ignored from the point of view of paradox management.

An extreme case where the consistency model absorbs paradoxes is in services that provide such weak guarantees to their end users that they can forgo replay entirely, absorbing the paradoxes resulting from the loss of user interactions as acceptable under the service's weak consistency guarantees. A web search service is an example of such a service: search results are rarely guaranteed to be completely accurate or complete, and hence retroactively-detected errors in delivered search results can simply be ignored. A more subtle example might be a free e-mail service, whose operators might prefer to discard recent incoming e-mail and user mailbox changes rather than dedicate the resources needed to absorb the overhead of replay with compensations. Cases like this one are likely to be rare, however, and may only exist in niche markets where the service offers such unique value and the cost of switching to another provider is so high that its users are willing to forgive occasional loss of data.

While absorbing paradoxes through a relaxed consistency model is ideal, most practical services will have at least some boundary-crossing interactions for which retroactive changes cannot be so easily masked. In all such cases, paradox-free undo can be provided up until the point where one of these unmaskable, altered boundary-crossing interactions occurs; services in which unmaskable boundary-crossing interactions are asynchronous and infrequent may be able to offer a reasonable window of undo without further paradox management mechanism.

For all other cases, however, or when undo is desired even for infrequent unmaskable boundary crossings, compensation is necessary and must be encoded into the service's verbs. This requirement leads to the key criteria for determining if a service can be made undoable: it must be possible to define all the needed compensations, one for each possible way in which each external interaction can be retroactively altered.

Services where compensation is not possible cannot be made undoable using the basic Three-R's undo approach we have described. As an extreme example, a service that controls a missile-launch facility is unlikely to be undoable, since no reasonable compensation exists for the effects of an improperly-launched missile. A more mundane service that is not easily made undoable is a block-oriented storage service. The users of such a service are nearly always other computer systems that run their own applications and services in their own right; these systems do not have the human end user's capacity for reasoning about inconsistency or for understanding compensatory explanations. Thus, without propagating undo requests from the service to its clients (which would violate our self-contained service assumption), undo cannot be provided for such a service.

Our verb-based paradox management framework requires that it be possible to perform any needed compensations using only *local* information—the information available in the input parameters and external output associated with the verb performing the compensation. Local information can be compiled over time to simulate some forms of more global information, such as knowledge of later verbs in the log, but ultimately it must be possible to perform compensations with only the information available in the verb log. Since the verb log tends to contain all key information about external interactions with a service, this requirement is not excessively restrictive. We will see an example of how local information can be compiled to satisfy an apparent need for global knowledge later, in our treatment of auction services in Chapter 7.

Probably the most central requirement for compensations, however, is that they be meaningful and acceptable to the end-users who experience them. These latter criteria are inherently service-specific and must be evaluated by examining both a service's explicit guarantees and the larger social context in which it is used. For example, services that already handle imprecise or tentative information, or that serve as front ends to complex error-prone processes, are likely to already have built-in mechanisms for compensating for errors or after-the-fact changes. Consider how banks can compensate for erroneous deposits and withdrawals by reversing them (and refunding or charging fees); how merchants can compensate for erroneously-handled orders by refunding them or absorbing their expense as part of the cost of doing business; how administrative assistants can reschedule meetings when one of the participants or the meeting facility itself discovers an unexpected conflict. The electronic versions of these same services (online banking, e-shopping, and group calendaring, respectively) must all include the same compensation mechanisms as their traditional counterparts, regardless of whether or not they provide undo. Thus, when providing undo, these mechanisms can be retasked to compensate for undo-induced paradoxes; since they are already part of the service's contract with its users, they are by definition acceptable and meaningful.

In summary, the question of the applicability of the Three-R's undo approach boils down to whether service applications can tolerate paradoxes. A few services conveniently offer such loose consistency models that managing paradoxes is trivial—as in the earlier web search example. Other services, like e-shopping and group calendaring, grew out of manual, human-driven processes and thus already have integral mechanisms for coping with the inherent inconsistency in complex real-world interactions; these mechanisms can be leveraged to compensate for undo-induced paradoxes. Yet others, like block storage, target other computers as end users and cannot tolerate paradoxes at all, at least, not without an extended undo model that supports distributed undo via multiple interacting spheres of undo. We expect most services with human end users to fall into the middle category, those with existing compensation mechanisms, since those services typically result from an attempt to automate or enhance existing business or social practices.

4.6.3 Comparison to existing approaches

Finally, we take a step back from the details of the Three-R's undo architecture in order to set it, and its paradox management approaches, into the context of existing work on inconsistency management. We begin by looking at replica systems, then extend our consideration to systems that handle situations analogous to our propagating paradoxes.

Paradoxes vs. inconsistencies in replica systems. 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 [126], IceCube [60], and Coda [109]. The similarity is not surprising, since 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. In both systems, inconsistencies can arise and must be managed. In Three-R's undo, inconsistencies (paradoxes) arise during replay when we try to execute the recorded timeline of verbs in the context of a version of system state that is different (due to repairs) from that when the verbs were originally logged. Similarly, in replicated storage systems, inconsistencies can arise during the process of applying a set of user updates to a stored object when the stored object has been modified by intervening updates from another user. If the updates are incompatible and cannot be reconciled, one of the users sees a final state that is inconsistent with their view of their updates [107].

Most replica systems manage inconsistencies by protecting each operation with some sort of guard or precondition check that examines the current state of the system and determines if the update can be safely executed. The Bayou system exemplifies this approach, coupling each verb with preconditions and application-defined merge procedures that provide compensation for the inconsistency that would result if the precondition fails [126]. At first, we believed the precondition-based approach used in systems like Bayou was the right approach to apply to undo; in fact, in earlier work we proposed that each verb be associated with preconditions that determined if the verb would re-execute with the same externally-visible results as its initial execution [17]. However, bogged down with the complexity of essentially predicting the execution of every verb, a near impossibility given the high semantic level of our verbs and the fact that our fault model makes no guarantees about the service's correctness, we eventually realized that Undo has unique characteristics that allowed us to vastly simplify our verb-based approach to external consistency.

The key simplifying characteristic 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. Paradoxes only arise when inconsistencies affect previously-externalized state. This insight motivated the choice of only testing for consistency of external output, rather than using Bayou-like preconditions to test every verb for inconsistency before executing it.

Furthermore, we use *post-execution* consistency checks, rather than the pre-execution or precondition checks used by replica systems. The reason for this is that, in Three-R's 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.

Despite our different and simplified approach to detecting paradoxes, 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 service's sphere of undo.

Propagating paradoxes. The problem of propagating paradoxes is analogous to a problem that arises in the selective undo models mentioned at the start of Chapter 2. In these systems, a user can choose to undo (or, equivalently, modify) a single operation well back in the system's history, without undoing later operations in the history. In such situations, conflicts can occur when the undo or alteration of an old command invalidates the later commands in the history. In our case, similar conflicts can occur when a paradox calls into question the validity of later verbs in the timeline.

The bulk of the work on conflict management in selective undo systems is based on commutativity, much like our approach of using verb commutativity to detect propagating paradoxes. For example, Prakash and Knister, working in the context of collaborative editing systems, use operation commutativity to determine if selective undo can be performed without conflicts [92]. In their approach, selective undo of a past command is allowed if and only if that command commutes with all later commands in the active history—in essence, the undone command is pushed through the history until it becomes the most recent command, at which point it is undone as usual. If the command encounters a non-commutative command along the way, the selective undo process fails. Pehlivan and Holyer extend this model in an interactive system by allowing the user to choose to roll back all dependent operations following the command to be undone [90]; our approach of squashing dependent operations is a generalization of this approach that allows the service to specify which, if any, dependent operations are suppressed.

Berlage investigates an alterative approach that replaces commutativity tests with predicate-based preconditions: in his approach, developed in the context of an editor for graphical objects, the application developer augments command objects with predicates that test the current system state to determine if the command can be undone or redone [10]. Berlage's approach can be seen as a generalization of the commutativity-based approach, allowing tests against system state as well as history entries, and is similar to the precondition-based approach used by Bayou to validate operations during reconciliation.

In the database domain, Liu *et al.* take a similar approach to the commutativity-based groupware approaches [69]. They define a history-rewriting approach that undoes the effects of maliciously-introduced transactions by collecting them at the end of the history, using commutativity rules to move them across regular transactions, then truncating the end of history. Where commutativity prevents moving a malicious transaction all the way to the end of the history, Liu's approach is to simply discard all conflicting non-commuting transactions, assuming them to be tainted by the original malicious transaction.

Our approach to detecting propagating paradoxes in a Three-R's undo system can be seen as a hybrid of the commutativity-based approaches and Berlage's state-based approach. Our undo architecture uses verb commutativity as the indicator of whether a paradox-generating verb affects later verbs in the history, but because our commutativity test is based only on external output, we also adopt aspects of Berlage's approach by always checking later verbs' failure status as a way to catch the rare cases where verbs appear to commute externally but, due to invisible internal dependencies, actually do not.

We handle detected propagating paradoxes with an approach similar to Pehlivan and Holyer's, squashing the verbs to which a paradox verb has propagated. Because our implementation of squashing typically does not just discard dependent verbs, it is a more flexible approach than either Pehlivan and Holyer's technique or Liu's history-rewriting scheme. Services can choose to execute a squashed verb anyway or to modify its execution, depending on the particular verb's operation; if they do choose to squash a verb completely, they can still define compensating actions that explain to the user why the operation was squashed.

4.7 Summary

In this chapter we have defined an architecture for implementing the Three-R's undo model defined in Chapter 2. Our architecture wraps a sphere of undo around an existing service application by interposing a proxy into the service's communications with its external users. The proxy maps user interactions into verbs, which serve both as replayable encapsulations of user intent and as a locus for defining service-specific policies for detecting and managing paradoxes.

Because paradox management is so central to the Three-R's undo approach, it directly impacts the potential applicability of undo as well as the recovery guarantees that can be provided across an undo cycle. We saw how applications need either an inherent tolerance for retroactive changes to exposed data, or existing mechanisms for compensating for such changes, to be suitable for Three-R's undo. Luckily, most service applications targeted at human end-users meet one or both of these requirements.

Chapter 5

A Java-based Implementation of the Three-R's Undo Architecture

> "Backward, turn backward, O Time, in your flight."
>
> — *Elizabeth Akers Allen*

To gain experience with the practical issues involved in providing Three-R's-style undo, and to validate the feasibility of the architectural approach described in the previous chapter, we have constructed a Java-based implementation of a service-neutral Three-R's undo system. The implementation leverages and extends several existing technologies to provide a robust, service-neutral platform for building undoable services

The basic organization of our implementation follows that of Figure 4-1, with the undo manager at its focal point. We will only discuss the service-neutral parts of the architecture in this chapter—verbs, timeline log, rewindable storage, undo manager, and control user interface. To these components, a service must add its own service-specific verb implementations as well as a service-specific proxy to complete the undo system. We will give examples of these components in Chapters 6 and 7.

For many of its components, our implementation leverages existing, well-proven subsystems—Network Appliance filers, BerkeleyDB databases, scoreboarding techniques, and predicate-based inconsistency detection, amongst others—and synthesizes them to provide Three-R's undo. The choice to reuse existing technology where possible proved to be a good one, as it helped reduce complexity, increase trustworthiness, and greatly simplified the implementation burden.

Our prototype is written in Java; we chose Java for its advantages in rapid prototyping and for the availability of features like type safety, garbage collection, and exceptions, all of which we hoped would improve the dependability of our implementation. The service-independent core of the prototype comprises about 16,000 lines of code (about 4,600 semicolons). To further improve the depend-

System	Lines of Code				
	in try/catch block	in catch block	reachable from try/catch block	reachable from catch block	total
Undo	28%	8%	63%	44%	25K
Osage	7%	2%	28%	11%	20K
Quartz	12%	5%	31%	6%	27K
Compiere	9%	2%	36%	30%	250K

Table 5-1: Analysis of recovery-oriented coding style. This table presents the results of a static code analysis performed by Westley Weimer, showing how the structure of our undo system's source code compares to that of three other open source Java programs: Osage, an object-relational database mapping layer [87]; Quartz, an enterprise job scheduler [95]; and Compiere, an enterprise ERP/CRM product [23]. The results show that the undo system has a significantly greater percentage of its lines of code in and reachable from recovery constructs than the other systems, highlighting its recovery-oriented coding style. Note that the version of the undo system used for this analysis included the verbs and proxy implementation for the e-mail service described in Chapter 6, resulting in an overall larger code size.

ability of our implementation, we adopted a recovery-oriented coding style that demanded scrupulous attention to exception handling, multiple retry of all exceptions corresponding to potentially-transient failures, recovery procedures to prevent corruption of the service's state should a time-travel procedure fail midway, and full explanation to the operator of all truly unsalvageable failure situations. An analysis of a slightly earlier version of our undo manager code by Westley Weimer, summarized in Table 5-1, shows the impact of such a recovery-oriented style: compared to comparable-size open source Java programs from sourceforge.net, our undo manager code has significantly more code resources dedicated to recovery and failure management.

5.1 Verbs

Verbs are by far the most important element in our Three-R's undo architecture. They also form the glue that binds our implementation together, acting as the fundamental data structure that connects the service-specific proxy, generic undo manager, timeline log, and GUI together.

To implement verbs, the undo system defines a Java interface, `Verb`, that specifies the API that all verbs must provide; service-specific code can then define actual verb classes that implement the standard interface, one class for each end-user interaction that the service supports. The key methods of the `Verb` interface are listed in Figure 5-1. Note that the core of the interface is a straightforward mapping of the sequencing and paradox management routines described in Sections 4.4 and 4.5. This core is supplemented by routines that supply the undo manager with more information about the properties of the verb:

- `isAsynchronous`: specifies whether the verb executes synchronously or asynchronously with respect to an end-user interaction

- `getFailureReplayPolicy`: specifies whether failed instances of the verb should be recorded and later replayed, or simply ignored

- `checkVerbComplete`: indicates whether the verb's data structure is completely populated and ready to record to the timeline log

```
interface Verb {
   // Sequencing APIs
   int      checkCommutativity(Verb v2, boolean quick);
   int      checkExecIndependence(Verb v2, boolean quick);
   int      checkExecReorderOK(Verb v2, boolean quick);
   int      getPreferredOrder(Verb v2);

   // Execution APIs
   int      execute(boolean readOnly, Date virtDate,
                     CxnState cxn, ResponseInfo respInfo);
   int      checkAsyncCompletion();

   // Paradox management APIs
   ConsistencyViolation checkConsistency(Tag originalTag,
                    boolean statusExternalized);
   boolean  handleInconsistency(ConsistencyViolation cv);
   boolean  squash(Vector otherVerbs);

   // Policy APIs
   boolean  isAsynchronous();
   int      getFailureReplayPolicy();

   // Information APIs
   User     getUser();
   void     setUser(User u);
   Tag      getTag();
   void     setTag(Tag tag);
   boolean  checkVerbComplete();
   void     setReplayDate(Date d);
   Date     getReplayDate();
   long     getReplayTimeMillis();
   void     updateReplayTimeEstimate(long millis);
}
```

Figure 5-1: Verb interface. The code snippet above illustrates the key methods of the `Verb` interface, including a straightforward mapping of the sequencing and paradox management routines described in Sections 4.4 and 4.5, as well as additional helper functions to supply the undo manager with more information about the verb. The interfaces have been simplified slightly for presentation, and thrown exceptions have been elided.

- `get/setReplayDate`: provide a record of when the verb was replayed
- `getReplayTimeMillis/updateReplayTimeEstimate`: retrieves and updates the verb's estimate of how long it will take to replay.

The `Verb` interface also defines routines to manage the verb's execution. Here, we had a choice to make: whether to embed the code to execute a verb directly in the `Verb` class itself, or whether to keep that code as part of the service proxy, simply handing verbs back to the proxy when they needed to be executed. We decided to take the former approach, requiring that verbs define an `execute` method that could be invoked by the undo manager. This approach has the advantages of streamlining the proxy and centralizing the code related to each verb into one (Verb) class, and makes it easier to re-sequence and queue verbs in the undo manager. It also pushes the some of the complexity of managing open connections to the undo manager where it only need be implemented once; we discuss this issue below in Section 5.4. Finally, the Verb interface includes a checkAsyncCompletion method that reports if an asynchronous verb has completed its execution, which the undo manager needs to know in order to clear entries in its scoreboard of running verbs.

Other than the required routines in the Verb interface, which are used by the undo manager, all state and methods defined by particular verb classes are service-specific. Although the undo manager sees this additional state as opaque, it does require that certain parts of it follow a pre-defined structure. To that end, all Verbs must contain a Tag, a container data structure that wraps all the information needed to execute the verb and to check its external consistency, as well as 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. The tag has three components:

1. a boolean value indicating whether the verb's original execution succeeded or failed; if the verb is asynchronous, the tag also includes a timestamp indicating when the execution status becomes externally visible
2. an InputParameters object, which is an opaque, application-defined object that contains whatever parameters are required to execute the verb
3. an ObservedOutput object, also an opaque object, that records enough information to characterize any externalized output produced by the verb. Typically, the ObservedOutput structure will contain hashes or digests of the output rather than its complete contents.

Verbs and Tags are declared to be serializable so that they can be easily transferred to and from disk by the undo manager.

5.2 Time and the timeline log

The history of the undo system is maintained in the timeline log, a linearized record of all end-user interactions with the service system. The log is stored in a simple BerkeleyDB recno-style database, leveraging BerkeleyDB's built-in facilities to automatically provide caching, persistence, efficient record access, and recoverability for the log [113].

Each log record is assigned a sequential, unique, and immutable **log sequence number** (**LSN**). The LSN is the fundamental internal representation of time for the undo system, 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 tracks the system's **virtual time**—the LSN that corresponds to the version of state currently visible in the wrapped application service. This LSN is typically the last log record replayed, or the end of the log if the system is up-to-date. During normal operation, the virtual time matches real time. When the system is rewound during an undo cycle, the virtual time is set to the LSN of the log record immediately preceding the time point to which the system is rewound. When the system is replaying, the virtual time increments as each verb is read from the log and re-executed.

Any operation that alters the virtual time (other than by logging a verb) writes a special control record to the timeline log, so that, should the undo manager crash unexpectedly, it can reconstruct the appropriate virtual time by scanning the log. Control records are informational only, are always stamped with the current real time and appended to the end of the log (regardless of the virtual time), and are never replayed.

To keep the timeline log from growing without bound, it is synchronized with the rewindable storage layer. If the timeline log grows beyond its available space, the undo manager will truncate the beginning of the log and will inform the rewindable storage layer to discard its record of history preceding the truncation point. With a checkpoint-based rewindable storage layer like the one we use (described in the next section), the undo manager always truncates the timeline log at a point corresponding to a valid storage checkpoint. This "history truncation" can also happen in reverse: if the rewindable storage layer discards a portion of history (for example by discarding its oldest checkpoint), it informs the undo manager, which can then discard the corresponding portion of the timeline log.

5.3 Rewindable storage

At the base of the undo system is the rewindable storage layer, which provides stable storage for the application service's hard state as well as the ability to physically restore previous versions of that state. To support our Three-R's undo model, the rewindable storage layer needs to provide a **reversible rewind**, which makes it possible to cancel an undo cycle cleanly by undoing the rewind and restoring a snapshot of the system taken just prior to rewinding it. Unfortunately, we could find no storage layer offering both efficient and reversible rewind, 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 [53]. Out of the box, the filer provides efficient rewind capabilities, but suffers from a 31-snapshot limit and, more seriously, cannot provide reversibility: rewinding the filer to an old snapshot annihilates any later ones, blocking the ability to undo the rewind or restore the pre-rewind state. Furthermore, the filer does not provide a control interface that could be easily integrated into our Java-based undo system.

We began addressing these limitations by constructing a Java wrapper that hides the telnet/console-based command-line interface to the filer's snapshot management tools. The wrapper interfaces with the filer, tracks the filer's active snapshots, and provides the rest of the undo system with an API for creating, deleting, restoring, and listing snapshots.

5.3.1 Reversible rewind

To address the lack of reversible rewind, 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 reversible rewind with the following steps:

1. take a recovery snapshot

2. delete all data on the storage volume

3. scan the old snapshot to construct a list of directories and files
4. recreate every directory in the old snapshot on the now-blank storage volume
5. for each file in the old snapshot, use the filer's forward-restore capability to restore it (in constant time, independent of file size).

To reverse the rewind, we need merely restore the recovery snapshot, which takes the system to the state it was in before the old snapshot was made current.

While the copy-forward workaround makes reversible rewind possible, it is significantly slower than the filer's native snapshot restore facility. For example, on a system with about 10,500 files and 10,000 directories comprising about 2.2GB of data, it takes on average 590 seconds, or almost 10 minutes, to reversibly rewind the system to an old snapshot (average of three runs, standard deviation <1%). The bulk of this time is spent copying files from the old snapshot into the active system, and so as expected, our experiments show that the rewind time scales roughly linearly with the number of files on the system. In contrast, using the Network Appliance filer's built-in snapshot restore capabilities, an old snapshot can be (non-reversibly) restored in a constant 8 seconds on average (10% standard deviation over 12 runs), independent of the number of files and directories. This is the order of magnitude rewind time that would be achievable in practice, given the proper interfaces into the filer to support reversible rewind.

5.3.2 Limited number of snapshots

To address the limited number of snapshots provided by the Network Appliance filer, we designed the undo manager to implement rewind by first restoring 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.

5.3.3 Checkpoint management

The filer and associated wrapper class described above provide the primitive operations needed to manage the rewindable storage layer: creating, listing, deleting, and reversibly rewinding to snapshots. But the undo manager needs additional functionality from the storage layer, namely the ability to periodically take snapshots as the system operates, to age out and eventually discard old snapshots, and to assign additional semantics to certain snapshots depending on the circumstances when they are created.

To meet these needs, we define a `CheckpointManager` module to interface with the filer's wrapper and provide the undo manager with a higher-level interface to storage. This module deals in **checkpoints**, which map to snapshots at the filer level but are given different terminology to indicate that they have more associated semantics than an filer snapshot.

The primary role of the checkpoint manager is to periodically take automatic checkpoints as the service operates, ensuring that there is a continuous sequence of checkpoints spanning the time covered in the system's timeline log. Having good coverage of the timeline log with checkpoints is important, since checkpoints define the points to which the system can be most efficiently rewound. To strike a balance between dense checkpoint coverage of the most recent past (the most likely target of a rewind) and long-term checkpoint coverage of the older parts of the timeline log, the checkpoint manager takes its automatic checkpoints at multiple configurable granularities (*e.g.*, every 10 minutes, every hour, every day, every week), aging out old ones to keep a specified maximum number of checkpoints available at each granularity. With a limit of 31 available snapshots in the Network Appliance filer, our default limits provide a checkpoint for every 10 minutes of the most recent hour, a checkpoint for each of the 13 hours preceding that, a checkpoint for each of the 5 days preceding the current day, and a checkpoint for each of the three weeks preceding the current week. In other words, the checkpoints span up to a month of time, with up to 20 snapshots concentrated in the past 24 hours. As each granularity fills, the oldest checkpoint at that granularity is discarded. When the oldest checkpoint in the system is aged out, the past history between that checkpoint and the next-most-recent becomes unreachable. When that happens, the undo manager is notified so that it can truncate the portion of the timeline log corresponding to the discarded history.

Automatic checkpoints are only taken when the system is operating normally, and are disabled during an undo cycle. Furthermore, to avoid wasting checkpoints on state that has not changed, automatic checkpoints are triggered only when verbs are recorded. Thus no automatic checkpoints are taken when the system is idle.

Besides automatic checkpoints, the checkpoint manager defines two special types of checkpoints. First are **reserved** checkpoints, which are managed separately from the automatic checkpoints described above and must be created and released explicitly. Creating a reserved checkpoint never causes another checkpoint to be aged out or discarded (requests are rejected if no reserved checkpoints are available), and so reserved checkpoints are used when the undo manager needs a checkpoint internally without disturbing any existing checkpoints. For example, the recovery checkpoint taken by the undo manager just before beginning the rewind process must be a reserved checkpoint, since if it were a regular checkpoint, the act of creating it could cause the target checkpoint of the rewind operation to be aged out and discarded. The checkpoint manager sets aside two of the filer's snapshots for the exclusive use of reserved checkpoints.

The second special type of checkpoint defined by the undo manager is a **permanent** checkpoint. Permanent checkpoints are also taken explicitly, but enter the aging rotation along with regular automatic checkpoints. However, unlike regular checkpoints, they can only be discarded once they become the oldest active checkpoint in the system. Permanent checkpoints are taken when an undo cycle commits, to ensure that the repairs and other changes made to the system during the undo cycle are captured and preserved. Were these post-undo checkpoints non-permanent, then it could become impossible to rewind the system to a point immediately following a prior undo cycle, as the repairs from that cycle would be lost along with their post-undo checkpoint.

One final responsibility of the checkpoint manager is to coordinate the taking of checkpoints with the proxy and undo manager. To ensure that checkpoints are consistent with their logged record in the timeline log, the checkpoint manager instructs the proxy to stop accepting connections and requests that the undo manager stall the scoreboard (allowing running verbs to complete but blocking execution of new verbs) before it triggers a snapshot at the rewindable storage level. Once the snapshot has been taken and logged in the timeline log, the scoreboard and proxy are re-enabled, allowing verb execution to resume. This process may be noticeable to end users as a “blip” in service availability; if desired, more sophisticated techniques based on determining the exact commit time of running verbs relative to the checkpoint could be used to avoid the brief interruption of service during checkpoints.

5.4 The undo manager

At the heart of the undo system is the undo manager, the component that mediates verb execution during normal operation and manages the system timeline by coordinating Rewind, Repair, and Replay. It ties together all the components described above, and is used by both the service-specific proxy and the system operator (via a user interface). In our implementation, the undo manager is a Java object that provides a set of APIs for each of these two users; we will discuss each of these API sets in turn.

5.4.1 Mediating verb execution: proxy APIs

The first set of APIs provided by the undo manager is used by the service-specific proxy to pass in verbs for sequencing and execution, and consists of two methods with the following signatures:

```
long recordOp(Verb v)

void enqueueInteraction(Verb v, boolean doLog,
                        CxnState cs, ResponseInfo ri,
                        InteractionNotifyer notifyer)
      throws MalformedVerbException
```

The first method simply takes the supplied verb and writes it to the log, without executing it or sequencing it, and returns the allocated LSN for the verb's record. This interface is provided should a service want to do its own sequencing and verb execution, but is not the recommended interface to use.

The second interface is the one that typically will be used by services. This interface takes a verb that has not been executed and inserts it into the undo manager's scoreboard of pending verbs, where it will later be executed and logged as described in the next section. The verb is expected to have a tag defined and preinitialized with the verb's input parameters. The method also takes references to a `CxnState` structure, which is a service-defined class that encapsulates an existing connection to the actual service, allowing the verb to reuse that connection during its eventual execution; and similarly a `ResponseInfo` structure, which is a service-defined class that holds a reference to the existing connection between the proxy and the end user, allowing the verb to send any responses it generates back to that user. These latter two structures are saved by the undo manager and handed back to the verb when its `execute` method is eventually invoked.

Finally, since the `enqueueInteraction` call returns immediately after enqueuing the verb for execution, it allows the service proxy to pass in a reference to an `InteractionNotifyer` structure, which is an asynchronous callback closure that gets invoked once the verb has completed execution, passing back the results of the verb's execution. The `InteractionNotifyer` callback allows a service proxy to treat verb execution as synchronous and blocking if it so desires; typically, verbs will execute asynchronously as sequenced by the undo manager.

Along with using the two methods described above, which are sufficient to allow a service-specific proxy to hand off verbs to the undo manager for execution, the undo manager defines two callbacks that a service-specific proxy must implement. These callbacks, `quiesce` and `unquiesce`, are used to inform the proxy when a checkpoint is being taken, so that the proxy can block incoming connections and, if necessary, tell the service synchronize any dirty state to disk. The callbacks (with different parameters) are also invoked as part of the undo cycle whenever the undo manager restores a storage checkpoint, causing the system's on-disk state to change. In this case, the service proxy is expected to shut down, suspend, or kill the service upon `quiesce`, and reinitialize it or reload its state from the storage later upon `unquiesce`; this process ensures that the service is aware of the changes to its underlying storage, and that future interactions will reflect those changes.

5.4.2 Mediating verb execution: undo manager operation

When the service-specific proxy transfers a verb to the undo manager using the `enqueueInteraction` API, it is immediately added to the end of a queue of pending verbs (the scoreboard). The enqueue routine scans the existing contents of the queue, checking if the new verb depends on—does not commute with—any already-present verbs. If so, those verbs are added to a dependency list attached to the new verb, and the new verb is marked as pending. If allowed and preferred by the verbs' sequencing policies, the new verb may bypass other pending verbs with which it does not commute, in which case the dependency lists are updated so that the bypassed verb now depends on the new verb. If the new verb has no dependencies or bypasses all of its existing ones, it is marked as ready to execute. Once the verb has been inserted in the queue and its dependencies updated, the enqueue routine returns, releasing the thread of control back to the proxy. The proxy's thread will only block in the undo manager if the scoreboard is full.

The undo manager has its own pool of worker threads that service the scoreboard queue, decoupling it from the proxy. When the queue is non-empty, these threads scan through it, selecting the next verb that is ready to run. When such a verb is found, it is marked as executing, and the worker thread invokes the verb's `execute` method, passing it the saved `CxnState` and `ResponseInfo` structures, if appropriate. As part of its execution, the verb is expected to fill in the `ObservedOutput` portion of its tag with any output it sent outside the sphere of undo. After the verb executes, the worker thread serializes the verb's tag, writes a record to the timeline log containing the serialized tag and the type of the verb (unless the verb failed and its policy specifies no replay upon failure), then removes the completed verb from the scoreboard and releases any dependent verbs.

When the undo manager's virtual time is set back from the present, indicating that an undo cycle is in progress, the execution process is a bit different. 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. Although the undo manager could simply reject all user interactions during an undo cycle, a better approach is to allow verbs to examine, but not change, system state. To do this, the undo manager first looks at whether the arriving verb is synchronous or asynchronous. If it is asynchronous, the verb is put on a queue of deferred verbs where it waits until the undo cycle completes, at which point it can be safely executed and logged. If the verb is synchronous, another approach is needed since the deferral delay could be arbitrarily long. Instead, the undo manager executes the synchronous verb, but passes a flag to the `execute` routine indicating that the execution must be read-only. If the verb cannot be executed read-only, the `execute` method 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.

```
interface UndoManager {
   // Status APIs
   int      getUndoStatus();
   Date     getVirtualTime();

   // Three-R's APIs
   void     rewind(Date rewindDate);
   void     repairDone();
   Vector   replay(Date endPoint, boolean abortOnParadox)
   void     commitUndo(boolean execDeferred);
   Vector   replayAndCommit(boolean execDeferred,
                             boolean abortOnInconsistency);
   void     cancelUndo(boolean execDeferred);

   // Timeline log editing APIs
   void           addHistoryEdit(long lsn, HistoryEdit he);
   HistoryEdit    removeHistoryEdit(long lsn);
   HistoryEdit    queryHistoryEdit(long lsn);
}
```

Figure 5-2: Undo manager time travel API. The code snippet above illustrates the key methods of the time travel API provided by the undo manager. These methods are used to carry out a Three-R's undo cycle, and are typically invoked by a human operator via a user interface. Most methods can take either a Java `Date` object or an LSN to specify times; for simplicity, we have only shown the `Date` versions here. The interfaces have also been simplified slightly for presentation, and thrown exceptions have been elided.

5.4.3 Managing the timeline: operation APIs

Finally, the undo manager is responsible for coordinating the entire undo cycle. The API that it provides for time-travel operations is listed in Figure 5-2; typically, these operations are invoked by a human operator via a user interface such as those described below in Section 5.6.

A Three-R's undo cycle begins when the undo manager's `rewind` routine is called. In response, the undo manager stops accepting verbs for execution, lets any currently active verbs drain, then asks the service to sync its state to disk by invoking the `quiesce` callback described above. Next, the undo manager takes a reserved checkpoint that acts as a recovery point, then restores the latest checkpoint predating the requested rewind point. If necessary, it then replays verbs from the timeline log to bring the system to the exact rewind point. Once the system is rewound, the undo manager sets the virtual time appropriately and enables read-only verb execution.

During the Repair phase of the Three-R's, the undo manager essentially remains idle, except to mark when changes are made to the service system or to the timeline log. In the first case, the operator invokes the `repairDone` API call when he or she completes changes or repairs to the service system; the undo manager uses the call as a cue to take a checkpoint that captures the repairs, providing a recovery point should a failure occur during replay. In the second case, the operator can request explicit changes to the timeline log via the `add/removeHistoryEdit` API calls. These calls update an internal undo manager table that marks certain LSNs as deleted or altered, and points to the new verbs in the case of altered LSNs.

When the system operator is ready to begin the Replay phase, the undo manager's `replay` method is invoked. The `replay` method carries out a scoreboarding process very similar to that described above for the original execution of verbs, with the exception that verbs come from the timeline log rather than from the service-specific proxy. Using scoreboarding for replay allows independent verbs to replay in parallel, reproducing the original parallelism in the verb stream and potentially increasing the throughput of replay over the original verb execution.

To implement this parallel replay, the undo manager repeatedly reads a verb record from the timeline log (respecting any edits made during repair). It reconstructs the verb from its stored tag, then inserts the verb into the replay scoreboard (which is a separate entity from the sequencing scoreboard, although identical in structure). As verbs are inserted, they are checked for dependencies on verbs already in the scoreboard just as during normal sequencing. Separate worker threads service the replay scoreboard in parallel, finding verbs that are free of dependencies and thus ready to run.

When a worker finds a verb that is ready to run, it checks to see if the verb fails to commute with any previously-replayed inconsistent verbs. If so, it invokes the verb's `squash` routine; otherwise, it invokes the `execute` routine. After execution of the verb, the worker thread invokes the verb's consistency-checking routine, passing it the tags generated by the original and new execution. If the routine indicates externalized inconsistencies, the verb's inconsistency handler is invoked. Once a verb has been processed, it is removed from the scoreboard and any dependent verbs are released.

Once replay is complete, the operator can choose to commit the undo cycle by invoking `commitUndo`, in which case a new checkpoint is taken, the virtual time is reset to the present, and any deferred asynchronous verbs are executed. The API also includes a `replayAndCommit` method that atomically replays to the end of the log then commits the undo cycle, ensuring that no new verbs arrive between replay and commit. Alternately, at any time the operator can cancel the undo cycle by invoking `cancelUndo`, in which case state is restored to the pre-rewind checkpoint and any deferred verbs are executed.

Finally, besides the APIs listed in Figure 5-2, the undo manager allows services or service proxies to register callbacks that are invoked when the undo system's state changes or when a Three-R's operation is carried out. These callbacks can be useful in cases where, for example, the proxy maintains extra data structures during replay to optimize verb execution, as we do in the e-mail service proxy described in Chapter 6.

5.5 Maintaining time-invariant names: the `UIDFactory` module

While not a core component in the undo architecture of Figure 4-1, an important adjunct module is the `UIDFactory`, a Java class that supplies service proxies with UndoIDs (time-invariant unique names) and name translation functionality. The `UIDFactory` is a database that provides a source of UndoIDs and maps bidirectionally between allocated UndoIDs and service-specific names (represented as `Serializable` objects). Each time the service-specific name for a particular object is changed by a user interaction, the service proxy (via that interaction's verb) enters a new mapping for the object's UndoID. The name mappings are temporally-aware: they have start and end timestamps, representing when they were originally created and when they were supplanted, respectively. New mappings for an UndoID do not overwrite the prior mapping, but rather are added to a history of service-specific names for that UndoID.

The database is synchronized with the undo manager so that only the mappings valid at the current virtual time are visible. This property allows a replaying verb to simply query the `UIDFactory` when it needs to translate a logged UndoID-based name to the active service-specific name; the `UIDFactory` uses the current virtual time to return the mapping that was in place at the time when the verb was originally executed. Furthermore, the `UIDFactory` tracks the progress of an undo cycle. Any mapping updates made during the cycle are considered tentative until the undo cycle completes, at which point they are either made permanent or discarded depending on whether the undo cycle commits or is cancelled. Lastly, the `UIDFactory` tracks checkpoint deletions and log truncation events, so that it can purge its history of names when older translations become unreachable.

Internally, the `UIDFactory` is built on top of two BerkeleyDB databases, one each for the forward and reverse mappings of names to UndoIDs. The last-allocated UndoID is stored persistently in the databases to preserve the uniqueness of UndoIDs across sessions. To handle tentative mappings made during an undo cycle, the `UIDFactory` maintains an active epoch, which is an integer value incremented every time the undo manager's virtual time moves backward (*i.e.*, every time `rewind` is invoked). When an undo cycle is committed, the current epoch value is made permanent. When an undo cycle is cancelled, the epoch counter is rolled back to the last committed epoch, and all intervening mappings are discarded from the databases.

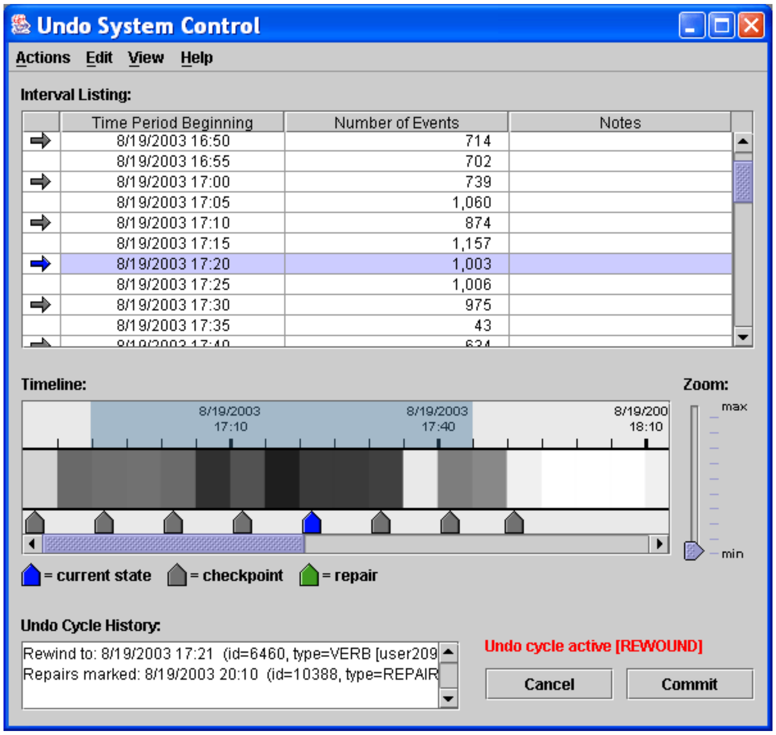
5.6 User interfaces

Our implementation of the undo architecture of Figure 4-1 provides two user interfaces for the system operator. The first interface is a simple text-based, request-response interface designed for debugging purposes and as a target for automation. It is accessed by connecting to a specific network port on the system running the undo manager, and offers direct access to the undo manager's API routines. For example, to rewind the system to LSN 904, an operator could telnet to the interface port and type "`REWIND 904`". Later, he or she could type "`COMMIT`" to commit the resulting undo cycle. No attempt is made by this interface to interpret parameters or results, and no error checking is performed. However, the interface is fast and implemented at a low enough level in the system that it should be always available whenever the undo manager is running; coupled with the fact that it provides several debugging commands to dump the state of the undo system's data structures, this makes it a useful back-door should the system become overloaded or start misbehaving.

Most operators will interact with the undo manager via the second interface, a Java-based graphical user interface (GUI).[1] Like the telnet-based interface, the GUI is also available remotely, in this case via a Java RMI connection to the undo manager. The GUI provides more structured access to the undo manager APIs, with additional error checking, simplified parameter sets, and task guidance.

Figure 5-3 shows the main window of the GUI. It presents a view of the current state of the system, represented as a continuously-updated visual timeline showing the density of processed verbs over fixed-length intervals that span the system's history log. The timeline is tagged with the current virtual time, the locations of active checkpoints, and with markers indicating where repairs were made as part of earlier undo cycles; these markers help the operator choose rewind points and keep track of the changes they have made to the system. During an active undo cycle, the GUI's main window is

[1] The undo manager GUI was originally developed for the undo system by Billy Kakes and subsequently heavily modified by the author.



	Time Period Beginning	Number of Events	Notes
➡	8/19/2003 16:50	714	
	8/19/2003 16:55	702	
➡	8/19/2003 17:00	739	
	8/19/2003 17:05	1,060	
➡	8/19/2003 17:10	874	
	8/19/2003 17:15	1,157	
➡	8/19/2003 17:20	1,003	
	8/19/2003 17:25	1,006	
➡	8/19/2003 17:30	975	
	8/19/2003 17:35	43	
➡	8/19/2003 17:40	634	

```
Rewind to: 8/19/2003 17:21  (id=6460, type=VERB [user209
Repairs marked: 8/19/2003 20:10  (id=10388, type=REPAIR
```

Undo cycle active [REWOUND]

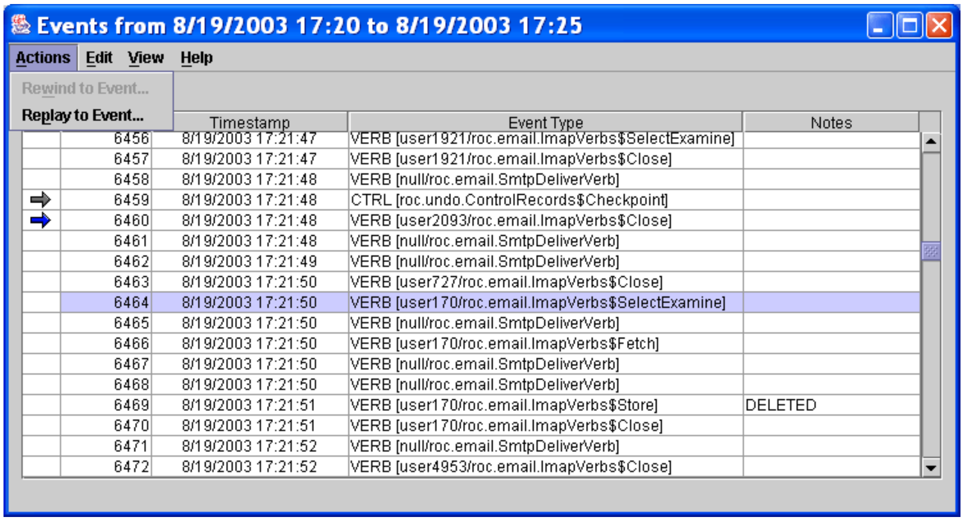
Figure 5-3: Main window of operator GUI. The undo system GUI's main display presents the operator with a view of the timeline of the system, with the level of activity over time represented as varying shades of gray. It also shows the locations of active checkpoints, the current virtual time, and any repairs made in previous undo cycles. Along with the timeline visualization, the main window shows the current undo cycle state and provides controls for committing and cancelling an undo cycle.

updated to display a list of all the time-travel commands that have been executed, the current undo state or operation in progress, and buttons to commit or cancel the undo cycle.

By clicking on an interval on the timeline of verbs, the operator can get a list of all the verbs that occurred in that interval, annotated with the times they were executed and the names of their associated end users, as illustrated in Figure 5-4. Verbs are displayed using the verb's Java type as an identifier (easily extended to a verb-specific representation as returned by the verb's `toString` method), keeping the GUI service-independent. After selecting a verb in this visual representation of a portion of the undo system's timeline log, the operator can rewind or replay the system to that point, or edit the timeline log to disable the verb during a subsequent replay. Before any time travel operation is carried out, the operator is presented with a dialog box describing the consequences of the operation.

5.7 Summary

We have described our implementation of the service-neutral portions of the Three-R's undo architecture developed in Chapter 4. The implementation is centered around verbs, semi-opaque representations of user/service interactions that can be managed and replayed by a service-neutral undo manager. Verbs are generated by a service-specific proxy, sequenced by the service-neutral undo manager, logged in a service-neutral timeline log, and later replayed through the same undo manager. The implementation also includes a rewindable storage layer based on a Network Appliance filer, with significant



		Timestamp	Event Type	Notes
	6456	8/19/2003 17:21:47	VERB [user1921/roc.email.ImapVerbs$SelectExamine]	
	6457	8/19/2003 17:21:47	VERB [user1921/roc.email.ImapVerbs$Close]	
	6458	8/19/2003 17:21:48	VERB [null/roc.email.SmtpDeliverVerb]	
➡	6459	8/19/2003 17:21:48	CTRL [roc.undo.ControlRecords$Checkpoint]	
➡	6460	8/19/2003 17:21:48	VERB [user2093/roc.email.ImapVerbs$Close]	
	6461	8/19/2003 17:21:48	VERB [null/roc.email.SmtpDeliverVerb]	
	6462	8/19/2003 17:21:49	VERB [null/roc.email.SmtpDeliverVerb]	
	6463	8/19/2003 17:21:50	VERB [user727/roc.email.ImapVerbs$Close]	
	6464	8/19/2003 17:21:50	VERB [user170/roc.email.ImapVerbs$SelectExamine]	
	6465	8/19/2003 17:21:50	VERB [null/roc.email.SmtpDeliverVerb]	
	6466	8/19/2003 17:21:50	VERB [user170/roc.email.ImapVerbs$Fetch]	
	6467	8/19/2003 17:21:50	VERB [null/roc.email.SmtpDeliverVerb]	
	6468	8/19/2003 17:21:50	VERB [null/roc.email.SmtpDeliverVerb]	
	6469	8/19/2003 17:21:51	VERB [user170/roc.email.ImapVerbs$Store]	DELETED
	6470	8/19/2003 17:21:51	VERB [user170/roc.email.ImapVerbs$Close]	
	6471	8/19/2003 17:21:52	VERB [null/roc.email.SmtpDeliverVerb]	
	6472	8/19/2003 17:21:52	VERB [user4953/roc.email.ImapVerbs$Close]	

Figure 5-4: Events list window of operator GUI. The events list window presents the operator with a list of all logged events (verbs and control records) that occurred during one interval in the timeline log. It also visually indicates where checkpoints, repairs, and the current virtual time fall in that interval. After selecting an event, the operator can rewind or replay the system to that event, or mark the event as disabled for a subsequent replay.

enhancements to support Three-R's undo semantics; a helper module that generates and tracks unique names for state; and a pair of user interfaces.

By centering our implementation on verbs, we were able to achieve a relatively clean split between the service-neutral undo components, described in this chapter, and the service-specific components, examples of which are described in the next two chapters. Verbs make this split possible: they encapsulate service-specific details about specific user interactions and how to execute them, while still exposing key policy information regarding verb execution semantics and paradox handling policy via a structured, service-neutral interface. Verbs are therefore the key to merging the generic undo system implementation with a specific service; once verbs are defined for a specific service, the service automatically achieves Three-R's undo functionality. We will see examples of how verbs are defined for e-mail and auction services in the next two chapters.

Finally, our implementation was able to leverage several exiting technological building blocks, including the Network Appliance Filer and several Berkeley DB databases. Coupled with our unusual recovery-oriented coding style, our reuse of these existing, proven-reliable components significantly bolstered the robustness of our implementation and reduced its complexity.

Now that we have discussed the implementation of the service-neutral parts of our Three-R's undo architecture, we are ready to see how the architecture can be deployed for a real service application. We will do this over the next two chapters, looking first at a prototype implementation of Three-R's undo for e-mail, then examining how undo functionality might be added to an online auction service.

Chapter 6

Undo Case Study: An Undoable E-mail Store Service

> "What we've got here is failure to communicate."
>
> — *Cool Hand Luke (Frank R. Pierson)*

The undo architecture described in Chapter 4 provides the foundation for building undoable self-contained services. But it is only a framework, albeit an extensive one. To completely materialize a working undoable service, the framework needs to be sheathed with service-specific structures:

- a set of verbs that span and encapsulate the set of interactions handled by the service
- a set of paradox detection policies that encode the service's external consistency model
- a set of paradox-handling and squash routines that describe how the service will treat undo-induced inconsistencies
- a proxy that bonds the undo framework to the service itself by translating between verbs and the service's native protocols.

These service-specific components are the keys to making a workable undoable service. They define how the undoable service will behave, how it will perform, and their form and flexibility ultimately prescribe exactly which services can be made undoable.

In this and the next two chapters, we explore what it takes to create the service-specific components of an undoable system. In this chapter, we examine one application in depth: an Internet e-mail service, chosen for its blend of practical importance, hard state requirements, and relaxed external consistency. We discuss its verbs, paradox policies, and proxy architecture, and analyze its performance and overhead. In the next chapter, we then turn to another important service, online auctions, with a paper analysis of what it would take to make it undo-enabled. Finally, in Chapter 8, we synthesize our experiences with the service-neutral architecture and our two case studies into a set of guidelines and criteria that can be used to establish whether a given service can be made undoable in our architecture.

6.1 Overview: an Undoable E-Mail Service

Our first target application for undo is an e-mail store service that receives mail via SMTP and provides retrieval access via IMAP. We chose e-mail as a first implementation for several reasons. First, e-mail is a widely deployed, increasingly mission-critical service; studies report that up to 45% of critical business information is stored in e-mail [82], and that loss of e-mail access can result in up to a 35% decrease in worker productivity [88]. Second, e-mail as a service depends on its hard state—the contents of users' mailboxes and folders. Hard state is always a challenge in recovery-oriented systems, since it must be preserved and cannot be regenerated if lost. Thus, by working with e-mail, we were forced to ensure that our undo retrofit could properly manage and recover hard state. Third, an e-mail store meets our definition of a self-contained service: it deals only with human users except for a one-way incoming stream of received e-mail. Finally, unlike other hard-state management systems like file systems, block stores, or relational databases, an e-mail system is an entire service that spans the stack from hard state to human end user; because it deals with humans at the top of this stack, e-mail naturally has a relaxed consistency model and offers the opportunity for managing undo-induced paradoxes.

Our undoable e-mail service follows the undo architecture of Chapter 4, and is built on top of the implementation of the generic undo manager described therein. Our implementation is a wrapper that surrounds an existing, unmodified e-mail store service. It requires that the e-mail store service communicate with its end users via the SMTP protocol (for mail transport) [61] and the IMAP protocol (for mail retrieval and folder/mailbox manipulation) [24]. The wrapper consists of a set of verbs that capture IMAP and SMTP protocol interactions, a set of policies for detecting and handling paradoxes, and a proxy that interposes on the existing service's IMAP and SMTP traffic streams, gluing the service to the undo manager.

6.2 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. Table 6-1 lists these verbs, their functionality, and their properties.

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 4.4.2. 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.

Verb	Changes state?	Externalizes state?	Async?	Description
DELIVER (SMTP)	✓		✓	Delivers a message to the mail store via SMTP
APPEND	✓			Appends a message to a specific IMAP folder
FETCH	✓	✓		Retrieves headers, messages, or flags from a folder
STORE	✓	✓		Sets flags on a message (*e.g.*, Seen, Deleted)
COPY	✓			Copies messages to another IMAP folder
LIST		✓		Lists extant IMAP folders
STATUS				Reports folder status (*e.g.*, message count)
SELECT				Opens an IMAP folder for use by later commands
EXPUNGE	✓	✓		Purges all messages with Deleted flag set from a folder
CLOSE	✓			Performs a silent expunge then deselects the folder
CREATE	✓			Creates a new IMAP folder or hierarchy
RENAME	✓			Renames an IMAP folder or hierarchy
DELETE	✓			Deletes an IMAP folder or hierarchy

Table 6-1: Verbs defined by the undo wrapper for an e-mail store service. All verbs except for DELIVER are for the IMAP protocol. Some verbs (STATUS, SELECT) do not change or externalize state and are hence not logged, but are included as verbs so that they are properly sequenced by the undo manager.

There is one noteworthy omission in the set of verbs listed in Table 6-1: we do not define a SEND verb to capture sending e-mail. This omission is intentional, and reflects our desire to focus only on the mail store part of the e-mail service, leaving the mail transport aspects of e-mail unprotected by undo. Ignoring mail transport is a reasonable initial simplification: mail transport and mail storage are largely independent problems, unlikely to both be affected by the same kinds of dependability problems and human errors. Furthermore, an undoable mail store can coexist peacefully with a non-undoable mail transport infrastructure (like the Internet). If we were to try to make mail transport undoable, we would no longer have a self-contained service, and would have to meet the Herculean challenge of ensuring that all mail handlers on the transport path either supported a distributed form of our Three-R's undo, or allowed "unsending" e-mail.

From the end user's perspective, an undoable e-mail store without undoable mail transport simply means that message sends will not be replayed after an undo cycle; if users based their decisions to send e-mail on some state of the e-mail store that is altered by an undo cycle, they must manually compensate for their sent e-mail, just as in the case of propagating paradoxes affecting deleted state, described earlier in Section 4.5.3.

6.2.1 Verb implementation: stripping context

Each of the 13 verbs listed in Table 6-1 is realized as a Java class that implements the generic `Verb` interface described in Section 5.1. One of the biggest challenges in developing the verb classes was in deciding what information to store in each verb's tag. Recall that the tag specifies the verb's input parameters, records a trace of its external output, and must be sufficient to reconstruct the verb for replay in a different system context. As we discussed earlier in Section 4.4.1, the latter criterion implies that the content of a verb's tag must be free of **context**—the tag must have a valid interpretation regardless of the system state in which it is examined.

For SMTP, it turns out to be easy to generate tags that are free of context. For the DELIVER verb, we simply capture the parameters passed in to each SMTP command and store them in the corresponding verb's tag. Since these parameters (the message itself plus its sender and intended recipient) are independently specified by the remote initiator of its associated e-mail message, they are necessarily context-free. Furthermore, the DELIVER verb externalizes no output, so context issues do not arise with the output portion of the verb's tag.

For IMAP, it is a different story, and here is where the challenge emerges. In IMAP, operations name state (folders and messages) using names that are highly context-dependent. 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.

It would be impossible to get context-free tags for the IMAP verbs if we simply took the same approach as for SMTP and just recorded the parameters passed in to the IMAP commands. Instead, we needed to map any names specified as parameters to the IMAP commands into the unique and time-invariant **UndoIDs** defined in Section 4.4.1.

For e-mail *messages*, the UndoID mapping is rather simple: each time a message is injected into the mail store (either by an SMTP DELIVER verb or an IMAP APPEND verb), a new UndoID is allocated and inserted into a reserved message header field (`X-ROC-UndoID`). Later, when creating a verb out of an IMAP command whose parameters refer to some existing message, we translate the IMAP names into UndoIDs by fetching the UndoIDs 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 use the `UIDFactory` module described in Section 5.5 to maintain a translation between UndoIDs and IMAP folder names. Since IMAP names are hierarchical, we encode them as tuples consisting of the unqualified name of the folder plus the UndoID of the folder's parent. Thus the IMAP folder `foo/bar/baz` would be encoded as "`baz`" plus the UndoID of `foo/bar`; this compound name would then be entered into the `UIDFactory` and would receive its own UndoID, which uniquely identifies it. Of course, the `UIDFactory` database has to be kept up to date, so any IMAP verbs that cause folder names to be changed (CREATE, RENAME, and DELETE) update the name mappings appropriately.

An example will be useful to illustrate the use of UndoID-based names and the verb tag. Consider the COPY verb, used to copy messages between folders. The IMAP `COPY` command operates on an already-selected folder and takes as parameters a set of IMAP message IDs and the name of a destination folder. The following structure is used to encapsulate these parameters in the verb's tag:

```
class CopyParams {
    long[] msgUndoIDs;
    long   srcFolderUndoID;
    long   assignedFolderUIDs[];
    String destFolderName;
}
```

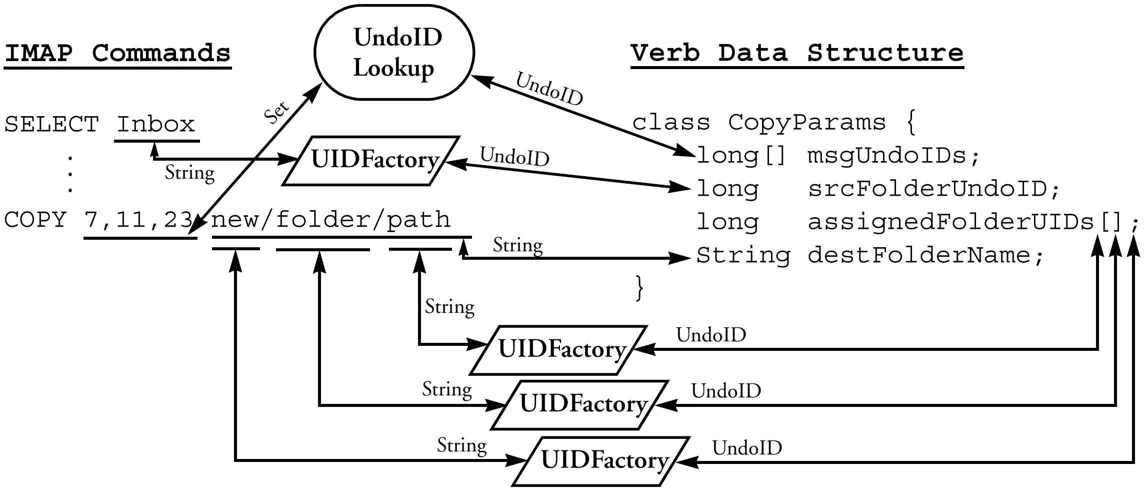


Figure 6-1: Translating IMAP contextual names into UndoIDs. The diagram illustrates how the context-dependent parameters to an IMAP COPY command (with associated folder SELECT) are translated to and from the context-independent representation used in verbs. The *UIDFactory* maintains a database mapping textual folder path components to and from context-independent UndoIDs, whereas the *UndoID Lookup* service translates IMAP message sequence numbers to and from UndoIDs by fetching the translations from the appropriate messages' headers.

Here, `msgUndoIDs` contains the UndoIDs of the messages to copy and `srcFolderUndoID` is the UndoID assigned to the source folder, whose name mapping is guaranteed to exist because it has previously been selected. The remaining two parameters capture the destination folder name and mapping. The `assignedFolderUIDs[]` parameter records the UndoIDs corresponding to each path component in the folder name, and the `destFolderName` parameter records the textual name of the folder path. Figure 6-1 illustrates how the parameters to an IMAP COPY command are translated to and from their context-independent representation as a `CopyParams` structure.

To replay a COPY, the undo system uses the `UIDFactory` to translate the UndoIDs in the `assignedFolderUIDs` array into a folder path. If one or more path component translations is not found, the undo system recreates the missing mapping, assigning the recorded UndoIDs to the path elements in the `destFolderName` variable. Then, the `srcFolderUndoID` is translated into a folder name, again using the `UIDFactory`. Finally, the UndoIDs in `msgUndoIDs` are translated to IMAP message names using the information in the reserved header fields of the messages in the source folder. At this point, the copy operation is ready to be re-executed, since all names have been retranslated to the current system context.

6.2.2 Verb implementation: managing connection context

Besides the context that may exist in verb parameters, another form of context is in the implicit connection state that exists when a client is interacting with the e-mail service. Connection context is primarily an issue for IMAP, where connections are authenticated and implicitly track the active IMAP protocol state (such as which, if any, mail folder is selected as the target of future commands). To capture the latter form of context, we explicitly record the currently-selected folder in the tag of any verb that acts on it. To handle the connection authentication context, we annotate each IMAP verb with the authentication information (username and password) of the active connection. When an IMAP verb is re-executed during replay, it asks the proxy to open a new connection using the saved authentication information; the proxy maintains a cache of these connections to reduce the overhead of replay.

6.2.3 Sequencing e-mail verbs

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.

In some cases, the commutativity tests for these same-user IMAP verb pairings differ depending on if the tests are being used to sequence verbs for execution or if they are being used to detect propagating paradoxes during replay. Typically, the tests are more conservative during original execution than during replay, because less information is available about the verb parameters—for example, the IMAP command may include message wildcards during original execution, whereas during replay the exact message set fetched is known.

Beyond commutativity, we had to address one further aspect of verb sequencing: ensuring that the proxy could tell when each verb completed its execution. For all of the IMAP verbs, this determination is trivial, since those verbs correspond to synchronous user interactions that are complete once their results are returned. The SMTP DELIVER verb, on the other hand, is asynchronous, and thus required a minor modification to the sendmail installation on the e-mail server itself: we inserted a wrapper around the local mail delivery agent that records the completion of each e-mail delivery in a file that is then periodically polled by the undo proxy.

6.3 Handling Paradoxes in E-mail

In order to implement the paradox management architecture described earlier in Section 4.5, we need to define paradox detection predicates, paradox compensation methods, and squash methods and commutativity tests to handle propagating paradoxes. Underlying all of these methods is a consistency policy for the e-mail service that defines what end users should expect in terms of consistency during normal operation and across undo cycles.

6.3.1 An external consistency policy for e-mail

Our consistency policy for e-mail consists of two components. First is an identification of what subset of externalized state is **salient** to end users—that is, the state for which inconsistencies are potentially noticeable, and hence should be managed as paradoxes. Any state not included in this subset can change arbitrarily without a paradox occurring. The second part of the policy describes what forms of inconsistency, if any, are acceptable within the salient state. These two policy components provide sufficient criteria for defining the per-verb paradox detection predicates.

To simplify the problem of defining a consistency policy for e-mail, we will treat e-mail transport (SMTP) and retrieval (IMAP) separately. The transport phase allows for a much more relaxed consistency policy 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 mailbox 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.

More specifically, on the retrieval side we define salient 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)
- the execution status of any state-altering interactive IMAP commands.

Even within this salient state, we allow certain inconsistencies to occur without generating paradoxes. We discount ordering differences in lists of messages and folders, and we ignore messages and folders newly-found during replay that were not present during a verb's original execution; these inconsistencies are typically masked by sorting in the user's e-mail client. Any other differences in the content of salient state, including differences in execution status that result when a salient-state-altering command fails, are considered inconsistencies and thus generate paradoxes.

On the transport side, the only state that is externalized by the SMTP DELIVER verb is the verb's failure status—whether a message delivery succeeds or bounces. Clearly, this failure status is salient, and any inconsistencies in failure status should result in paradoxes. But what do we do 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 and thus cannot manage the propagating paradox that would result.

We would like to preserve the ability for the operator to use undo to repair problems that caused e-mail to be erroneously rejected, which implies that we must find a way to successfully replay previously-failed deliveries. We can achieve this goal by exploiting the weak consistency semantics of e-mail delivery, in effect relaxing the external consistency model of SMTP delivery itself. Following the approach described in Section 4.4.2, we intercept and delay the delivery of bounce messages for a configurable interval (typically, 4 hours). Because SMTP delivery is asynchronous, we do not violate the protocol semantics by doing so, and we gain a window of time 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 length of the undo window (typically 4 hours).

With this change to the way bounces are handled, we can use the simple consistency policy mentioned earlier: the failure status of deliveries is salient, and any inconsistencies in failure status result in paradoxes. The difference is that, now, failure status is externalized only after the configurable delay window ends, giving the operator additional flexibility to make retroactive repairs in the meantime. Finally, note that any inconsistencies in the way message deliveries are handled (such as in the set of recipients that actually receive the message) are not externalized until message retrieval, so they are detected by retrieval verbs and can be ignored from the transport side.

6.3.2 Paradox detection predicates

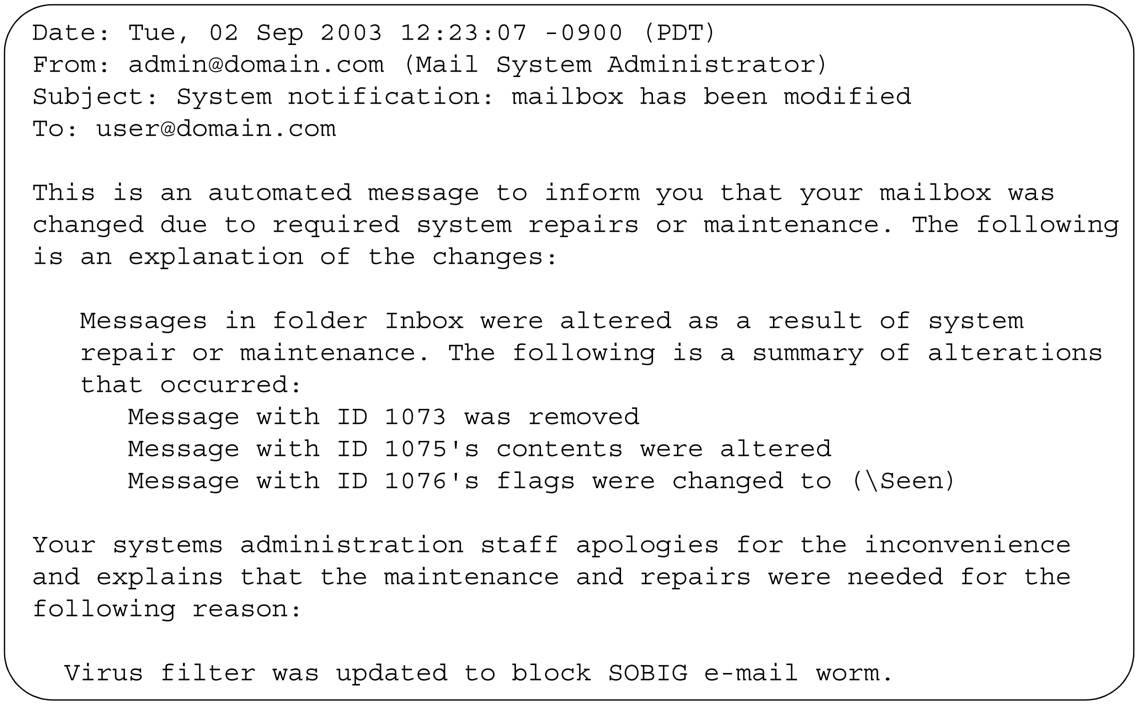
With the consistency policy in place, defining the per-verb paradox detection predicates is straightforward. During normal execution, every retrieval (IMAP) verb that externalizes state records a trace of the salient subset of that state in its tag. The trace can take one of several forms depending on the type of verb. For verbs that expose single objects, such as a message fetch, the trace consists of a hash of the object. For verbs that expose lists of objects, such as the list of folders or the list of messages in a folder, hashes are computed on each entry of the list, and the list of hashes is stored as the trace. Storing the list of hashes allows an order-independent comparison to be made during paradox detection checks, as required by our consistency policy.

During replay, each retrieval verb recomputes the trace based on the salient external output produced during replay, and stores it in the verb's replay tag. The verb's paradox detection predicates receive both the original and replay tags from the undo manager, and can simply compare the traces to determine if a paradox occurred (using an equality test for single-hash traces, and an order-independent list comparison that ignores new entries in the replay tag for hash-list traces). The paradox detection predicates also verify that the verb's failure status (a boolean value) is the same between both versions of the tag; if not, a paradox is declared.

On the transport side, the paradox detection predicate is trivial. If the original DELIVER verb succeeded while its replay execution failed, a paradox is declared. If the original execution failed while its replay succeeded, there is no need to declare a paradox; in this case, we are assured that the replay occurred within the delay window for the reporting of the original failure, since the undo manager will not replay a failed verb once its status has been externalized. Any other situation is paradox-free.

6.3.3 Compensating for paradoxes

On the retrieval side, we compensate for detected paradoxes primarily by inserting explanatory messages into the user's mailbox, apologizing for the inconsistencies, explaining what they are, and saying why they were necessary. Figure 6-2 shows a typical explanatory message that a user might receive. Note that while our current prototype labels inconsistent messages by their IDs, it could be extended



```
Date: Tue, 02 Sep 2003 12:23:07 -0900 (PDT)
From: admin@domain.com (Mail System Administrator)
Subject: System notification: mailbox has been modified
To: user@domain.com

This is an automated message to inform you that your mailbox was
changed due to required system repairs or maintenance. The following
is an explanation of the changes:

   Messages in folder Inbox were altered as a result of system
   repair or maintenance. The following is a summary of alterations
   that occurred:
      Message with ID 1073 was removed
      Message with ID 1075's contents were altered
      Message with ID 1076's flags were changed to (\Seen)

Your systems administration staff apologies for the inconvenience
and explains that the maintenance and repairs were needed for the
following reason:

  Virus filter was updated to block SOBIG e-mail worm.
```

Figure 6-2: Example of an explanatory message sent to a user upon a paradox. This message corresponds to the case where undo was used to retroactively repair the e-mail store's virus filter, causing a previously-visible message to be removed as well as several other alterations. Other paradoxes will produce similar messages that differ in the contents of the explanatory text.

to provide more user-friendly labels (for example, based on message subjects) in all cases except where original messages have been removed.[1]

In some cases, like when new folders (CREATE) or messages (APPEND) 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.

On the transport side, there is no need to compensate for paradoxes. The only paradox that can occur is when a previously-successful delivery fails. In this case, the bounce message generated during replay will take care of any explanation needed; because bounce delivery is asynchronous and not guaranteed to be immediate, there is no problem with sending the bounce during replay. Some care is needed to respect the commit/cancel nature of the undo cycle, such that bounces generated during replay only get sent when the cycle is committed, but this is easily handled by queueing bounces and hooking the undo manager's callbacks to determine when the undo cycle is complete.

[1] The case of removed messages is a problem because the state needed to generate the user-friendly names—namely, the headers of the removed messages—is not available once the system has been rewound. However, the case of removed messages could be handled as well if the user-friendly names were embedded into the log records for FETCH verbs, along with the corresponding UndoIDs, at the cost of slightly greater storage utilization.

All of the compensations described above are implemented as methods in their associated verbs. They rely on common utility functions to handle such tasks as inserting messages into folders. If a compensation method fails—for example, if an explanatory message cannot be added to a folder—the compensation method throws an exception that propagates up and eventually aborts the replay stage of the undo cycle with a message to the operator.

6.3.4 Squashing propagating paradoxes

On the retrieval (IMAP) side, we take a very simple approach to handling propagating paradoxes once they are detected by the undo manager. For those verbs that irretrievably destroy state, we define a squash routine that blocks execution of the verb and leaves an message for the user explaining that they might see extra messages or folders. These verbs include those that discard messages (EXPUNGE, CLOSE) or folders (DELETE), or that overwriting existing state (STORE). For the remaining retrieval verbs, we define null squash routines, as none of those verbs can destroy state. In a few of these latter cases, namely for verbs that only retrieve state like FETCH and STORE, we do set a `silent` flag that causes further inconsistencies produced by these non-state-altering verbs to be ignored.

On the delivery (SMTP) side, we define a null squash routine for the DELIVER verb, as we always want to preserve and attempt to deliver incoming mail regardless of earlier paradoxes.

6.3.5 Recovery guarantee for e-mail

With the combination of paradox detection predicates, compensation methods, and squash methods that we have defined, our undoable e-mail store service offers a recovery guarantee based on the lose-no-work guarantee introduced in Section 4.6.1. At a high level, it promises that no state created or modified by end users will be lost as a result of an undo cycle. It does not guarantee that the state will appear exactly as it did before the undo cycle; however, it does promise that any previously-viewed state that changes as a result of an undo cycle or retroactive repair, relative to the consistency policy defined in Section 6.3.1, will be accompanied by an explanatory message pointing out the differences or indicating where the original state has been relocated. In other words, if any differences are visible to the end user in the content of previously-viewed messages, the key headers of those messages, or in the lists of messages and folders that the user had previously retrieved (other than reordering or the addition of new entries to those lists), the user will find an e-mail message explaining the discrepancy; for violations affecting messages, the explanation will be placed in the folder that contained the affected message, while for violations affecting folders, the explanation will be placed in the user's inbox.

In the best case, where the operator invokes undo to recover from a problem that did not cause incorrect processing of user results—for example, a failstop system crash, an upgrade that produced a non-working system, a performance problem, and so on—the guarantee promises that end users will see no significant differences in their state, modulo the consistency policy of Section 6.3.1. In the worst case, where the operator uses undo to repair a problem that has affected the way e-mail interactions were processed, the user may find that the contents of messages and of folders have changed, and worse may find that actions they had previously taken (particularly, expunging deleting messages and deleting folders) have been undone and need to be re-executed if still appropriate. Again, in these situations explanation is provided to users to indicate what has changed and why, helping them understand what state and actions they need to re-evaluate given the consequences of the undo cycle.

While this guarantee does put some burden on the system's end users to clean up the mess resulting from a particularly paradox-prone undo cycle, it still offers a significant improvement over the alternatives, which are either to throw away all work by restoring a backup or to fail to recover from the problem. That said, the guarantee could be enhanced further through at least two more improvements: first, the undo system could provide the *operator* with a guarantee of how many users would be affected and how severely, *before* the replay process is begun; this could be done by simulating replay inside a virtual machine cloned from the active state of the rewound system. Alternately, the operator could be provided with this guarantee after the replay phase but before committing the undo cycle. This approach would allow the operator to back out (cancel) the effects of the undo cycle if the paradox consequences were too great, without requiring extra mechanism to predict or simulate the effects of replay.

A second improvement to the recovery guarantee would result if the undo system could provide a way for individual users to express their own consistency policies to the system. Then, users who did not care about, say, altered message contents or paradoxes propagating to message deletions could indicate as such, thereby avoiding having to deal with extraneous explanations and compensations. Both this approach of involving users in the recovery guarantee and the above approach of dynamically providing a situation-specific guarantee to the operator are interesting directions to pursue in future generations of the undo architecture.

So far, we have only talked about recovery guarantees to the end users of the e-mail store service. The system also provides a recovery guarantee to those external entities that send mail to an undo-enabled e-mail store. This recovery guarantee is much simpler: it promises that all messages sent to the service will be delivered, or, if delivery is impossible, will be bounced back to the sender. This is the same guarantee provided by standard SMTP e-mail delivery, with the exception that in our system, bounces may be delayed by several hours and supplanted in the meantime with status messages indicating a potentially-transient delivery problem.

Finally, note that our undo recovery guarantee makes the reasonable assumption that the IMAP and SMTP protocols remain stable and unchanged across undo cycles, and that the implementation of the e-mail service does not add or remove extensions to the core protocols across undo cycles.

6.4 E-mail Service 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. Threads are drawn from a thread pool that maintains both a low and high watermark for active threads, helping to reduce the overhead of accepting a new connection while also preventing overload by limiting the maximum number of threads in the system at once.

Each thread handles an entire protocol session (the length of a connection) as a unit of work; a natural extension to increase concurrency and throughput would be to make the proxy more event-driven by multiplexing threads amongst connections. The worker thread 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. 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 trade-off between attack vulnerability and the system's ability to recover from configuration errors affecting address validation, and is probably a decision that should ultimately be left to site policy.

6.5 Evaluation of Overhead and Performance

With an implementation of a specific service defined and its verbs and proxy in place, we can now explore the overhead and performance of the prototype undo manager described in Chapter 5, as manifested in our undo-enabled e-mail service. Our goal in this section is only to determine whether Three-R's undo is even feasible; our prototype has neither been highly tuned for peak performance nor optimized for minimal overhead, so our results should not be taken as the best-case behavior of an undoable system. Also, the experiments in this section are not designed to evaluate the potential of undo for improving dependability or addressing common failure scenarios; that form of evaluation requires a dependability benchmark incorporating human subjects, a topic we will return to in Chapter 10.

6.5.1 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. Table 6-2 gives details of the machine configurations. 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 20,000 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 [118]. SPECmail simulates the workload seen by a typical ISP mail server with a mixture of well-connected and dialup users. The benchmark's measurements are based on **sessions**, defined as a single complete set of client interactions with the mail server, from login to logout, without think time.

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 sum-

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 6-2: Machine configurations for experimental evaluation of undo-wrapped e-mail store service.

mary 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. Unless otherwise noted, the benchmark was typically 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.

6.5.2 Time overhead: throughput and latency

Since our undo system wraps around an existing e-mail store service as a proxy, it is likely to add some performance overhead to end-user requests. Our first set of experiments was designed to evaluate this overhead.

Throughput. We began by analyzing the throughput of the e-mail service with and without the undo facility in place. Following the metrics used by SPECmail2001, we defined throughput in terms of the number of sessions completed by the benchmark client: in SPECmail2001, a benchmark run is valid if at least 99% of the sessions are failure-free and if at least 95% of them meet a specified level of quality of service (QoS). We determined the maximum throughput of each system by varying the applied load (via the number of simulated end users) until the throughput metrics dropped below the 99% completion or 95% QoS thresholds. We examined IMAP and SMTP separately, both because the SPECmail benchmark does so and because in our Sendmail/UW-IMAP service the IMAP and SMTP protocols are handled by separate server processes.

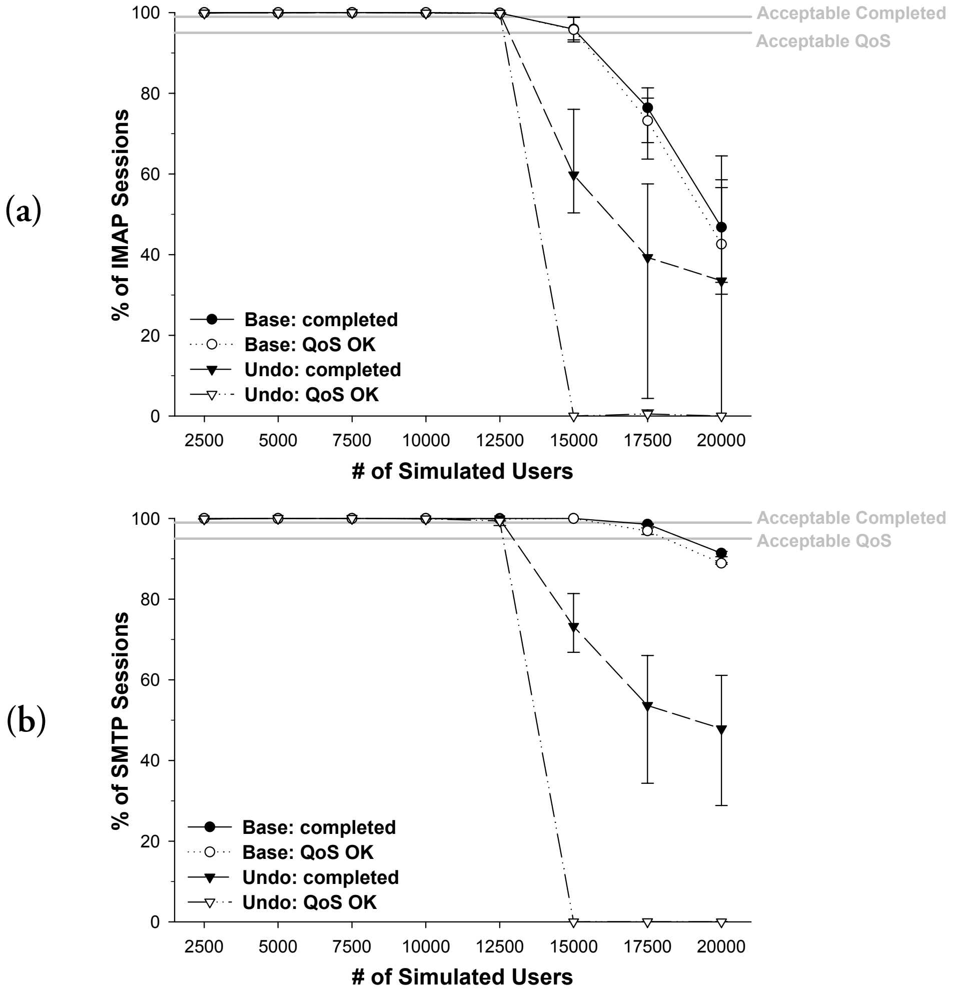


Figure 6-3: Throughput comparison of e-mail store service with and without undo wrapper. The graphs plot the fraction of SPECmail simulated end-user sessions that completed successfully (solid symbols) or met SPECmail's quality of service (QoS) guarantees (open symbols), for different levels of applied load. The circles represent data taken from the base, unmodified e-mail store service; the triangles represent data from the undo-wrapped e-mail store service. The upper graph, (a), plots IMAP sessions while the lower graph, (b), plots SMTP sessions. All data points represent the average of three runs, with error bars showing the minimum and maximum values over those runs. The graphs show that the undo-enabled service offers slightly less maximum throughput in the IMAP case, and more noticeably less in the SMTP case.

Figure 6-3 plots the throughput results for both IMAP and SMTP using the system configuration of Table 6-2. For the IMAP results in Figure 6-3a, we see that the undo-enabled system tracks the throughput of the non-undo-enabled system almost exactly up through 12,500 simulated users; the non-undo-enabled service edges out the undo-enabled service at 15,000 users, then both systems behave poorly after that. These results suggest that the undo wrapper reduces the system's maximum throughput compared to the e-mail service alone, but below the saturation point (about 12,500 users in this case) has little effect on throughput. The SMTP data in Figure 6-3b bears out this observation, although in this case the undo wrapper affects the saturation point more significantly, limiting it to about 12,500 users with the undo wrapper enabled versus about 17,500 users without the undo wrapper in place.

In these experiments, the undo wrapper limits maximum throughput because of resource limitations on the system running the proxy and undo manager: at over 12,500 simulated SPECmail users, the proxy runs out of threads in its thread pool and starts dropping or interminably delaying incoming connections. Also, because the undo proxy uses the same thread pool to service both IMAP and SMTP connections, when one protocol saturates it affects the other. This coupling explains why the undo-enabled system's SMTP saturation point is so much lower than the base non-undoable system's: the base system's IMAP server saturates at about 15,000 users, saturating the proxy with stalled IMAP worker threads and leaving few or no threads to proxy the incoming SMTP connections that could still be handled by the base system's SMTP server.

Because the proxy is thread-limited, it should be possible to remove the proxy's throughput bottleneck by providing it with more threads and more hardware resources to run those threads concurrently (or by rewriting the proxy to multiplex threads across connections in a more event-driven style). Because we were limited by available hardware resources, we were unable to directly test this hypothesis by scaling up the proxy. However, we were able to carry out a set of experiments with the e-mail store server itself scaled down, simulating what would happen if we shifted the balance of processing power by scaling up the proxy. Figure 6-4 shows the results from these experiments, which were performed with the e-mail store server configured with only one of its four CPUs active and with its available memory reduced by half, to 1GB.

As can be seen in Figure 6-4, with the proxy's resources boosted relative to the e-mail store server's, the undo-enabled system tracks the throughput of the non-undoable system almost exactly, with only a slight deviation in the SMTP case. These results show that, despite the extra sequencing it performs, the undo wrapper does not represent an inherent throughput bottleneck as long as the proxy has enough thread resources to keep up with the connection load being handled by the server itself.

Latency. Next, we compare the user-visible latency with and without the undo system in place. Figure 6-5 reports the results of a set of experiments designed to compare the execution latency of individual IMAP and SMTP commands. We executed each protocol command in a tight loop on an unloaded system, and measured the end-to-end, client-visible latency with and without the undo system in place. The measurements were performed using the hardware of Table 6-2, the latency metrics were collected from the machine labeled "workload generator", and that machine was connected to the e-mail service and undo proxy via gigabit Ethernet.

What we see in the results of Figure 6-5 is that the latency impact of undo is highly variable. The undo system imposes little or no latency overhead on some metrics, like the latency to establish an IMAP connection or to login to the IMAP server, or to close an IMAP folder, or to append a small message to an IMAP folder. In other cases, the need to generate verbs, strip context, and log operations adds a noticeable amount of latency, but latency that is still well below the threshold where an end user would start to feel the system getting unacceptably sluggish (a threshold typically pegged at about one second [78]); the IMAP LIST, SELECT, CREATE, and STORE commands, SMTP connection establishment, small IMAP message fetches, and large IMAP message appends are all good examples of this case.

On the other hand, the undo system imposes a very significant amount of latency overhead when fetching messages larger than about 128KB; in the worst case in Figure 6-5, fetching a two megabyte

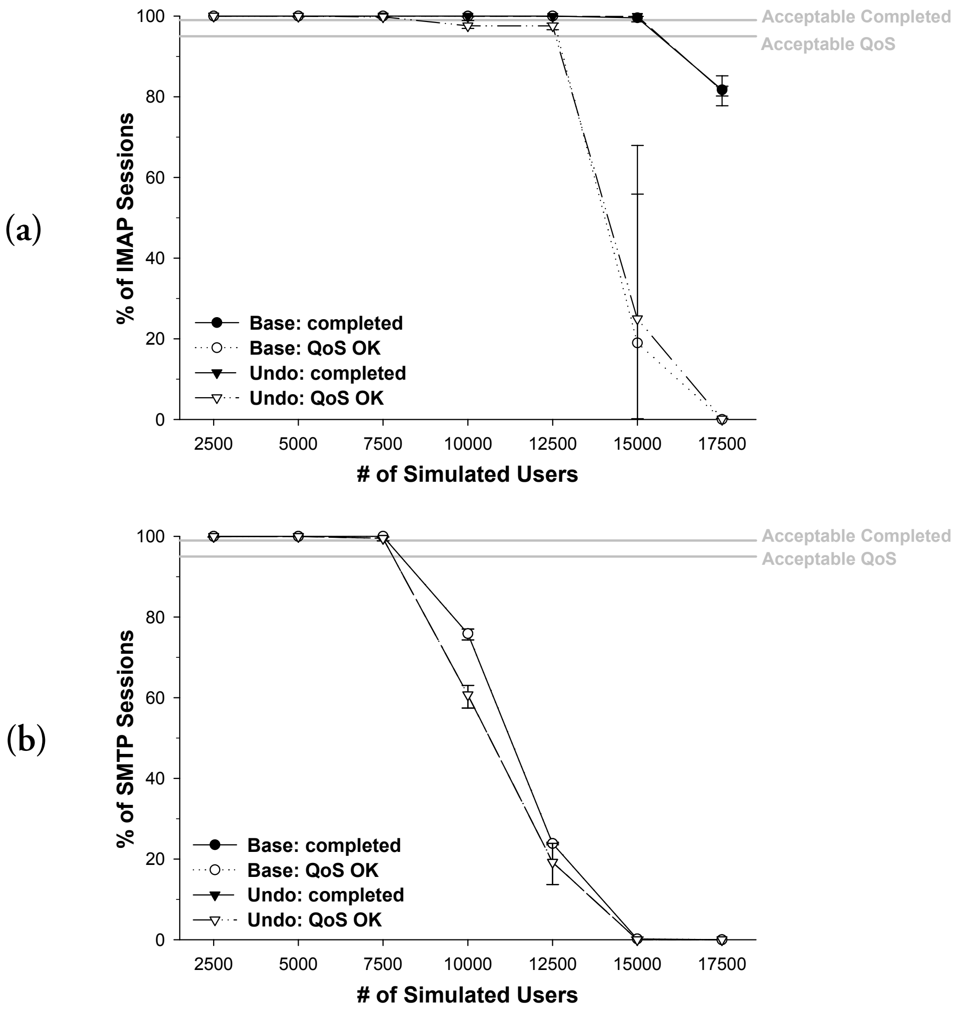


Figure 6-4: Throughput of e-mail store service with service resources reduced. The graphs in this figure are analogous to those in Figure 6-3, except that in this case the CPU resources of the e-mail store service have been reduced from 4 CPUs to 1 CPU, and the service's DRAM has been halved, from 2GB to 1GB. With the balance of power thus shifted from the e-mail store service to the undo proxy, the proxy easily keeps up with the e-mail service, resulting in essentially no noticeable impact on maximum throughput.

message, the extra overhead is on the order of three seconds. This behavior results from our implementation of the FETCH verb as a split-phase operation (as discussed in Section 4.4.2): the undo system must first fetch the verb from the server in its entirety, then later dole it out to the client, at minimum doubling the native operation latency. Rearchitecting the FETCH verb to overlap the two phases of execution would significantly improve the latency overhead in this case.

Finally, the SMTP results in Figure 6-5 deserve special mention. The `smtp-data-*` entries in the graph refer to the latency of delivering a message of the specified size via SMTP. As can be seen in the figure, the undo-enabled system actually performs *faster* than the base, non-undo-enabled e-mail store service. This surprising result is explained by the fact that the undo system does not proxy incoming SMTP traffic, but rather carries out the SMTP dialog directly with the client, generating DELIVER verbs that are executed asynchronously against the underlying e-mail server; the proxy is simply able to complete the interaction more quickly than the underlying SMTP server.

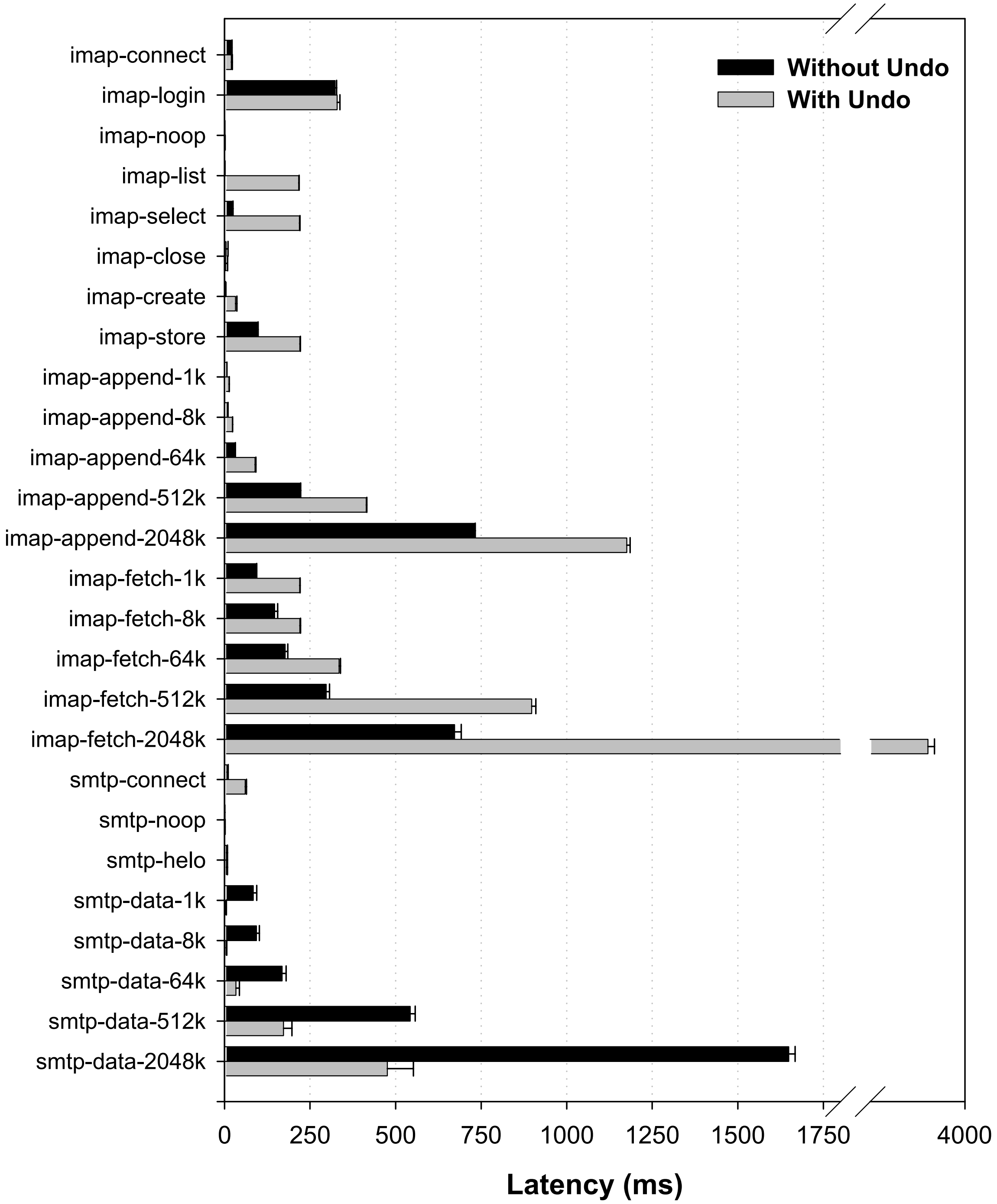


Figure 6-5: Latency overhead of undo for individual IMAP and SMTP interactions. Each bar plots the mean client-visible execution latency over 50 trial executions of the respective command. The upper (black) bar in each pair represents the latency of the command without the undo facility present; the lower (gray) bar represents the latency when the undo facility is in place and is actively proxying and logging verbs. Error bars indicate 99% confidence intervals on the sample means. Timing resolution was 16ms, although commands were run in tight loops where possible to increase accuracy. In most cases, the undo-enabled system performs the same or slightly slower than the base e-mail store service, with the exception of large message fetches (retrievals), where split-phase operation adds significant latency overhead, and SMTP message deliveries, where the undo system's decoupled verb execution results in reduced latency when the undo proxy is enabled.

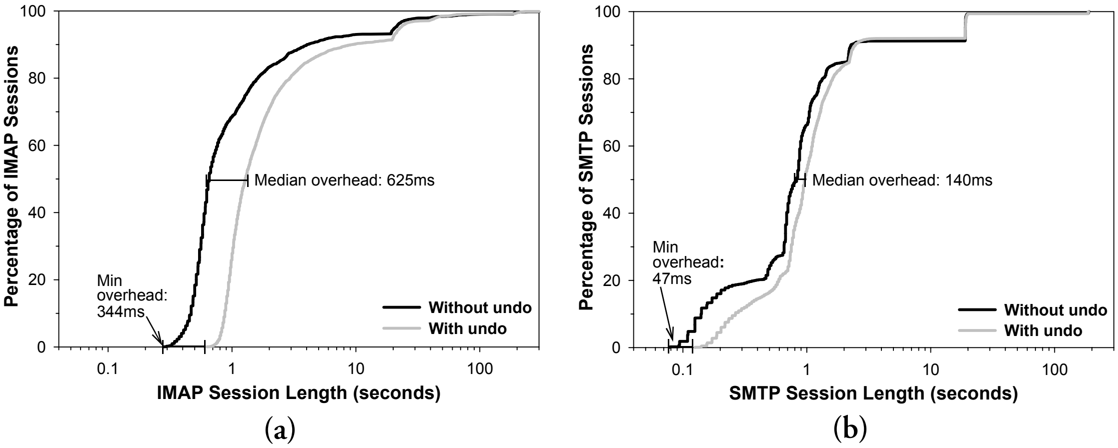


Figure 6-6: "Real-world" latency overhead of undo. The two graphs show the cumulative distributions of SPECmail session times for IMAP (left) and SMTP (right) sessions under a 10,000 user workload. The latency overhead for the minimum and median session in the distributions are marked. While the overheads are not negligible, they are spread across multi-command sessions and fall below the threshold of human annoyance.

To see how Figure 6-5's mixed bag of latency results would play out in real-world workloads, we decided to embark on a second set of experiments. Using the SPECmail workload to represent real life, we measured the cumulative distributions of the lengths of complete IMAP and SMTP *sessions* in a 30-minute SPECmail run with 10,000 simulated e-mail end users, both with and without the undo system in place. Sessions represent entire sets of end-user interactions with the e-mail service, from login to logout, and thus provide a way to look at the aggregate latency impact of the undo system across typical collections of protocol commands.

Looking at the session-length distributions, plotted in Figure 6-6, we see that adding undo functionality to the base e-mail store server does not significantly alter the shapes of the latency distributions, essentially just shifting them to the right. In the IMAP case, the shift corresponds to the extra overhead imposed by the proxy for connecting, logging in, selecting a mailbox, and fetching messages; the overhead diminishes as sessions get longer, since these cases correspond to simulated dialup users where the time to transfer the message data to the client swamps any delay added by the undo proxy.

In the SMTP case, we see that the latency advantage of the undo-enabled system disappears for a real-world workload for sessions shorter than about two seconds; the distribution curves cross just after that point with a slight edge to the undo-enabled system for longer sessions. The disappearance of the latency advantage for short sessions can be explained by the fact that full SMTP sessions involve more than simply the `smtp-data` commands that showed such a latency improvement in Figure 6-5. Unfortunately, the undoable system is slower to process SMTP connection establishments and the initial protocol handshaking, which swamps any later latency improvement in handling the data, at least for small messages. For longer sessions involving more data transfer, the latency advantage does show up, but its impact is minimal for the same reason as for long IMAP sessions: these correspond to slow clients whose slow data transfer rates overwhelm any latency differences between the undoable and non-undoable systems.

6.5.3 Space overhead

Next, we consider the storage overhead of providing undo for our e-mail store service. The storage overhead for undo consists of three parts: the extra storage capacity needed for checkpoints, the storage consumed by the timeline log, and the storage used by the `UIDFactory` database of mail folder name mappings (discussed in Section 6.2.1).

The first part of the overhead, the checkpoint storage, is difficult to measure directly in our implementation. Our checkpoints are provided by snapshots on the Network Appliance filer. Because snapshots are an integral part of the filer's non-overwriting file system and used even during normal operation of the filer, their storage overhead cannot be easily isolated. Typically, a filer will devote 15% of its storage resources to snapshots; we adjusted that portion upwards to 40% for our experiments, which proved easily sufficient given the rate of change of the SPECmail workload. Since snapshots are recycled, the overhead dedicated to keeping them should remain relatively constant, and thus they represent a fixed up-front "tax" on the amount of storage purchased for the e-mail store service.

More of a concern is the storage for the timeline log. The timeline log contains copies of all incoming e-mail and hashes of all outgoing data viewed by end users, and thus threatens to grow and consume storage space quickly. To evaluate the log's growth rate, we measured the amount of log data accumulated during the 30-minute measurement interval of a 10,000-user SPECmail benchmark run. 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 the overhead of this bug 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 and about 19% of which is residual Java serialization and BerkeleyDB logging overhead. 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 by compressing the text of the e-mail messages embedded in the log.

The third component of storage overhead, the folder name database, is relatively static and consumes storage 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.

6.5.4 Performance of the undo cycle

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.

Rewind. With the reversible rewind workaround of Section 5.3.1 in place, it took on average 590 seconds, or almost 10 minutes, to rewind the 10,000-user system (average of three runs, standard deviation <1%). Figure 6-7a shows the rewind time for several different numbers of simulated SPECmail users; the results show that rewind time scales roughly linearly with the number of simulated SPEC-

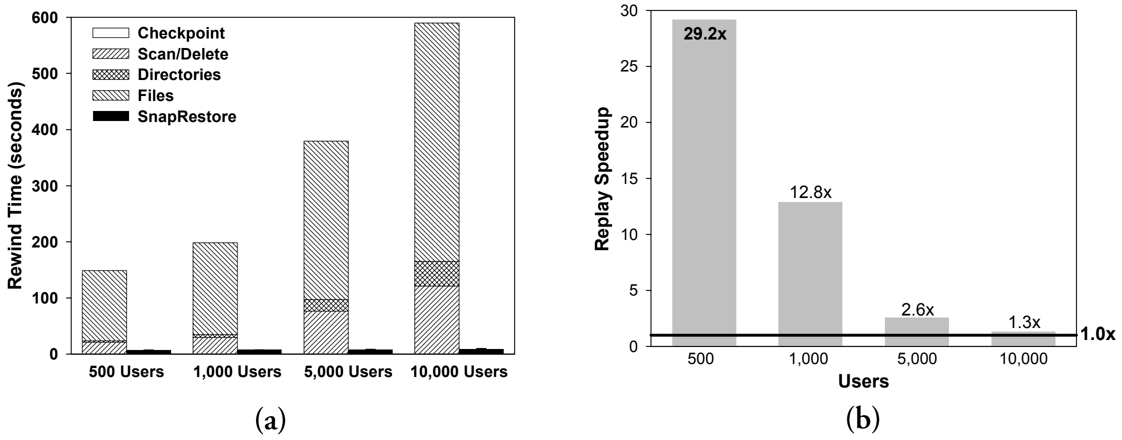


Figure 6-7: Performance of Three-R's Undo Cycle for E-mail Service. The left graph, (a), shows the measured rewind time for various numbers of SPECmail users, broken down into the time spent restoring checkpoints, scanning and erasing the file system, creating directories, and restoring files; for comparison, the bars marked "SnapRestore" plot the rewind time using the Network Appliance filer's built-in non-reversible rewind. The right graph, (b), plots the performance of replay for various numbers of SPECmail users, represented as speedup over the original 30-minute execution time of the benchmark run.

mail users. Figure 6-7a also illustrates the breakdown of time spent in each phase of the reversible rewind workaround. The bulk of the rewind time is spent copying files from the old snapshot into the active system, and, since this time is heavily dependent on the total number of files in the file system, it is not surprising that the overall rewind time scales roughly with the number of users. Finally, Figure 6-7a compares the rewind time with our reversible rewind workaround to that with the Network Appliance filer's built-in snapshot restore capabilities ("SnapRestore"). With SnapRestore, 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 reversible rewind.

Replay. Turning to replay, Figure 6-7b plots the time to replay the logged verbs from a SPECmail benchmark run for several different numbers of simulated users. The replay times are represented as speedups over the original 30-minute run-length. 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. The achieved replay rate falls short of the maximum throughput supported by the e-mail store server (which hovers at about 12 verbs/sec for the SPECmail workload, based on the throughput results of Section 6.5.2), indicating that we are not maximally using the service's capacity during replay. Furthermore, even if we were able to saturate the e-mail store server during replay, at 12 verbs/sec the absolute time for replay would still be significantly longer than we would like.

One of the reasons for the slow replay rate lies in the dependencies that exist between non-commutative verbs. While the undo manager's out-of-order replay algorithm attempts to maximize the parallelism of replay, it is limited by the number of worker threads it can maintain simultaneously and by the size of the "scoreboard" used to find verb-level parallelism. Again, this is a place where rearchitecting the thread model to multiplex threads across slow-running verbs could provide increased performance through greater parallelism.

As an approximation of how much performance could be gained by extracting more parallelism, we disabled dependency checks and reran the replay phase of the Three-R's undo cycle using the same setup as above. Of course, the results of such a replay are incorrect and could never be used in practice, but the approach is useful to get a sense of the impact of dependencies. We found that, with dependency checks turned off, the replay system could push about 13 verbs/sec through the e-mail store service—higher even than our estimated saturation point of about 12 verbs/sec. (This is not a contradiction, since our estimate of the saturation point is based on the SPECmail workload. SPECmail, despite simulating users in parallel, implicitly imposes dependencies within each user's session by generating the session's requests serially; our dependency-free replay ignores even these per-user dependencies.)

Even if we could rearchitect the undo manager to extract all parallelism from the replay verb stream and saturate the server, we would still experience potentially long replay times—on a near-saturated system, times on roughly the same order as it took to generate the log. To get replay to run faster than normal execution, even when the system is near saturation during normal execution, we would need to couple the undo wrapper more tightly to the service itself. An approach that would help, particularly in the e-mail case, would be to remove the overhead of establishing, authenticating, and tearing down SMTP and IMAP connections for each replayed verb, which can take several hundred milliseconds. While these overheads are not significantly different during replay than during original execution, there is no need for them during replay, as the undo proxy is a known, local, trusted entity to the service and need not faithfully simulate the connection setup processes used by remote untrusted users.

Thus, we hypothesize that significant improvements in replay speed could be realized through more optimized connection management. A particularly-useful improvement would be achieved if the IMAP and SMTP protocols provided a "batch mode" that allowed a trusted entity (like the undo proxy) to reuse a single authenticated connection to replay the interleaved interactions of multiple users. Such an improvement could even help with the current undo manager's limited extraction of parallelism, since without the need to re-establish an authenticated connection for each verb replayed, verbs would execute faster and free up limiting thread resources sooner.

6.6 Discussion

Our goal at the start of this subsection was to evaluate the feasibility of Three-R's undo from a performance and overhead standpoint. The conclusion appears evident: despite its limitations, our unoptimized prototype demonstrates that Three-R's undo *is* feasible. Its overhead in terms of space is well within acceptable limits, especially considering the abundance of inexpensive storage available today. Given enough resources, our prototype undo wrapper imposes little or no impact on the service's maximum throughput or overload behavior. While it does impose extra latency on user interactions, the per-command latency remains below the threshold at which humans perceive significant sluggishness except in the case of fetching large messages (a problem easily fixable with minor rearchitecting of the FETCH verb). Furthermore, the overall session latency still falls below sluggishness perception thresholds, and is mostly concentrated in the connection setup and initial protocol handshaking stages of the session.

Much of the visible undo-induced latency can be attributed to the undo system's implementation as a proxy-based wrapper. While some latency is unavoidable—the cost of writing log records, for example—most of the communications and context-extraction latency impact could be mitigated by building the undo system directly into the e-mail service itself. As we pointed out earlier, however, doing so would sacrifice the advantages of our wrapper-based, black-box approach, namely recovery power, reusability, upgrade independence, and tolerance of server failures.

Moving on to the performance of the Three-R's cycle itself, rewind appears to be limited by the snapshot technology of the Network Appliance filer, and with certain API changes to the filer could be made unobjectionably fast. Replay speed is the only Achilles' heel of our prototype system. Our lackluster showing on absolute replay time for 10,000 users can be partially attributed to running the system near saturation, leaving little headroom for replaying user interactions faster than they were originally recorded; the fact that we could achieve replay speedups of between 2x and 29x by reducing the workload suggests that overprovisioning will be crucial for good Three-R's replay performance.

Even without overprovisioning, though, significant improvements in replay speed would be desirable for a practical, production-quality Three-R's undo system. We have suggested two ways of improving replay speed without overprovisioning—by enhancing parallelism of replay and by optimizing connection management through protocol changes—but additional work is needed to explore the potential benefits of these approaches.

6.7 Related Work

We have already discussed one system that attempts to provide undo-like functionality for an e-mail store service: Network Appliance's SnapManager for Exchange [82]. SnapManager, unsurprisingly, is also based on filer snapshots as its fundamental time-travel mechanism, and, in the case of a system failure, allows an operator to restore a previously-archived snapshot of the mail system, then replay forward using the Exchange transaction logs. Similar functionality can be provided by systems that build e-mail on top of a transactional database.

SnapManager and similar systems bear a good deal of resemblance to our system, but they lack several key properties that our undo-enabled e-mail store service provides. 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 black box approach uses external proxy-based logging that 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 providing undo for the e-mail *store* parts of the global e-mail network, there has been work on developing undo-like functionality for the transport/sending aspects of e-mail as well. Many closed corporate-targeted e-mail systems support revoking or "unsending" messages as long as they have remained within the confines of the closed system and have not yet been read; Microsoft's Exchange, SoftArc's FirstClass, Novell's GroupWise, and America Online's internal e-mail are good examples of such systems. Rubin *et al.* have demonstrated a clever use of cryptography that makes similar revocation functionality possible on open networks like the Internet, although their approach has not been widely deployed [104].

These revocable e-mail systems all address a different and orthogonal undo problem to the one we have investigated. However, they do provide an intriguing approach to extending our undo facility to handle more of the transport side of the e-mail problem. To bring the transport side of e-mail under the purview of our undo facility, we would need to intercept and log outgoing e-mail, treating it as externalized state much like the contents of messages exposed to users via the IMAP FETCH command. Should the contents of an outgoing message change as a result of an undo cycle, this could be detected upon replay using our paradox predicate approach, and compensated for by unsending the original message and resending a new, altered one in its place. If the unsend failed because the original message had already been read, a compensating explanatory message could be sent instead. To handle propagating paradoxes involving outgoing e-mail, we would need a way to correlate message sends with the events that provoked them, for example associating an outgoing forwarded message with the verb that delivered the original message being forwarded; this is a challenging problem, especially when the outgoing message's dependencies are not directly visible to the system, and is worthy of future investigation.

Chapter 7

Undo Case Study: Towards An Undoable Auction Service

> "Few men have virtue to withstand the highest bidder."
> — *George Washington*

Amongst network-delivered services, e-mail must surely be followed closely by auctions in any ranking of relative importance. Auction services play important roles across several economic sectors, from handling over $22 billion of consumer-oriented transactions per year through the Internet auction service eBay [31], to serving as the underpinning of negotiation and resource matching in online marketplaces for business-to-business services and commodities.

From the perspective of an operator-targeted undo system, auction services also offer an interesting target. They share many of the properties of the e-mail store service we studied in Chapter 6, such as being self-contained services with human users and nontrivial hard state. But auction services add a new twist to these shared properties: their users often have a financial stake in the correct operation of the system, complicating the problem of paradox management and requiring different tactics than we used for e-mail. Throughout this chapter, our exploration of an appropriate paradox management approach for auctions will shed light on the capabilities and limitations of the Three-R's undo model.

While we have not built an undoable auction service prototype, we have developed a paper design for an undoable consumer-oriented auction service. Our design follows the undo architecture of Chapter 4, and is built on top of the implementation of the generic undo manager described therein. Like our undo wrapper for e-mail store services, our approach to undo for auctions is to wrap an existing auction service with undo functionality, defining verbs, paradox policies, and a protocol-specific proxy.

Since auction protocols are not as standardized as e-mail protocols, our design adopts the protocols of the RUBiS auction system, an open-source research system patterned after commercial consumer-oriented auction services like eBay [2] [105]. We will assume an auction model like eBay's that allows "Buy it Now" sales as well as traditional auctions, and that carries out its auctions with visible bids and proxy bidding (such that each bid specifies a secret maximum bid and the auction service incrementally rebids on the bidder's behalf until the maximum bid is reached).

7.1 Verbs for Auctions

The state of a RUBiS auction service is composed of four types of information:

- static state: unchanging information used to categorize auctions, such as geographical regions of the world
- user information: a user's name, nickname, e-mail address, rating, and feedback
- item state: a for-sale item's ID, name, description, price (for "buy it now" sales), reserve (for auction sales), and quantity; and information on its seller
- auction state: the bid history for a given item, plus bookkeeping information about the start and end of the auction.

End users access and update this state via the RUBiS protocols, which define 26 separate interactions between end users and the auction service. Each RUBiS interaction takes the form of a web page request or submission transmitted over HTTP. One additional (27th) "meta-interaction" occurs when an auction completes, and consists of e-mail notification to the seller and winning buyer (if any). Six of the RUBiS interactions fetch static web pages that never change and do not update state; we ignore them, leaving 21 dynamic interactions to handle. Internally, RUBiS handles these 21 dynamic interactions with Java servlets or Enterprise Java Beans (EJBs), although, as with e-mail, these internal implementation details are not relevant to our undo system design.

Table 7-1 shows the 21 dynamic RUBiS interactions, and indicates which of them alter and expose non-static state (we ignore static state from the point of view of paradox handling, since it is not a factor in auction consistency). These 21 interactions define the verbs for the auction system. Only six of those verbs actually need to be logged, as indicated in the table: the 5 state-altering verbs and one of the externalizing verbs. We will see why the other externalizing verbs can be ignored in the next section, when we define our external consistency model for auctions.

The AUCTIONDONE verb deserves special mention because it is unlike any verb we have seen before in the e-mail system: it represents a "push" interaction that needs to be generated by the service itself rather than by an interaction with an external entity like an end user. AUCTIONDONE could be implemented in one of two ways. In the purely proxy-based approach, the auction service's undo proxy could intercept the outgoing e-mail messages that indicate an auction's completion, generating the AUCTIONDONE verb and logging it at that point. During Three-R's undo, auction-completion e-mails generated during replay would be intercepted as well, and correlated with the logged AUCTIONDONE verbs (by means of the auction ID); the correlated results would then be compared

Verb	Changes state?	Externalizes state?	Async?	Logged?	Description
AboutMe		✓			Display personal info, current auctions
BrowseCategories					Display item categories
BrowseRegions					Display geographic regions
PrepBuyNow		✓			Display info for a BuyItNow purchase: item details, price, seller details
PrepBuyNowAuth					Login for PrepBuyNow
PrepBid		✓			Display info for placing a bid: item & seller details, current max bid, #bids
PrepBidAuth					Login for PrepBid
PrepComment					Display form to comment on a user
PrepCommentAuth					Login for PrepCommentAuth
PrepRegisterItem					Display form for starting new auction
RegisterItem	✓			✓	Initiate an auction w/supplied details
RegisterUser	✓			✓	Adds a new user to user database
SearchByCategory		✓			Display item details matching category
SerachByRegion		✓			Display item details matching region
StoreBid	✓			✓	Record a bid and update auction DB
StoreBuyNow	✓			✓	Record a BuyItNow and update DB
StoreComment	✓			✓	Record a comment and update user's feedback score/profile
ViewBidHistory		✓			Display bid history for an item
ViewItem		✓			Display item details
ViewUserInfo		✓			Display user details & feedback profile
AuctionDone		✓		✓	Logged when auction finishes and winner/seller are notified

Table 7-1: Verbs for the RUBiS auction service. The RUBiS protocols define 26 interactions, of which 20 generate non-static responses. These 20 interactions plus the meta-verb AuctionDone (in gray) define the set of verbs for our undoable auction service. All auction verbs are synchronous, and the majority are neither logged nor replayed, as discussed in the main text.

using the AuctionDone verbs' paradox detection predicates and any resulting paradoxes handled by the verbs' compensation methods.

An alternate approach would be to modify the service itself to generate the AuctionDone verbs. During replay, completion of any active auctions would be suppressed until the corresponding AuctionDone verb was replayed, at which point the auction would be allowed to complete, generating a new AuctionDone verb that could be compared with the original using the standard paradox detection and handling approach. This second approach is simpler and easier to understand, but requires that the service be modified to suppress auction completion and to generate the AuctionDone verb.

Finally, notice that there are no asynchronous verbs listed in Table 7-1: all verbs immediately report their status back to the end user. This is in contrast to our e-mail case study, where we leveraged the asynchrony of the SMTP deliver verb to open up a window for undo (see Section 6.3.1). While asynchronous verbs increase the opportunity for undo by providing a window in which problems can be repaired without users noticing any resulting paradoxes, for auctions the lack of asynchronous verbs does not mean that undo is useless. This is because, while the *verbs* may all be synchronous, the *auctions* themselves are not. The STOREBID verb that places a bid may reflect its success or failure back to the user immediately, but the user does not know what the end result of that bid will be until the auction closes. Even if the user looks at the current top bidder immediately after placing a bid, events in the future can change the effect of their bid—another user could outbid them, or another user's later outbid could be retracted, restoring the first user to top bidder status; even eBay provides a bid retraction mechanism that can reactivate an earlier dormant bid. Thus, there is certainly opportunity for using undo to repair problems that affect auction bidding: while the bidding verbs are synchronous, their ultimate effects are tentative until the auction completes, providing a window for undo.

7.1.1 Verb implementation issues

Context, one of the major implementation challenges for e-mail undo, is less of a problem in auction services since auctions (in RUBiS and typically in most online auction services) have unique and unchanging identifiers that can be directly used as UndoIDs. Also, there is no need to strip and translate connection context, since auction protocols consist of isolated interactions with the service with each interaction using its own independent connection.

Sequencing is also simpler in auction systems than in e-mail. There are no long-running verbs, and all verbs that view or update state are tied to specific auctions that are independent of each other. All that is required is for ordering to be preserved between verbs that update or view the same auctions, identified by auction ID. Verbs that update or view user information or item state are likewise sequenced by the associated user or item IDs. We do allow verbs corresponding to different items or auctions to bypass each other, which can cause some global reordering of bids across different auctions during replay, but each auction itself will behave consistently during replay as during original execution.

7.2 Handling Paradoxes in Auctions

Again, as with e-mail, we must consider what situations constitute paradoxes for an auction service, how to detect and compensate for those paradoxes, and how to manage propagating paradoxes.

We are lucky with auctions in that the typical design of auction services makes paradoxes much less likely than in a service like an e-mail store. Recall that paradoxes occur when *end-user-visible* state changes as the result of a retroactive repair or rollback of *system-level state*. In auction services, there is very little system state that can affect how end user interactions (namely, auction bids) are processed; there are no pluggable and frequently-updated filters or transformation routines that intercept user interactions and affect their processing, unlike in e-mail where we had to contend with spam/virus filters and mail routing tables. Unless the undo or retroactive repair alters the auction's bidding algorithm implementation, or bidding support routines like currency conversions, it is unlikely that paradoxes will occur during replay. In other words, recalling the nested spheres-of-undo model of the Three-R's from Section 3.1, the outer sphere in an auction system contains little state that is likely to affect the correctness of the inner sphere.

More typically, it seems likely that auction services will be affected by dependability problems that affect a user's ability to execute new interactions (or generate verbs). Corrupted state, misconfigurations affecting performance or availability, failed upgrades, crashes, and partial service failure are all examples of failure modes that can be repaired using undo while not creating paradoxes, unless they happen to alter the algorithms for bid processing. Even if a failure prevents users from performing tasks like bidding and starting auctions, the lack of true asynchronous verbs in an auction service means that the resulting errors will be immediately reflected to users, will not be incorporated into the timeline, and will therefore not have a chance to cause paradoxes upon replay. Most of the observed failure modes of the eBay auction site seem to fall into this class of non-paradox-generating failures, including the famous multi-day outage of 1999 [35].

Of course, using undo to recover from some of these non-paradox-producing problems may still require some form of compensation for the auction system. For example, to placate its users eBay typically extends the length of open auctions when they have been unavailable due to a dependability problem. Since this sort of compensation does not result from a paradox, it cannot be captured in our verb-based compensation framework. It could, however, be implemented in the auction service proxy: the proxy could hook the undo manager's time-travel-notification callbacks, and automatically extend open auctions when an undo cycle commits.

Even though paradoxes are likely to be rare given the failure modes of an auction service, they are still possible, and thus we still need to detect and compensate for them. As with e-mail, we will base our approach to paradox management on an external consistency policy that defines what end users of the auction service should expect in terms of consistency during normal operation and across undo cycles.

7.2.1 An external consistency policy for auctions

Unlike e-mail users, external users of an auction service take on different roles as the auction system evolves over time. Some auction service users are **bystanders**, only looking at item listings, perhaps following auctions, but never starting auctions or placing bids. Other users are **participants**, and are involved in one or more active (uncompleted) auctions, either having started an auction by offering an item for sale, or having bid on an auction. A subset of participants are **stakeholders**: they are either sellers or the current top bidders in an auction. Note that now and for the rest of this section, we will consider "Buy It Now" sales to be a special form of single-bidder auction that completes immediately when the item is bought, and so we will not treat such sales separately.

Changes made during an undo cycle can cause external inconsistencies that affect all three classes of users. Such inconsistencies include whether a certain user has won an auction or not, the winning bid in the auction, the specifics of the bid history of an auction, whether particular auctions are valid or not, changes to the item descriptions for an auction, or changes to a user's rating or personal data.

These inconsistencies become paradoxes only when they affect stakeholders or when they elevate or demote a user to or from a stakeholder role. We allow bystanders to see inconsistencies following an undo cycle—items may change description, top bidders may change, and so on—without any compensation, since they have no stake in the auctions whose particulars have changed. We even allow non-stakeholder participants to see inconsistencies, as long as those inconsistencies do not affect a participant's stakeholder status, again because a non-stakeholder participant has already been outbid and no longer holds a stake in an ongoing auction. We take the risk of not managing these inconsistencies as paradoxes both to keep the undo system simpler and reduce the potential amount of post-undo compensation; we feel that it is an acceptable compromise given that non-stakeholders cannot be financially affected by changes in the operation of the auction system.

For stakeholders, however, inconsistencies do cause paradoxes that must be managed; similarly, if a stakeholder is demoted to a vanilla participant or a participant is elevated to stakeholder, the change in role represents a paradox that also must be managed, since a financial stake is involved. Thus, paradoxes occur for stakeholders in the auction system when:

- they are demoted to regular participants by losing top-bidder status as a result of undo cycle changes
- the items for which they are stakeholders are altered (description or price changes)
- the winning bid changes for auctions they have won, or their winner status for any completed auctions changes
- the seller's information changes for any auctions in which they are top bidder
- an ongoing or completed auction in which they are the top bidder is cancelled.

The only other paradox-generating scenario in the auction system is when a non-stakeholder participant becomes the new top bidder for an auction due to undo-induced changes.

A key point to notice is that, unlike in e-mail, we declare the above situations to be paradoxes *regardless of whether the stakeholder has observed the relevant state*. So, even if the stakeholder never looks at the item description, for example, he or she is informed of paradoxes that result if the description changes as a result of undo. We believe that this is a reasonable policy choice for auctions for several reasons. First, inconsistencies can potentially have financial implications for stakeholders, and therefore they deserve to be notified of them. Second, paradoxes in auctions are likely to be rare, especially since there are not many stakeholders relative to bystanders and ordinary participants, so slightly overzealous compensation will not unduly burden the system or its users, unlike with e-mail. Third, with this policy choice we do not need to record every verb that externalizes state, since we do not care whether state relevant to a stakeholder has been observed. Instead, we limit our log to verbs that modify state (such as STOREBID and AUCTIONDONE); this reduces the size of the timeline log and speeds up replay.

Managing auction paradoxes. How we manage the paradoxes listed above depends primarily on whether or not the auction in question has completed. For auctions that have not completed, we can use an approach similar to the one we used in our e-mail system: we can notify the stakeholders involved by e-mail and explain to them that the auction details have changed and why. In particular, if the top bidder for an auction has changed, the original and new top bidders must both be notified. We believe that notification is sufficient compensation for ongoing auctions, since even when the top bidder or bid changes, the resulting bids will never be more than the maximum value specified in the bidder's original proxy bid submission, and therefore, regardless of who ends up the top bidder, they will not pay more than they were originally willing to pay. If an item's details change as the result of undo, and the top bidder wants to back out of the auction as a result, the bidder can rely on existing auction mechanisms for bid retraction or auction annulment; most consumer-targeted auction services already offer such mechanisms to be used in the not-so-uncommon cases where sellers are found to be lying.

Finally, for many auction services, it may make sense to also extend the length of any affected auction that is near its completion time, to allow the auction's stakeholders to read and react to the notification messages. eBay already uses this auction-extension approach to placate its users when it suffers downtime or dependability problems, so adapting it for use as part of paradox compensation is a reasonable approach.

Auctions that have completed (and the related "Buy It Now" sales) pose somewhat more difficulty for paradox management, since completed auctions often involve an exchange of physical goods and money, neither of which are easily retracted. The worst case is when it is discovered upon replay that the wrong user was selected as the auction winner; slightly less bad is when the right user won but for the wrong price or for an item with the wrong description.

There are no truly satisfactory solutions for either case, and in practice the providers of the auction service will have to make a policy choice on the spectrum between declaring all completed auctions final and absorbing the financial costs to make things right for the involved users. In the former case, no paradox management is necessary other than to discard the results of the replay execution in favor of the original execution's results. In the latter case, if the winning bidder underpaid for the item due to a failure in the auction service, the service provider could pay the seller the difference, writing off the loss as a cost of doing business. Luckily, it is unlikely that a paradox could involve a bidder paying more than their maximum bid in a completed auction, so bidders themselves should never have to be recompensed for overpaying.

Finally, if the wrong bidder is selected as winner due to a problem with the auction service, again the service provider has little recourse but to possibly compensate the seller for lost profit, to possibly also compensate the bidder who should have won, or to offer the buyer and seller the option of cancelling their transaction and re-running the auction. For auction services that use escrow to finalize their completed auctions, a more satisfactory solution to the wrong-bidder or wrong-item problems might be to cancel the transaction (and later re-run the auction) up until the point where the escrow process completes, after which the auction results are final; this approach offers a window of undo that matches the length of the escrow process.

7.2.2 Implementing the external consistency policy

We can implement the consistency policy just described using only six of the verbs listed in Table 7-1: REGISTERITEM, REGISTERUSER, STOREBID, STOREBUYNOW, STORECOMMENT, and AUCTIONDONE. We get away with not recording any of the other verbs because we do not care whether inconsistencies are actually externalized when declaring paradoxes; instead, we treat each state-altering verb as if it immediately externalized its results. Thus, the tags of the state-altering verbs capture any potential external inconsistencies, and provide enough information to detect and manage paradoxes.

The greatest challenge in implementing the consistency policy of Section 7.2.1 is that it requires knowing ahead of time whether an auction has completed or not, and whether a user generating a verb (*e.g.*, by submitting a bid) is going to end up a stakeholder or not by the end of the undo cycle. We need to make these distinctions because we only want to declare paradoxes and compensate for verbs submitted by those users who are stakeholders at the end of the undo cycle.

To deal with the distinction between completed and uncompleted auctions, we must defer all compensations until we know whether or not the auction has completed—that is, until we either see an AUCTIONDONE verb or until the undo cycle commits without an AUCTIONDONE verb being encountered. A similar approach works for distinguishing stakeholders from non-stakeholders: we defer sending explanatory compensation until we know which users are stakeholders at the end of the undo cycle.

Mapping this deferred compensation back to verb-specific paradox detection and compensation predicates results in a rather different approach than the one we took for e-mail. For auctions, we would use the verb paradox detection predicates to identify *potential* paradoxes—paradoxes that would exist were the verb's user a stakeholder. The per-verb compensation routines do not perform the actual compensation, but rather queue enough information in the proxy so that the compensations can be carried out later. The accumulated information concerning potential paradoxes could be exposed the system's operators, giving them an indication of the paradox-related consequences of the undo cycle should they choose to commit it. When and if the operator chooses to commit the undo cycle, the proxy can invoke a global compensation method that uses the queued potential paradox information to identify true stakeholders and completed auctions; the compensation method can then discard compensations destined to non-stakeholders and can perform the remaining compensations as appropriate based on each auction's completion status.

As an example of how this approach might be implemented, consider the case of detecting and compensating for paradoxes in an auction's top bidder, top bid, and related information like item description and seller information. All of these paradoxes can be managed through the conjunction of the STOREBID and AUCTIONDONE verbs (and STOREBUYNOW, handled analogously to AUCTIONDONE and thus not discussed further). As mentioned above, we treat STOREBID as if it were externalizing, so we assume its tag contains a record of the top bid and top bidder after the new bid has been processed, along with a hash of the item description and the seller's information (all of which can be obtained by querying the auction service when the bid verb is executed). We also assume that AUCTIONDONE records a hash of the item description and seller information along with the final selling price and winner.

To track *potential* paradoxes arising through STOREBID verbs, the proxy must maintain a table with an entry for each auction encountered in the log; the entry should contain a reference to the latest inconsistent STOREBID verb for that auction, if any. Then, when a STOREBID verb is replayed, the verb's paradox detection predicate compares the tag of the original logged verb with its replay tag, and a paradox is declared if any differences are found in the top bidder, top bid, item description, or seller information. The compensation routine for STOREBID paradoxes looks up the associated auction in the proxy's table, and either adds a new entry or replaces the current entry with a reference to itself. It does not perform any other compensation that might be visible to the end user. If no paradoxes are detected, the paradox detection routine sets the entry in the proxy table to a special value indicating no paradoxes.

Thus, at any point in the replay process, the proxy's table indicates whether there is an uncompensated-for paradox for each seen auction, and contains a pointer to the STOREBID verb that detected the paradox, if any. This verb contains the identities of the current stakeholders for that auction, recorded as the top bidders in the original and replay copies of its tag.

If an auction does not complete within the undo cycle, its entry will remain in the proxy's table when the undo manager signals the end of the cycle. At this point, the proxy knows who the stakeholders are for each incomplete auction: they are the top bidders in the STOREBID records pointed to by the entries in the proxy's potential-paradox table. So, for every entry in the table, the proxy can send the appropriate compensating messages to those stakeholders using the information in the referenced STOREBID verbs.

If, on the other hand, an auction completes during replay, then a corresponding AUCTIONDONE verb will be replayed. The AUCTIONDONE verb's paradox detection predicate performs the same comparisons as the STOREBID predicate. Paradoxes detected by this verb are real, not potential, and thus the AUCTIONDONE verb's compensation method can carry out the task of informing the original winner (and potentially new winner) of any discrepancies in the auction, and invoking whatever other compensation policy the service provider has selected for paradoxes in completed auctions. Regardless of whether it finds paradoxes, however, the AUCTIONDONE verb should remove its corresponding entry from the proxy table (if any), to indicate that the auction has been handled.

The remaining three verbs, REGISTERITEM, REGISTERUSER, and STORECOMMENT only generate paradoxes if they fail during replay. These paradoxes can be handled immediately by sending their invoking users explanatory e-mails detailing the failure and suggesting that they try the operation again. The consequences of a failed STORECOMMENT to a seller's rating are captured by the STOREBID paradox handling discussed above. A failed REGISTERUSER verb will cause later bids submitted by that user to fail, but these failures will be treated as if the bids were not entered, and the resulting paradoxes (if any) will also be handled via the STOREBID mechanisms discussed above. The further consequences of the REGISTERITEM verb are handled by the propagating paradox routines discussed in the next section.

7.2.3 Propagating paradoxes

Dependencies across verbs in an auction service take two forms. In the first case, a user's bid on a particular item may depend on the history of bids already made by other users. These dependencies are only visible in the user's mind, however, and may represent complex reasoning invisible to the undo system. To deal with these dependencies, we take the approach that a user's bid represents a willingness to pay a certain amount for an item (*i.e.*, we assume that users should be bidding at most their true valuation for the item), and should therefore be valid regardless of the prior bid history. Under this assumption, we assume that paradoxes affecting one bid (one STOREBID verb) do not propagate to later bids or verbs. We make the same assumption about inter-auction dependencies, and thus paradoxes from AUCTIONDONE or STOREBUYNOW verbs also do not propagate. Also, as mentioned above, the consequences of paradoxical STORECOMMENT and REGISTERUSER verbs are handled via existing paradox-detection mechanisms, and thus these verbs also do not have dependencies and therefore do not cause propagating paradoxes.

The second case of true dependencies in an auction system is between the REGISTERITEM verb and later verbs. Since REGISTERITEM verbs start auctions, their failure affects all future verbs that concern that auction. These dependencies would be picked up using our undo architecture's standard mechanism of commutativity-based dependency discovery, and thus the future verbs will all be squashed by the undo manager. In response to being squashed, later STOREBID and AUCTIONDONE verbs can follow the same sort of procedure used for detected inconsistencies, using an extended version of the proxy's tracking table to identify the appropriate stakeholders, and later notifying them that the auction has been cancelled. If a completed auction is cancelled—that is, if an AUCTIONDONE verb is squashed—the same policy choices face the auction service provider as with inconsistent completed auctions.

To implement these approaches to propagating paradoxes, the StoreBid, AuctionDone, and StoreBuyNow verbs must define squash routines. Since squash routines are called with the identity of the non-commutative paradox-producing verb that caused them to be squashed, they can check if that verb is a RegisterItem verb. If not, the squash routine does nothing; if so, the squash routine takes the appropriate steps to register a potential paradox (for StoreBid) or to compensate for the paradox (AuctionDone/StoreBuyNow). The remaining three verbs should define no-op squash routines.

7.2.4 A recovery guarantee for auctions

The recovery guarantee for an undo-enabled auction service is straightforward: any stakeholder in an ongoing auction will be notified of undo-induced changes to that auction's item state and auction state, and to the user state of the auction's seller. If the top bidder for an auction changes as a result of undo, both the old top bidder and new top bidder will be notified of the changes. No bidder will end up with a bid higher than his or her maximum specified bid. The same notification guarantees hold for completed auctions, although in this case more compensation might need to be performed to placate the seller (or correct buyer), or alternatively the provider may choose to suppress the notification for completed auctions, adhering to a policy that completed auctions are final.

7.3 Discussion

Our design for an undo wrapper for auction services highlights the flexibility of our verb-based undo architecture, especially when set in contrast to the e-mail service undo wrapper described in the previous chapter. While both e-mail and auction services share the same fundamental design point of a self-contained service with human end users, they take very different approaches to external consistency and paradoxes. In e-mail, paradoxes were declared only when the undo cycle induced changes to state that had already been seen by an end user, and all such visible changes ended up as paradoxes. In auctions, on the other hand, paradoxes exist only for a small subset of the service's users—the stakeholders—but for that group they occur upon *any* undo-induced change, whether previously-visible or not.

Both of these paradox detection policies were easily expressed in the verb-based framework we developed in Chapter 4; the mapping was slightly more natural in the e-mail case where every paradox gets immediate compensation, but the auction case shows that even more complicated deferred and selective compensation policies can be implemented with only minor additional complexity in the proxy.

In fact, our auction design suggests that even fully state-based paradox management approaches may be possible in our framework. A fully state-based approach is one in which paradoxes are detected by comparing the pre- and post-undo system state without regard to the verbs that formed it; our auction design is a variant on this approach, effectively comparing the pre- and post-undo end states of all active auctions in order to identify and compensate for stakeholder paradoxes only. Yet, despite its state-based flavor, we were still able to map this policy to our verb-based management framework, requiring only an extra table in the proxy to track the last bid verb that updated each auction's state. Similar approaches may work in general for state-based external consistency policies.

Finally, the auction case also demonstrates how our generic paradox-management architecture scales in terms of efficiency with the requirements of the underlying service. In the auction case, our paradox policy did not require that the undo system track each time a piece of state was externalized, unlike with e-mail; we were able to take advantage of this property by designing the verb set for auctions to minimize the amount of timeline logging needed, reducing the overhead of undo and potentially improving replay performance greatly.

Chapter 8

Guidelines for Constructing Undoable Self-Contained Services

> "Irreversibility is the effect of the introduction of ignorance into the basic laws of physics."
>
> — *Max Born*

Our case studies of e-mail and auction services have illustrated how the service-neutral undo architecture of Chapter 4 can be adapted to provide operator undo functionality for two rather different self-contained services. In both cases, however, we stumbled over roadblocks in the designs of the underlying services, having to work around issues like inadequate naming, long-running verbs, and meta-verb events.

Many of these roadblocks—and their limiting workarounds—could be removed by redesigning the services and protocols to integrate undo functionality directly into the application code of the services themselves. Doing so, however, would violate our black box assumption and obliterate the benefits of building undo as a service wrapper, namely fault isolation, independence from application upgrades, and broader recovery capabilities including OS-level recovery. A more appealing alternative is to keep the undo functionality in a service wrapper, but then redesign the service and its protocols with an eye to undoability, essentially removing the roadblocks to undo without actually implementing undo within the service itself.

To encourage service designers to carry out such redesigns, and to aid new service designers in thinking about the potential undoability of their services and protocols, we have developed a set of guidelines for building or redesigning a service with an eye to its eventual undoability. Several of these guidelines match the way services are commonly designed today, and so we label them as **common practice** guidelines. The remaining **undo-specific** guidelines are less likely to be intentionally followed by existing services, yet are crucial to successfully retrofitting a service with Three-R's undo functionality. Following these guidelines during service design or redesign ensures that the task of supplementing the service with an undo wrapper will be simpler, faster, and result in a more efficient undoable system.

The guidelines that we develop in this chapter are once again targeted at self-contained services. Our guidelines are focused at the service implementation level; we assume throughout that the high-level service semantics are compatible with Three-R's undo, namely that they support paradox detection and compensation as discussed earlier in Chapter 4, Section 4.6.2.

We will begin our discussion by considering the guidelines specific to undo, then will briefly enumerate the guidelines that fall under common practices for service design. Some of our guidelines will be mandatory, and must be met for undo to be possible; we will indicate these with the word *must*. Others help simplify the task of building the undo system's verbs and proxies and are advisable for obtaining the best undo behavior and performance, but can be set aside if necessary; these will be introduced with the word *should*.

8.1 Undo-Specific Service Design Guidelines

Our first collection of design guidelines encompasses the eight fundamental properties that a service's implementation needs in order to be made undoable. They divide into three categories, which we will consider in turn: two guidelines concerning the service's structure, five concerning its protocol design, and one concerning undo cycle performance.

8.1.1 Service structure

The first step in making a service undoable via the architecture of Chapter 4 is to ensure that its state is self-contained. When state is self-contained, it can be installed as a cohesive unit on a rewindable storage device, thereby making it possible to rewind the entire service coherently to a prior point in history. As a guideline, then, we require the following:

1. **self-contained state**: all persistent service state must be stored within a single state store, including system-level state (OS binaries, system-level configuration) and service-specific state (application binaries, application-level configuration, user data), and excluding state unrelated to the service.

Recalling our original definition of a self-contained service in terms of spheres of undo, this guideline states that the service's persistent state must be captured within a single, non-overlapping sphere of undo. Consequently, it requires that the service's state be bounded, and that those boundaries be known to the undo system designer. Note that this mandatory guideline can be met by services that keep their state in a distributed or shared state store, as long as those stores provide a logical view of a single, indivisible storage container holding only the service's persistent state.

The self-contained state guideline is not met when a service's state boundaries cannot be clearly identified. For example, this can happen when two distinct services store their service data in a common, shared database instance; in this case, one service cannot be rewound without affecting the other. Distributed services or services that are formed from an agglomeration of multiple independent sub-services are also examples where the self-contained state guideline fails to be met.

The second key structural guideline for design of an undoable service is that it be possible to interpose the undo system's proxy between the service and its clients in such a way that the proxy can collect all the information it needs to generate the timeline log of verbs and then later replay it. More specifically:

2. **possible to proxy**: it must be possible to intercept every communications flow crossing the boundary of the service's sphere of undo, to identify the boundaries of intercepted requests, and to inject new commands into the service.

This guideline requires that the service designer have an exhaustive list of all APIs that alter or expose the service's self-contained state, and that all those APIs can be intercepted.

There are several ways that a service could fail to meet this guideline. First, it could use back channels (like files on a shared disk) to communicate with external entities, resulting in sphere-of-undo boundary crossings that cannot easily be intercepted and proxied. It could use an end-to-end authentication protocol like SSL to authenticate and encrypt client connections, in which case the undo proxy could not intercept the communications flow without cooperation from the service. One way to accomplish this cooperation would be to have the proxy assume the identity of the server and handle user authentication itself, using a pre-authenticated secure connection to pass requests on to the service.

The proxy guideline could also be violated if the service has requests that have no clear boundaries. For example, some requests might execute asynchronously on the service, appearing from the proxy's point of view to complete before they actually affect service state. Without an accurate understanding of request boundaries, the proxy cannot properly sequence verb execution and may not be able to generate a consistent timeline log. In such cases, the service must provide either an asynchronous notification of the completion of such requests, or at least a means of querying whether they have completed.

Another type of problematic request without a clear boundary is a request that generates a continuous stream of asynchronous "push" output. If it is possible to intercept this output and correlate it with the verb that instigated it—essentially treating the pushed output as a delayed response to an earlier request—then the proxy guideline can be met. For example, an order confirmation e-mail in a shopping service can be seen as delayed output from the original ORDER command. Alternately, if the push output cannot be viewed as a delayed response to an earlier request, then the undo wrapper will have to treat it as the output of a null verb or meta-verb (like the AUCTIONDONE verb in our auction service). In this case, the push output must contain enough information to tie it to specific pieces of user state, so that appropriate commutativity and paradox tests can be defined for it.

8.1.2 Service protocol

Once state containment has been established, the next issue for a service designer is the protocols that the service uses to interact with external entities. The protocols play a key role in determining a service's potential for undo, since they define the undo wrapper's vantage point into the workings of the service. If the protocols are inadequate, the undo manager will not be able to intercept and log the service's interactions with end users, and will not be able to collect enough information to properly detect and manage paradoxes.

There are five key guidelines for building an undo-compatible protocol:

3. **completeness**: the set of commands defined by a service protocol must be complete: for each command that names or alters a piece of state, the protocol must provide a command to retrieve the current value of that state, and vice versa.

4. **transparency**: protocol commands should be as transparent in their parameters and effects as possible. The protocol command's parameters should explicitly specify its behavior, without reliance on implicit context; wildcard parameters should be avoided.

5. **naming**: any service state accessed, altered, or deleted by a protocol command should be explicitly named with a time-invariant unique name in either the request or response. When state is created, the protocol should provide a means for assigning a specific name to that state.

The completeness, transparency, and naming guidelines ensure that the proxy can capture all the information needed to replay protocol interactions and detect and manage paradoxes. Completeness ensures that any state altered by a command can be retrieved to check it for paradoxes, and that any state retrieved by a command can be reset during replay. Transparency simplifies the task of defining verbs and optimizes their behavior, by ensuring that protocol commands expose all the information needed to construct a verb. Without transparency, the proxy has to make additional requests to the service to expose implicit context, adding overhead and potential for inconsistency. The naming guideline helps avoid the complexity of having to create and maintain shadow UndoIDs in the proxy wrapper, as we had to do for e-mail.

6. **deferred reporting**: service protocol commands should synchronously report as little information about their effects as possible, instead deferring that reporting to separate commands that query the effects. As a special case, if the capability for retroactive repair of failed user interactions is desired, the service protocol commands corresponding to those interactions must not immediately report their failure status back to their initiators.

The deferred reporting guideline helps minimize paradoxes by forcing clients to explicitly ask for information concerning a command's execution, rather than needlessly exposing it to potentially-uninterested clients. This approach helps expose user intent to the undo system, as clients should only ask about information that they care about. Furthermore, the deferred reporting guideline governs the potential for retroactive repair of problems that incorrectly cause protocol commands to fail. In order to retroactively repair failures of protocol interactions, those failures cannot have been exposed to the initiators of the failed interactions. Deferring failure reporting allows a window for retroactive repair, particularly if the client never attempts to check the status of the failed interaction. We saw an example of how this kind of deferral might be implemented in our e-mail case study, where we delayed SMTP bounce messages by four hours to provide a window for retroactive repair of delivery problems.

7. **brevity**: commands that alter or read volatile state should be as short as possible in terms of running time to avoid the problems of long-running verbs discussed in Section 4.4.2.

Finally, the brevity guideline discourages long-running verbs, helping to prevent scheduling conflicts when sequencing verbs for execution. As a special case of this guideline, if a command requires or generates a significant amount of data, it should be designed so that it can be split by the proxy into two phases, one in which the I/O is performed with the client, and another shorter phase in which the state-altering or -accessing command is actually executed with the service.

8.1.3 Undo cycle performance

Finally, we wrap up our undo-specific guidelines with one targeted at increasing the performance of the Three-R's cycle:

8. **optimized replay interface**: the service should provide a dedicated replay interface that bypasses the usual client authentication protocols and supports command batching.

With an optimized replay interface, the undo proxy can simply funnel a properly-sequenced verb stream into the service, without having to pay the overhead cost of setting up and tearing down simulated client connections or of authenticating and masquerading as each different end user represented in the verb stream. Furthermore, a well-designed replay interface can reduce the need for convoluted replay parallelization code in the undo proxy.

8.2 Common Practice Service Design Guidelines

Besides the guidelines discussed in the prior section, there are five other guidelines that must or should be met by a service destined for use with a Three-R's undo system. However, these guidelines represent common practice in service design, and therefore are likely to be met already by most well-designed existing and new services. As such, we focus our discussion on their implications for an undoable system, rather than rehashing their well-known implications for service design.

8.2.1 Recovery

The first two common practice guidelines concern the service's recovery design, and are met by necessity in most commercially-deployed services:

1. **write-through**: before acknowledging the result of a state-altering end user request, the service must ensure that the corresponding state changes are persistent and recoverable.
2. **fast restart**: the service must be able to completely regenerate or refresh its transient and soft state from its persistent state store. As an optimization, the restart procedure should be as rapid as possible.

These two guidelines are often met by existing services as a way to provide fault tolerance, rapid recovery from transient errors, and prophylactic reboots [18]. They are mandatory for an undoable service, since our black-box approach requires that time travel take place only at the level of persistent state; the two above guidelines ensure that the service's transient and soft state remain synchronized with its time-travelling persistent state.

8.2.2 Service protocol

Our final three common practice design guidelines concern the service's protocol design. Specifically:

3. **request-response structure**: the service's protocol must have a command-based, request-response structure, with individual commands that manipulate and/or retrieve identifiable state objects at a granularity that reflects the intent of the end user.

4. **well-behaved commands**: protocol commands must be deterministic, such that different executions of the same command with the same parameters in the same system context must always produce the same results, or at least acceptably consistent results from the point of view of the service's users.[1] Commands must either complete successfully or fail cleanly, without leaving partial or incorrect changes to service state.

5. **idempotency**: there is no requirement that commands be idempotent. However, if *all* commands are idempotent, or provide a way to empirically determine if they have been executed, then the overhead of taking snapshots and of synchronizing the service with the timeline log can be reduced, and the restrictions of the write-through guideline (#1, above) are not needed. Idempotency also implies the same properties attached to the well-behaved commands guideline.

The request-response and well-behaved command guidelines simply ensure that the protocol used by the service is well-formed and well-behaved. They make it possible for the undo system to record operations and replay them later with consistent results. Any reasonable service should meet these guidelines. The idempotency guideline can simplify the coordination between service and undo proxy by allowing the proxy's timeline log to run ahead of the service itself; this optimization also helps minimize the overhead of the undo proxy during normal operation.

8.3 Discussion

The thirteen guidelines presented above provide a starting point for designers of new services who want to provide for the later possibility of wrapping those services with Three-R's undo functionality. Moreover, they also can be treated as a set of criteria for evaluating existing services or service implementations to determine what changes, if any, need to be made before the service can be made undoable. If the service implementation violates any of the "must" guidelines, the designer has little choice but to modify the service or its protocol to bring it into compliance with the guidelines, or else must forego undo functionality or accept that unmanaged paradoxes will occur, depending on the severity of the violations. On the other hand, violations of the "should" guidelines can be handled in either the service itself, or, with a bit more complexity, in the undo system wrapper; we saw several examples of the latter approach in the case study of the e-mail store service, such as the workaround used to map unique UndoID names to the insufficient IMAP names.

To give a sense of how well existing services meet the guidelines for undoability, we have evaluated the IMAP/SMTP e-mail store service and the RuBiS-based auction service against the 13 guidelines discussed in this chapter. Table 8-1 shows the results of this analysis. As can be seen in the table, the auction service meets more of the guidelines for undoability "out of the box," lacking only the optional optimized replay interface and idempotent commands. In contrast, the e-mail service requires a few direct modifications to the service implementation to meet the mandatory criteria, as we have

[1] Acceptable consistency is defined by the same relaxed consistency model used when defining paradox detection predicates for the service's verbs.

Guidelines for Undoable Services			Mandatory?	E-mail Store Service	Auction Service
Undo-Specific	**service structure**	self-contained state	✓	✓	✓
		possible to proxy	✓	after modifications	✓
	service protocol	completeness	✓	✓	✓
		deferred reporting	✓	after modifications	✓
		transparency			✓
		naming			✓
		brevity			✓
	undo cycle performance	optimized replay interface			
Common Practice	**recovery**	write-through	✓	✓	dependent on implementation
		fast restart	✓	✓	
	service protocol	request-response protocol	✓	✓	✓
		well-behaved commands	✓	✓	✓
		idempotency			

Table 8-1: Evaluation of e-mail store and auction services against guidelines for undo. The table shows how well the IMAP/SMTP-based e-mail store service and the RuBiS-like auction service (studied in Chapters 6 and 7, respectively) match the guidelines for undoable services defined in this chapter. The auction service meets more guidelines than the e-mail store service and thus can be wrapped with undo more easily. The e-mail service required service modifications to provide notification of asynchronous verb completion (making it possible to proxy) and to delay reporting of failed SMTP interactions (providing deferred reporting), and further required extra complexity in the undo proxy layer to overcome the un-met non-mandatory guidelines.

already discussed in Chapter 6: a means of deferring SMTP bounce messages to provide deferred reporting, and a means of querying whether SMTP deliveries have completed.

Even with these modifications, however, the e-mail service still does not meet many of the non-mandatory guidelines for optimal undoability. This fact explains why the undo wrapper for the e-mail store service is so complex: the non-mandatory guidelines had to be met by inserting extra functionality into the verbs and proxy to provide missing functionality like unique names and to handle implicit context and long-running interactions.

In summary, our mandatory guidelines give service designers an idea of what changes are needed to make an existing service implementation undoable, while our non-mandatory guidelines offer suggestions for making changes to the service to simplify the task of designing its undo wrapper. For services that cannot change, the guidelines become a set of criteria that can be used to evaluate both the service's potential for undo and the cost of making it undoable. Services that do not meet the mandatory guidelines cannot be made undoable without modification; the implementation complexity of adding undo to a service that does meet the mandatory guidelines is directly related to the number of non-mandatory guidelines that are not met.

Chapter 9

Extending Undo to Hierarchical Services: Nested Spheres of Undo

> "Every seeming equality conceals a hierarchy."
>
> — *Mason Cooley*

Up to this point, we have focused on providing undo for monolithic, self-contained services, defined by a single sphere of undo. This architecture covers a reasonably significant set of applications, as illustrated by the case studies in the previous chapters. It can even work for applications with more complex architectural structures, as long as they can be treated as a single self-contained sphere of undo like the one that Figure 9-1 depicts. For example, an e-mail store service like the one in Chapter 6 could be seen as a collection of smaller per-user e-mail store services, but we were able to successfully treat it as a monolithic block. Likewise, at an application service provider (ASP), several distinct services running on the same machine and using the same state store could be treated as a single mega-service with one self-contained sphere of undo.

We say that services like these examples are **hierarchical**: from the outside perspective of an undo wrapper, they can be treated as a unified collection of state defining a single sphere of undo, but internally one could imagine decomposing them into a hierarchical structure of nested state stores, each of which might otherwise define its own sphere of undo. A key defining characteristic of hierarchical service architectures is that they typically cannot be teased apart into multiple independent services with their own unique state stores, either because the subservices are coupled by shared state (as in the e-mail service example) or by virtue of sharing the same underlying hardware/software execution platform (as in the example of an ASP running multiple services on the same machine sharing a single operating system instance).

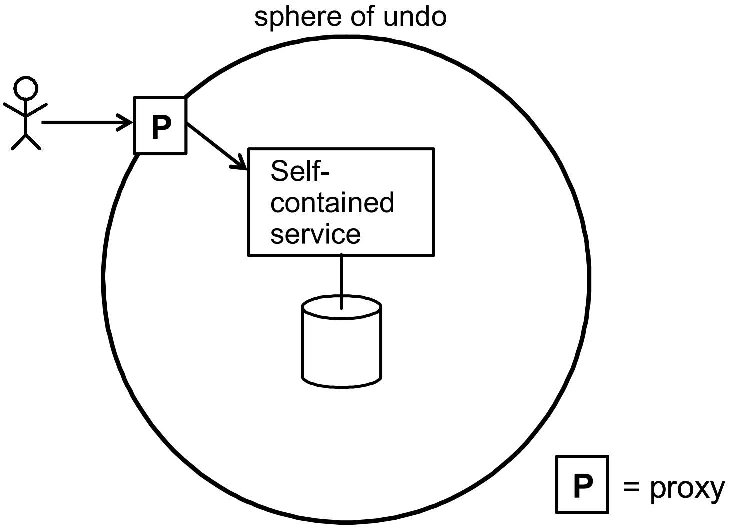


Figure 9-1: Self-contained single-service architecture. A self-contained service encompasses a single sphere of undo that communicates only with human end users. Requests from those end users are proxied, logged, and replayed at the sphere boundary. This figure uses a schematic graphical representation for spheres of undo that we will use and extend throughout this and the following chapters.

If we want to provide undo for hierarchical services, we must either stick with the approach we have developed in the previous chapters, shoehorning the hierarchical service into a single monolithic sphere of undo, or extend our undo model to explicitly support hierarchies of state; no option exists to separate the subservices into their own independent spheres of undo. While the shoehorning approach is possible, it has significant drawbacks in terms of overhead: to apply Three-R's undo to only a portion of the hierarchy—such as a single service on a multi-service machine or a single user's e-mail in an e-mail store service—the entire service hierarchy must be rolled back and replayed, requiring much more work than is necessary to recover the desired subset of state.

Thus, in this chapter we will explore the possibility of extending Three-R's undo to support hierarchical services directly. Like our approach to self-contained services, we will base our approach on the construct of spheres of undo. Recall that a sphere of undo is a static construct whose boundaries are established by system state: a sphere of undo encloses the set of non-transient system state that can "time travel" as a single unit, and logically extends to include the parts of the system that operate on that state.

Since hierarchical services by definition are composed of nested hierarchies of state, there is a natural mapping between the state hierarchy of a hierarchical service and a collection of **nested spheres of undo**: each layer of the hierarchy represents an additional sphere of undo that encloses and contains the spheres of undo defined by lower layers of the state hierarchy. Figure 9-2 illustrates this nesting of spheres of undo for the e-mail service and ASP examples introduced above.

Because spheres of undo are defined by storage, a key property of nested spheres is that their storage is nested as well, so that rolling back the outer sphere will automatically roll back the inner sphere, but not vice versa. Figure 9-3 illustrates how the nesting of storage translates to this nesting of timelines. Note that peer spheres contained within the same enclosing sphere share no special relationship unless they interact with each other, in which case those interactions can be described by one of the

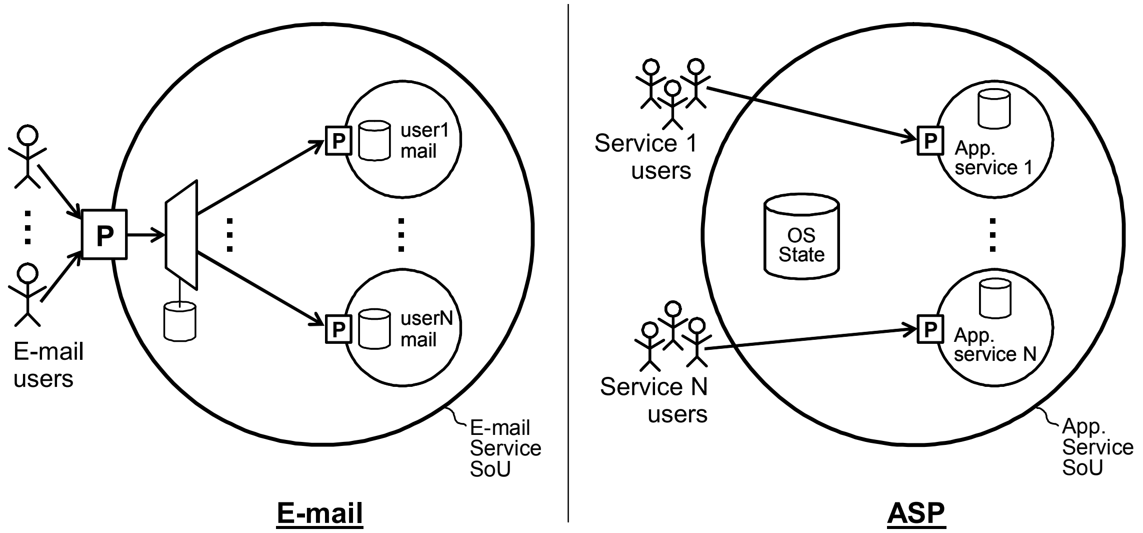


Figure 9-2: Examples of nested spheres of undo. The example on the left shows a nested spheres of undo composition for an e-mail service, with each user given the ability to roll back their own mailboxes via their own internal sphere of undo. In this case, the outer sphere receives all external requests and demultiplexes them to the appropriate inner spheres; the inner spheres are coupled by their common dependence on the outer sphere for initial request processing and demultiplexing. The example on the right shows a nested spheres of undo composition for an application service provider (ASP), with each application service offered by the ASP given its own inner sphere of undo. Here, the inner spheres of undo receive external requests directly, but are coupled by virtue of sharing the same execution and operating system platform.

composition models that we will discuss in the next chapter when we consider undo for distributed services.

Nesting spheres of undo gives us the ability to extend Three-R's undo functionality to a wider range of service architectures and to improve the efficiency of undo for hierarchical services that we would previously have shoehorned into a single sphere of undo. As we will see later in this chapter, our model of nested spheres of undo will also stretch to capture some systems not traditionally thought of in a service context. Developing undo models for hierarchical services will therefore lead us to a more sophisticated view of undo, ultimately providing an understanding of how to take Three-R's undo for operators from the service domain into the everyday desktop computing environment.

9.1 Mapping Service Architectures to Nested Spheres of Undo

Nested spheres provide a powerful composition construct that can capture several significant classes of service architecture ignored by the single-sphere model that we have discussed so far. They can also serve as a model for incrementally integrating a command-based redo of administrative actions into single-sphere services; as we will explore later in Section 9.3, this idea will be the first step in bringing a meaningful undo to the desktop environment. Before we get to these latter points, let us first consider the basic service architectures that suggest a nested-spheres model:

1. a service composed of finer-grained subservices that each handle a disjoint portion of the overarching service's request stream and hard state;

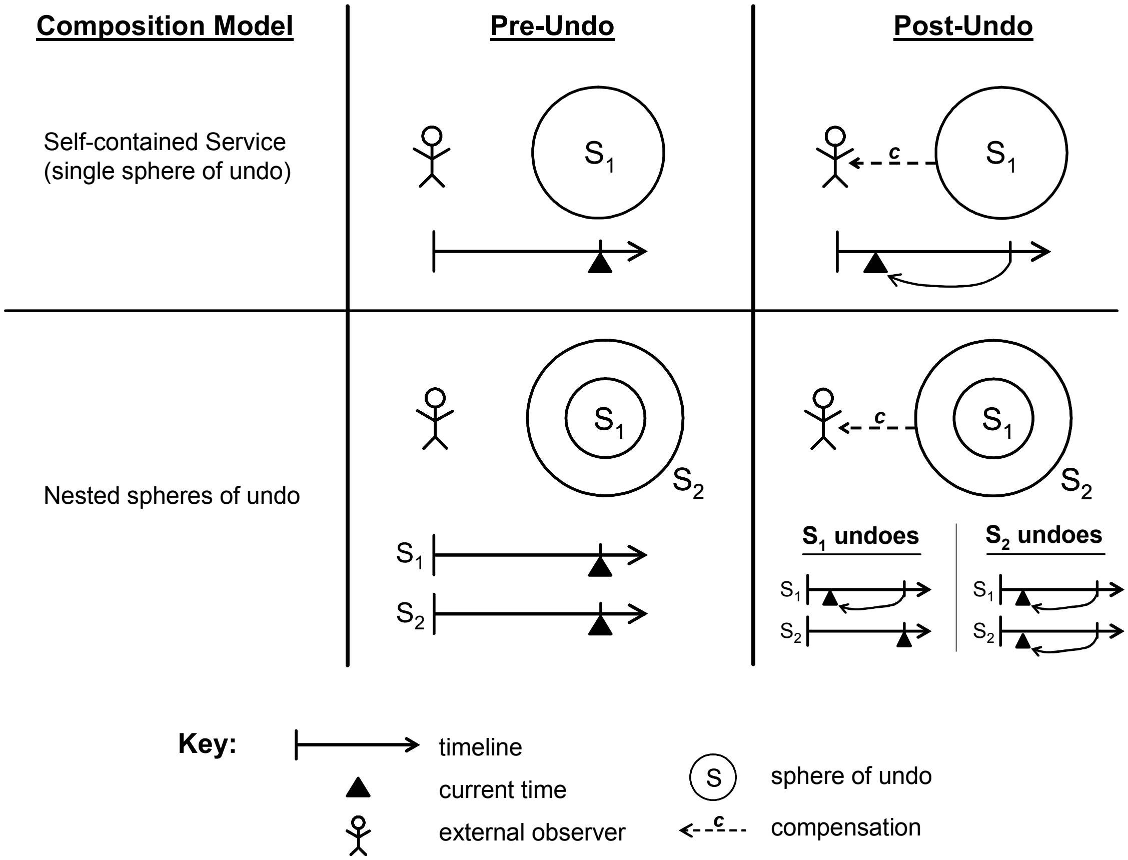


Figure 9-3: Timeline behavior of nested spheres. The figure illustrates how the timelines of nested spheres behave during undo, and compares that behavior to the behavior of a self-contained, single-SoU service. Unless otherwise noted, S_1 is the sphere on which undo is invoked. If the undo process can cause compensations or explanatory messages to flow outside of a sphere, these are indicated with dashed arrows tagged with the letter c.

2. a service composed of independent subservices that are coupled by sharing the same deployment platform (*e.g.*, operating system or application framework instance).

We saw these two architectures depicted schematically earlier in the examples of Figure 9-2.

The first architecture is typical of multi-user systems such as e-mail where users maintain independent state stores, and of clustered systems where different nodes handle different subsets of the overall service's requests, such as online services. The key properties of such systems are that the individual subservices can be rolled back using undo independently of the rest of the system, and that **coupling state** exists that controls how requests propagate to the subservices (and that may alter the processing of those requests). For example, in an e-mail store service like the one discussed in depth in Chapter 6, the individually-undoable subservices would be individual users' mail stores (allowing users to individually undo their own mistakes), and the coupling state consists of the filtering and routing configuration state that determines how incoming messages are filtered for viruses and spam, and how they are routed to appropriate user mailboxes. Similarly, in a clustered service, the coupling state might determine how requests are initially processed and how they are routed to cluster nodes for further processing.

When coupling state exists along with a desire for subservice-granularity undo, a nested-spheres undo approach is mandatory. Errors in coupling state can affect all subservices, so if the timeline for the coupling state is altered via undo, the subservice timelines need to be adjusted correspondingly to make sure they pick up any changes in the coupling state. The nested model makes this automatic, since by rolling back the outer sphere, both the coupling state and the subspheres are rolled back in unison.

The second architecture for nested spheres also concerns itself with coupling, but in a different way. In this architecture, the subservices truly are independent, but end up with coupling as a side effect of sharing the same execution platform. For example, consider an ASP or web-hosting service where multiple services (or instances thereof) run on the same physical machine. While the services themselves may be independent, and may directly process their own independent request streams, the fact that they share a hardware/software substrate requires that they be wrapped in an enclosing sphere of undo. To understand why, consider what happens when an operator wants to repair a problem or misconfiguration of that software substrate, such as patching an OS bug or reconfiguring an application service framework like J2EE. To fully correct the consequences of the problem, the operator must undo the substrate *and* the services running on it. In this case, the substrate is the coupling state, as the services running on it may be affected by alterations to its timeline. With a nested-spheres architecture, undoing the outer sphere and its coupling state automatically undoes the effects on the inner subservice spheres, achieving the desired behavior.

A service is a candidate for one of these two forms of nested spheres composition if all of the following criteria hold:

- the service as a whole can be viewed as a collection of finer-grained subservices for which independent undo is meaningful and desirable
- each subservice handles a part of the overall request stream and manipulates a disjoint subset of the overall service's state
- requests to different subservices are independent and commutative
- the subservices are coupled such that there is additional functionality and state that exists outside any particular subservice but affects all subservices, either directly (as in architecture #2) or by the way requests are routed to the subservices (as in architecture #1)
- hard state is maintained by a storage system that supports rollback for the individual subservice state stores as well as a global rollback of all state associated with the service.

9.2 Using Nested Spheres to Extend the Undo Model

Both of the nested-sphere architectures that we have considered so far can be seen as specialized cases of a general architecture, namely:

- a service with a hierarchy of state and different actors on each level, such that changes to the higher levels of state affect the lower levels but not vice versa.

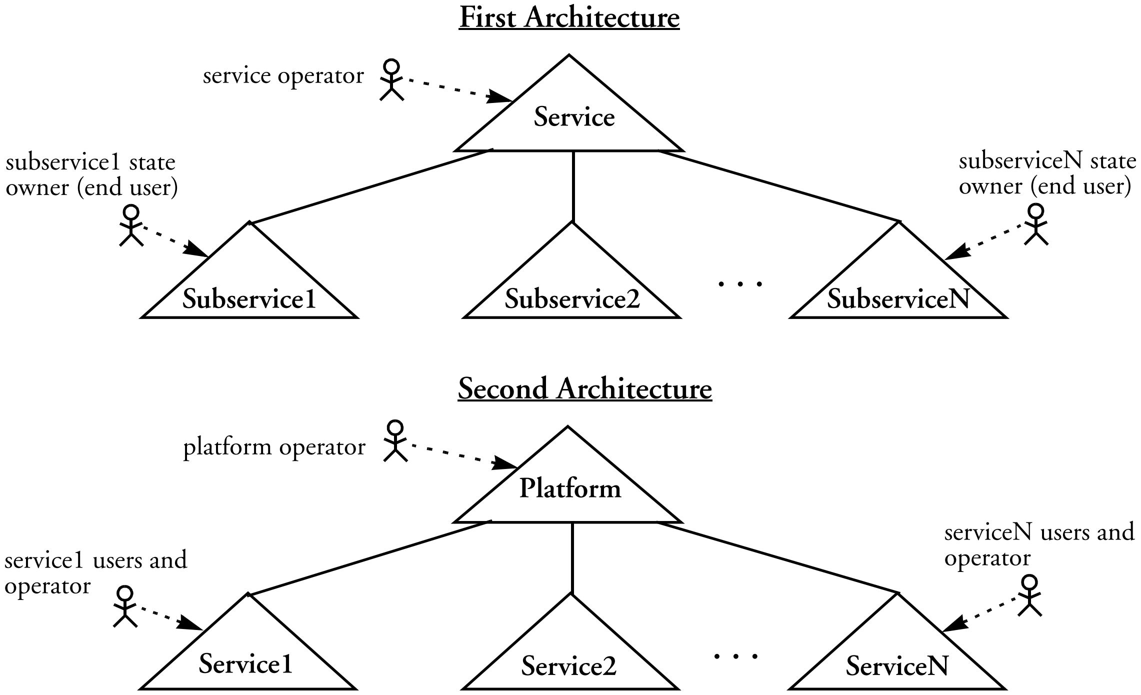


Figure 9-4: Nested sphere architectures as instances of general hierarchical design. The first architecture, typified by our e-mail service example, is represented as a two-level hierarchy with the service as a whole at the upper level and each subservice (representing a user's portion of the overall service) at the lower level. The second architecture, typified by our ASP example, is represented as a two-level hierarchy with the platform at the upper level and the services themselves at the lower level. The stick figures represent the actors at each level of the hierarchy.

Figure 9-4 illustrates the two architectures and how they map into the general hierarchical structure. In the first architecture, there is a two-level hierarchy. The lower level of the hierarchy is a subservice, and the actor at that level is the owner of that subservice's state, typically a particular end user, or in the cluster case the users and operator of a particular cluster node. The higher level of the hierarchy is the service itself, and here the actor is the operator of the entire service, the one who controls the coupling state. Similarly, the second architecture also has a two-level hierarchy, with the independent services and their users (as actors) at the lower level, and the overall collection of services and platform at the higher level, with the higher-level actor being the operator of the platform.

By extending this notion of hierarchy to include not only state but the undo model itself, we can use the notion of nested SoU's to model services that provide much more sophisticated undo models, including services that provide different forms of undo to match different classes of problems that occur. This new power comes from using nested spheres to nest undo models themselves, just as we saw in the simple case of Chapter 2, where we expressed the Three-R's undo model in terms of two nested spheres of undo. Figure 9-5 recalls this nesting.

The nested-sphere formulation of the Three-R's undo model provides a natural avenue for expanding our conception of undo models. For example, some services might include a well-defined administrative interface that provides the operator with some basic, well-understood commands that affect the service's configuration state. An example might be a control panel that provides constrained

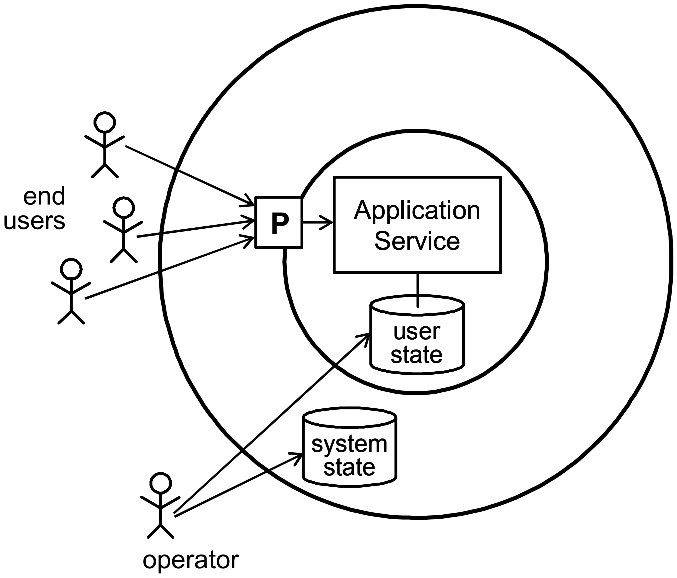


Figure 9-5: Recasting of Three R's undo as nested spheres. The inner sphere represents the application service and all end-user state; end users interact directly with this sphere. The outer sphere captures all non-end-user application state and system state; the operator interacts with this sphere. The inner sphere provides command-oriented undo/redo, and the outer sphere provides only a bulk committable-cancellable undo.

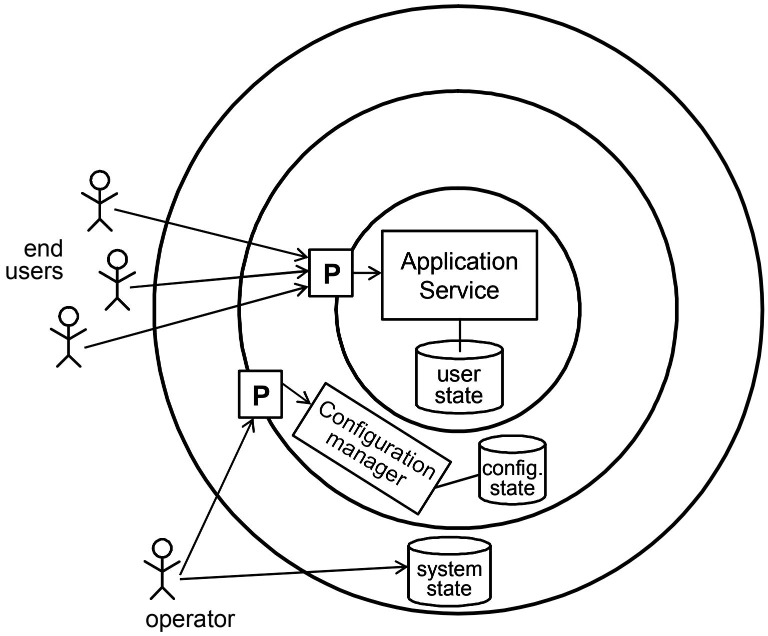


Figure 9-6: Three-level nesting to capture well-known operator configuration actions. The inner and outer spheres are as in Figure 9-5; the middle sphere holds configuration state and captures well-known changes to it as expressed by an operator's interaction with a configuration tool.

access to configuration variables, or that controls software patch installation. We might like to provide a command-oriented undo/redo model for these operations, while still relying on the bulk undo-only model for other operator actions that are not well understood by the system. Figure 9-6 shows how this is easily modeled by inserting another layer of nested sphere. Now, the operator is a first-class actor that generates commands that can be rewound and replayed at the middle sphere level. Should some operator make an error with one of these commands, or want to retroactively repair its effects, they can invoke Rewind on the middle sphere, rewinding the inner sphere as well due to the nesting, then Repair the history in the middle sphere and Replay the middle sphere, which again will cause the inner sphere to replay as well. Should operators want to use undo to repair a problem that goes beyond an administrative command (such as a software upgrade or a security breach at the OS level), they still have the bulk Rewind capability of the outer sphere, which will roll back all changes to the system; Replay is a no-op on the outer sphere, but will cause both the middle and inner spheres to replay.

Following this approach, each sphere can choose exactly what it will log, and how it will react to a Rewind or Replay request propagated from an enclosing sphere (or, for the outer sphere, directly from the operator). Each sphere similarly defines the history and consistency models that its associated external actors will see; in the example above, the innermost sphere preserves all work for its external actors, the end-users, while the outermost sphere preserves no work for its external actor, the operator who bypasses the service's administrative interface by performing arbitrary commands or software upgrades. Thus, the nested-sphere composition technique provides significant power in extending our undo model to handle whatever undo approach best matches a service and its collection of external actors.

9.3 Case Study: Desktop configuration management

Throughout this chapter, we have been hinting at the possibility of extending Three-R's undo to the desktop environment, where it could play an important role by allowing desktop users to recover from system administration problems like botched software installations, buggy drivers, and OS "aging" or corruption. In this section we tackle the problem head-on, treating the desktop as a service that provides application functionality to its end user, and using it as a case study of the flexibility and power of nested spheres of undo.

In a sense, each application—like Microsoft Word—that runs on a desktop provides some service to the user: the application services a request stream and manipulates hard state consisting of the application's data files, like documents, letters, reports, and so on. Many applications provide a limited form of undo already by retaining a history of verb-like representations of user actions for the duration of an editing session. It is not hard to imagine these applications being extended into full spheres of undo by making those transient histories permanent, perhaps by storing them along with a versioned representation of the documents themselves.

Applications also have configuration state affecting how they behave. Unfortunately, this state is rarely covered by the application's internal undo mechanisms, and hence it is something we would like to tackle with our undo system. To do this, we could wrap the application's SoU with a new sphere that handles the configuration state, creating a two-level nesting as shown in Figure 9-7. The user's request stream breaks down into configuration updates and document updates; the outer configuration sphere would intercept, convert to verbs, and log the configuration commands, while passing editing commands on to the inner sphere. This approach gives us a record of the configuration commands and provides the opportunity for embedding a command-based redo model in the outer sphere. Then, users who want to undo a configuration command can rewind the outer sphere and selectively replay all but the undesired command.

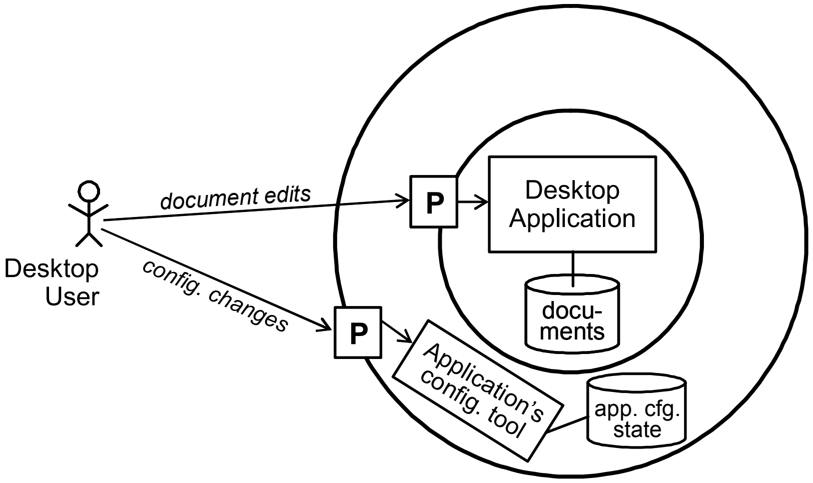


Figure 9-7: Two-level nested SoU model for desktop application. The inner sphere captures the user's changes to end-user state such as edits to documents. The outer sphere captures changes made by the user to the application's configuration, such as preferences or options that may affect the way documents are processed.

In our nesting model as defined in Section 9.1, replaying configuration commands would involve rewinding and replaying the inner spheres as well. While this is necessary to avoid inconsistency when the configuration commands affect the behavior of the editing commands, in the common case where configuration will not affect existing documents, we might want to allow the user to restore the inner spheres directly, without replaying, by copying their state forward in time much like the model used by the GoBack recovery package [102]. Determining when it is possible to skip the replay of inner spheres requires an understanding of the commutativity relationship between configuration verbs and verbs defined by the application's inner sphere; only if non-commuting configuration verbs are altered or omitted during undo of the outer sphere is replay of the inner sphere necessary.

Stepping up a level, the desktop as a whole runs multiple applications, or multiple services, atop a common OS platform. This is a natural description of a nested composition, where the OS platform defines an outer sphere of undo within which the application SoU's are nested. We choose a nested model because the OS platform sphere contains significant coupling state (in the terminology of Section 9.1), notably shared libraries, configuration repositories, and the binaries of the applications themselves. The OS platform also defines, in a sense, its own service model: it provides a configuration and software deployment service. For example, the Windows XP OS platform provides control panels that define the look-and-feel of applications, that control the installation and deletion of software packages (hence creating and destroying inner spheres of undo), and that control the configuration of the platform itself (via driver and feature installation and configuration). Unix-like systems offer similar functionality but without the well-packaged control panel interface.

Paralleling the approach we took with the configuration sphere for individual applications, it makes sense to treat requests to the OS configuration service as verbs at the platform level, and provide a rich command-oriented undo model in the OS platform sphere that allows the administrator-users to roll back and selectively replay these verbs. In doing so, we can even solve the software uninstall problem: the administrator can roll back the platform, undoing all software installations in bulk; then selectively redo the platform operations, reinstalling the desired software packages while omitting the undesired ones; then, as part of committing the undo cycle, the inner spheres will be replayed, restoring configuration changes and work done in the applications themselves.

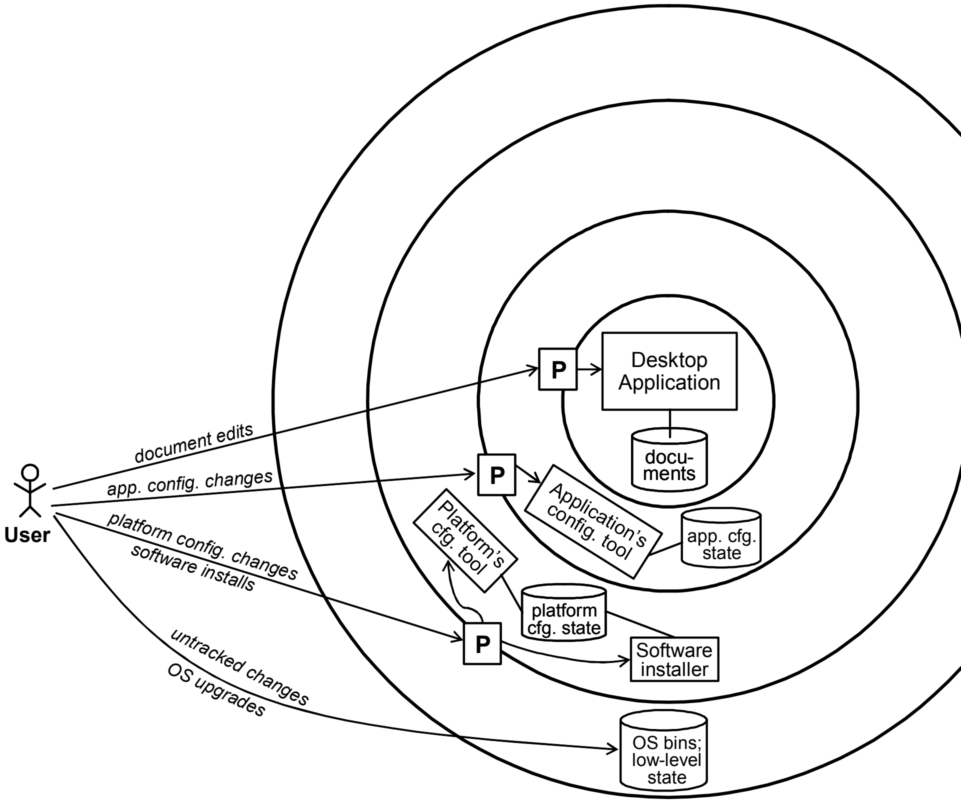


Figure 9-8: Nested SoU model for desktop environments. The inner two spheres are as in Figure 9-7. The second-from-outer sphere captures well-known configuration state and state changes for the platform in a command-oriented model, including for example OS preferences or application installs and patches. The outer sphere provides a back-stop bulk committable-cancellable undo for changes not captured by the inner spheres.

As with the application's configuration sphere, an optimization in the common case might be to not replay the inner spheres, but simply restore a state-based record of their pre-rewind state. It may be more difficult to apply this optimization than the corresponding one for the application's configuration sphere, since in this case, performing the needed commutativity test requires an understanding of the independence of operations performed across different vendor's software packages and the OS itself. If this information were readily available, there would be less need for undo and problems like software install and upgrade conflicts simply would not exist.

We can take further advantage of the ability to nest different undo models by wrapping the OS platform sphere in yet another sphere of undo, this one capturing changes to the system resulting from unforeseen corruption or unanticipated administrative actions that are not captured by the platform's standard configuration interfaces. These might include state corruption due to buggy application software or installers, damage due to viruses, direct erroneous manipulation of the file system by the administrator, or broken upgrades/patches to the OS itself. This sphere represents a back-stop recovery mechanism, and should provide an undo model similar to the one we used for self-contained services (with a bulk, state-based undo and no redo).

Figure 9-8 illustrates the complete composition that we have developed. Note that each of the nested spheres has its own external actors (or roles shared by the same human actor) and logs a verb history of its own set of requests. The outer sphere does no verb logging since its undo model is bulk-based, to recover from unanticipated problems. The middle spheres log software deployment and configuration commands through the platform and application control panel or configuration interfaces, and the platform's software installer interfaces; these spheres can provide a command-based undo model with bulk undo but selective redo of administrative commands. Finally, the inner spheres log application actions, allowing traditional productivity-application undo, extended to work across editing sessions.

We assume a storage system underlying the nested spheres that provides time travel for the hard state at each level. The outer two spheres work at similar granularities and can share the same hard-state rollback mechanism, for example NetApp-like snapshots, while the inner spheres need finer-grained rollback, for example with versioned files stored atop the snapshotting storage system.

Of course, making this model of desktop undo work requires significant extensions to existing applications and OS platforms. Both applications and the OS platform need to be extended to log configuration changes and software installs as verbs, and retrofitted with integrated undo managers that can log and replay those verbs. The good news here is that many of these operations are already designed to be scripted and/or automated, so packaging them into verb-based units that can be recorded and later automatically replayed may not require significant restructuring of the system. Also, significant user interface work is needed to present all the different undo possibilities to the user, without drowning them in the complexity of the nested timelines. Finally, the nested approach we have described requires significant support from the storage system; while all the pieces are available today (snapshots and versioned files), integrating them may prove a challenge. Here, the good news is that some commercial packages (like GoBack, as described in its associated file system patent [111]) already integrate snapshots, file versioning, and time-travel capabilities, so the proof of concept exists.

9.4 Implementing Nested Spheres of Undo

Nested spheres can be implemented with only minor architectural changes to the base Three-R's undo architecture of Chapter 4. The key to implementing nested spheres lies in the implementation of rewindable storage. The storage layer must support individual rewind of the storage of individual inner spheres. It must also provide a bulk rewind that rewinds the outer sphere's storage consistently with all of the inner spheres' storage. In other words, the hierarchical structure of nested spheres must be mirrored in a hierarchy of rewindable storage.

The other key implementation concern for nested spheres is the placement of proxy points—points where incoming requests are intercepted, turned into verbs, and stored in a history log. Proxies need to be placed at the boundary of every sphere that transforms or demultiplexes requests; for example, by filtering requests or assigning them to inner spheres based on their content. A request stream may be proxied at multiple sphere crossings; for example, in the e-mail system illustrated earlier in Figure 9-2, incoming mail deliveries are proxied at the outer boundary of the system as well as at the inner sphere(s) corresponding to the mailboxes to which they are ultimately delivered. Request streams may cross several sphere boundaries before being proxied, but, once proxied, they must not cross any further sphere boundaries unless those crossings are also proxy points; this requirement allows inner spheres to undo without involvement of their enclosing spheres. If requests produce external output, that output must be recorded in the verb logs at each proxy point along the request's path.

Given the appropriate proxy and logging points, and the needed support from the storage layer, nested sphere undo proceeds simply following the Three-R's model. For the outer sphere, undo begins by bulk-rewinding the storage of the outer sphere and all inner spheres. Replay then takes place at the outermost proxy point for each request stream; that is, the first proxy point a request encounters when entering the system. If a request passes through multiple proxies—for example, at both an outer and inner sphere—it is only replayed from the outer proxy, and the outer proxy always takes responsibility for detection and handling of external paradoxes using its own verb log.

This nesting is easily implemented by adding three new APIs to the undo manager of an inner sphere:

- `nestedRewind(time or lsn)`: resets the (inner) sphere's virtual time to the desired rewind point, sets aside the original verb log following that rewind point, and primes the sphere to record a new log as replay proceeds driven by the outer sphere;
- `commitNestedUndo()`: called when the undo cycle is complete, to install the new verb log recorded during replay and permanently discard the old log;
- `cancelNestedUndo()`: cancels the undo cycle by restoring the old verb log from the pre-undo execution.

With these APIs, the only extension needed to the log architecture of Chapter 4 is the ability to generate a second log and swap it in to replace the original. This functionality is easily implemented either by using multiple BerkeleyDB log databases or by adding a generation number to the log sequence number (LSN).

Should an inner sphere wish to undo unilaterally, it simply follows the same undo procedure defined in Chapter 4 for a standalone, self-contained service sphere. No special consideration is needed of the outer, enclosing sphere. Note that in cases where the request stream is proxied and logged by both the inner and outer spheres, changes made to the logged history as part of undo in the inner sphere will not propagate to the log of the outer sphere, and thus may be silently discarded later if undo is invoked on the outer sphere. If this is undesirable, then the inner sphere must implement a log-comparison procedure as part of the commit operation described above. By comparing the original and replay-generated input-verb logs at the inner sphere, the inner sphere can identify discrepancies and handle them by treating them as paradoxes with respect to the original input verbs, using those verbs' defined paradox-handling routines. This process has significant complexity, however, and therefore should only be used when absolutely necessary to maintain the service's semantics.

9.5 Related Work

Since spheres of undo were inspired by the spheres of control developed in the database community, we might expect to find analogies between nested spheres of undo and nested spheres of control. However, the spheres of control model surprisingly does not have an equivalent notion of nested spheres. There is somewhat of a parallel with non-distributed nested transactions if we treat each arriving verb as defining a new transaction [47], but nested transactions fail to capture the interesting behavior of nested SoU's, particularly with regard to coupling state, the nesting of undo models, the fact that outer spheres can be containers that pass requests through to their inner spheres without processing them, and the implications for multi-level rewindability of a static hierarchy of state.

Looking beyond traditional database models as captured in spheres of control, the only directly related work in the literature for nested spheres of undo is Edwards *et al.*'s work on temporal models in the Flatland system [33]. Flatland is a digital whiteboard application that supports "segments", or zones, of activity, each with its own local history. Changes to each segment can be undone (rolled back) and redone (rolled forward) independently of the global history of the collection of segments making up the whiteboard; but the entire whiteboard can also be rolled backward and forward as a whole, "snapping" all segments to the global time and rolling them backward or forward in concert. To support these operations, each segment maintains its own history, and the whiteboard maintains a global history synthesized from the local segment histories.

Flatland's multi-level history model is very similar to our nested spheres of undo; each Flatland segment can be seen as its own sphere of undo, all nested inside an enclosing sphere that represents the global history of the whiteboard. Like our nested spheres of undo model, Flatland's inner spheres (segments) can time travel independently of the enclosing (global) sphere, but time travel in the enclosing sphere takes precedence over the timelines of the inner spheres.

Despite these structural similarities, however, our nested spheres composition has some unique properties that distinguish it from the Flatland approach. First is the notion of coupling state, which is not present in Flatland's mode. Flatland assumes that global time travel can be accomplished by composing local time travel operations on the inner spheres; in our model, changes to coupling state can alter the request stream that reaches the inner spheres, in some cases requiring that the histories of the inner spheres be regenerated, rather than replayed, during undo/redo of the enclosing sphere.

The Flatland model is also heavily concerned with how to synthesize the global timeline from per-segment timelines that interact (as when operations in one segment provoke dependent operations in another sphere). Our model makes most of these concerns moot, as our global, outer-nest timeline logically consists only of *external* operations (input verbs); inter-sphere verbs between inner-nest, coordinated spheres are not explicitly replayed, but rather are regenerated during Replay as a result of re-executed external verbs. Since external input verbs are, by definition, not provoked by other spheres of undo, they are independent and can easily be sequenced into a global timeline.

9.6 Summary

In this chapter, we tackled the problem of providing efficient Three-R's-style undo for hierarchical services, services that could be viewed as single monolithic self-contained services, but that naturally decompose into smaller subservices for which independent undo is meaningful and worthwhile. Hierarchical services cover three common architectural patterns:

- collections of independent subservices coupled by virtue of a shared execution platform
- multi-user services where users' ostensibly-independent service state is coupled by filters, routers, or transformation code
- typical desktop systems combining multiple applications, each with configuration state, on a shared execution platform with its own configuration state.

We approached hierarchical services by mapping the hierarchy to a nesting of spheres of undo. Subservices define their own spheres of undo, and are nested inside enclosing spheres that also include the coupling state shared amongst subservices. Inner spheres of undo can carry out the Three-R's undo process independently, but undoing an outer sphere propagates the rewind and replay operations to any inner spheres it contains. Thus, retroactive repairs of an outer sphere or the coupling state it contains are automatically propagated to the inner spheres affected by those repairs. The implementation complexity required to realize nested spheres is tractable, requiring only small changes to the undo architecture of Chapter 4, consisting mostly of three new APIs for the undo manager and minor modifications to the timeline log management code.

Nested spheres of undo enable finer-grained, more efficient undo: it is only necessary to rewind/replay the subservice needing repair and any of its subservices; other services at the same or higher levels of hierarchy can be left unchanged. They furthermore provide a way to extend Three-R's undo to non-service contexts like the desktop environment, by providing a framework for nesting undo models as well as state: each sphere of undo in a nesting can define its own approach to rewind and replay—be it command-based, state-based, selective, or complete—and thus tailor its recovery behavior to the type of information and operations that it handles.

Chapter 10

Evaluating Recovery-Oriented Tools: Human-Aware Recovery Benchmarks

> "For better or worse, benchmarks shape a field."
>
> — *David Patterson*

At the start of this dissertation, we posited that services with system-level undo could offer improved dependability compared to equivalent services without undo. Now that we have established a model, framework, and specific implementation of system-level undo, we are finally in a position to test that hypothesis. To carry out such a test, however, we will need to find an evaluation methodology powerful enough to quantify the dependability gains or losses that result when Three-R's undo is added to a service.

Surprisingly, the methodology that we need does not exist. While there has been significant work in recent years on dependability benchmarking [14] [16] [57] [74] [130] [138], none of it has explored the problem of quantifying dependability when a human operator is involved in the recovery process. Thus, before we can evaluate the dependability consequences of a recovery mechanism like undo, we need to develop a **human-aware** methodology for benchmarking dependability, or, more precisely, a **human-aware recovery benchmark**. Developing this methodology will be the focus of the first half of this chapter, and we will eventually arrive at an approach that merges traditional dependability benchmarks with user studies involving human participation. With that methodology in place, we will return in the second half of the chapter to the question of the dependability consequences of undo. Through a set of benchmarking experiments using our undoable e-mail store service prototype and involving 13 human participants, we will demonstrate that system-wide undo facility does indeed improve dependability, confirming our original hypothesis.

10.1 An Approach for Human-Aware Recovery Benchmarks

Recovery benchmarks are a special case of general **dependability benchmarks**, which measure a system's overall dependability as expressed in terms of metrics like availability, performance, reliability, correctness, performability, and quality of service. A traditional dependability benchmark works by measuring the impact of injected software and hardware faults (a **faultload**) on the performance and correctness of a test system being subjected to a realistic workload [16] [74]. For example, a traditional dependability benchmark might inject a hardware fault, then measure the deviation in the test system's performance relative to its fault-free performance. The length and magnitude of any such performance deviations are measures of the performance component of the test system's dependability in response to that fault. Recovery benchmarks are similar except that they focus only on the test system's behavior following the point where the system behaves erroneously or fails; they do not concern themselves with injected faults that remain inactive and cause no visible error or failure symptoms.

To date, all published dependability benchmark methodologies are designed to be run without human operator intervention, in order to eliminate the possible variability that arises when human behavior is involved. But this approach ignores the significant human contribution to dependability, and certainly is inappropriate for benchmarking a recovery process in which a human operator is a major player, like with undo. Thus, we must adapt traditional dependability benchmarking techniques to benchmark the combination of both the test system and the human operator.

To accomplish this joint measurement of system and operator, we extend the traditional dependability benchmarking methodology by allowing the human operator(s) to interact with the system during the benchmark. The key insight of our approach is that we can *indirectly* capture the human contribution to dependability: we simply treat the operator's interactions with the system as a source of perturbation, much like the perturbation introduced by an injected fault. With this approach, any effects that the human operator has on the system's dependability are reflected in the traditional metrics already being collected: performance, correctness, availability, and so on. There is no need to measure or incorporate fuzzy user-centric metrics like human error rate, user satisfaction, think time, or comprehension level. The human operator's involvement with the system simply becomes part of the faultload.

For recovery benchmarks, the human operator's interactions with the system are mostly **reactive**: they are instigated in response to a need for recovery caused by some other source of failure. General human-aware dependability benchmarks will also typically include **proactive** interactions, where the human operator is asked to perform preventative maintenance tasks as well as to recover from problems. Although scripted proactive interaction is not appropriate for recovery benchmarks, since recovery is inherently reactive, we do want to capture the dependability consequences of human errors made during proactive tasks, as these are still a significant source of human error and consequently system failure [85]. To accomplish this, we incorporate proactive tasks into the injected faultload by simulating the errors that result when proactive tasks are performed incorrectly, much as in Vieira and Madeira's work on DBMS dependability benchmarking [129]. Then, our recovery benchmark's indirect dependability measurements capture the effects both of the erroneous task as well as of the resulting recovery process carried out by the system and its human operator.

While the above discussion of our approach implies that human operators must participate in the benchmark process, one might wonder if we could simulate the human perturbations and thus eliminate the human. Unfortunately, this reduces to an unsolved problem—if we were able to accurately simulate human operator behavior, we would not need human system operators in the first place! While the HCI community has developed techniques for modeling human behavior in usability tests [56], even in those approaches human involvement is required at least initially to build the model, and the resultant models are typically highly constrained and system-specific, making them inappropriate for use in a comparison benchmark.

Thus, we are left with the approach of using live human operators in the benchmarks; this is the only way to truly capture the full unpredictable complexities of the human operator's behavior and the resulting impact on a system's dependability. A major concern with involving live humans, even indirectly, is that they will introduce so much variability as to make the benchmark results useless. While this may be true for a naïve benchmark design, we will demonstrate in the following sections that it is possible to reduce variability through careful management of the human benchmark participants, and to further control it by designing the benchmark experiments to treat variability as a useful indicator of the applicability of recovery, rather than as a confounding factor.

10.2 Practical Methodology for Human-Aware Recovery Benchmarks

The discussion in the previous section provides a conceptual approach to building human-aware recovery benchmarks like those that we will need to evaluate undo. Next, we turn that approach into a realizable benchmarking methodology, addressing the practical challenges that arise when we do so. We begin with a summary of the methodology.

10.2.1 Methodology overview

Our methodology for human-aware recovery benchmarks is an extension of the traditional dependability benchmarking approach [16] [74]. It consists of four stages:

1. **develop faultload:** first, a faultload is developed that mimics faults and errors likely to be seen in practical deployment of the system under test. In addition to identifying the categories of faults and errors to use, this stage of the methodology also includes the process of developing fault- and error-injection technology that can be used to apply the faultload to the system under test during the benchmark.

2. **develop workload and metric collector:** during the benchmark process, the system-under-test is subjected to an artificial user workload both to simulate actual load on the system and to provide a means for collecting data concerning the test system's performance and availability. The workload should be representative of deployed workloads, and ideally should be based on an industry-standard performance-oriented benchmark workload for reproducibility. The workload generator should collect data on the performance, correctness, and availability behavior of the test system as it services the generated workload.

3. **recruit, select, and train human participants:** because our methodology uses human participants in its benchmark experiments, a key step in the process is to recruit potential participants. After recruiting is complete, the participant pool needs to be screened to select a set of actual participants with similar background and skills. Finally, the

selected participants need to be trained to further reduce the variance in their knowledge and experience.

4. **perform benchmark experiments:** the overall benchmark study consists of multiple experiments, each involving one of the participants selected in the previous step. Each experiment takes the form of a human trial, with the human participant playing an active role in maintaining the test system as it is subjected to the workload and faultload developed in the first two steps. The results of each experiment consist of the time-varying dependability metrics as synthesized from the raw data collected by the workload generator.

Note that the third step in our methodology—recruiting human participants—is unique to our benchmarking approach, and is not found in traditional dependability benchmarking methodologies. Likewise, both the inclusion of human participants in the fourth step and the format of the benchmark experiments themselves represent a deviation from traditional dependability benchmarking methodology.

The methodology as presented above raises several difficult practical challenges: selecting an appropriate faultload for a human-oriented recovery benchmark, managing the process of obtaining and preparing human participants for the benchmark, and finally developing detailed experiment designs that can provide useful results despite the variability inherent in human participant behavior. We discuss each of these challenges in turn in the following sections.

10.2.2 Selecting a faultload

The first practical challenge in deploying a recovery benchmark is to develop the faultload. We initially thought that we could develop the faultload by combining traditional hardware/software fault injection with proactive maintenance tasks derived from published studies of how operators spend their time [3] [62] [63]. We found this approach unworkable in practice: time studies do not provide enough information to identify or simulate the failure modes of common maintenance tasks, and hardware/software fault injection is unlikely to produce results mimicking the maintenance-related failures that dominate server dependability.

Our solution is to turn to actual deployed systems and their real-world system administrators for inspiration. Ideally, we would perform a **task analysis**, in which we would shadow real operators as they carried out their day-to-day duties and distill our observations into a faultload [67]. However, task analyses are costly, require difficult-to-gain access to the shadowee's workplace, and their results are often embargoed by confidentiality agreements. A compromise that provides similar data with far lower cost and fewer restrictions is to instead perform a **task survey**, in which practicing operators and system administrators are surveyed in order to identify dependability-critical real-world recovery and maintenance tasks.

Surveys can be performed much more quickly and at much lower cost than task analyses, since they can be broadcast to a large audience via online web forms and furthermore do not require bringing outside observers into an operator's workplace. As such, surveys can produce much higher response rates than traditional observation-based task analyses, and therefore can expose more of the space of possible tasks. The only downside of surveys is that they are self-reported and thus may not provide as high-quality data as an observation-based task analysis. For the purposes of generating a faultload, however, perfection is unnecessary; *any* real-world fault data will likely expose useful dependability behavior.

After some pilot trials, we settled on the following survey questions as a good starting point:

1. Describe three of the most common tasks you perform that require interaction with the server.
2. Describe any incidents you can recall within the past 3 months in which data was lost or the service became unavailable or degraded (if applicable). Please indicate whether those incidents were a result of administrative intervention or due to failures of hardware, software, or network components.
3. Describe any service administration tasks you performed in the last 3 months that were particularly challenging (if applicable). If the execution of any of these tasks resulted in lost data or service unavailability, please explain.

These questions form the core of the survey, but should be surrounded with simpler multiple-choice questions to gather information about the respondent's experience and environment, as a means of normalizing answers and understanding unusual responses. It is crucial that the survey offer total anonymity, as otherwise administrators may not be willing to reveal the details about major failures and outages.

The second and third questions above are the key to developing the faultload. Question 2 gathers data on real-world dependability incidents and their causes; responses to this question can become faultload scenarios directly. But since real-world dependability incidents are often infrequent, there may not be enough data from question 2 to generate the faultload. In this case, question 3 comes into play, by identifying challenging and error-prone tasks. These tasks, along with the most common tasks from question 1, can then be reproduced on a test system in the experimenter's lab to identify possible failure cases that can be exploited for the faultload. Examples of this process are given for our case study in Section 10.3.2.

10.2.3 Selecting, recruiting, and training participants

With the faultload in place, the next challenge in a human-aware recovery benchmark is to select, recruit, and train the human participants that will act as operators during the benchmark experiments. These are crucial steps since they are the primary means of controlling the variance inherent in any experiment involving people.

The first step in participant selection is to decide on the target population; for recovery benchmarks, the choice will usually be between recruiting experienced system operators (experts) or using technically savvy but non-expert users. For recovery benchmarks designed to evaluate and compare existing, deployed systems, expert participants are the appropriate choice: they can be drawn from the existing user base of the deployed system; they will showcase the best-case behavior of each system; and they will minimize variance.

But when recovery benchmarks are being used to evaluate new recovery tools or approaches that have *not* been deployed in practice, non-expert participants are more appropriate. There are two reasons for this somewhat non-intuitive result. First, systems that have not been widely deployed will not have a wide base of expert operators from which to recruit; the only experts available may be the system designers. More importantly, if experts are recruited that are familiar with other systems, or worse with an earlier version of the system under test, they will bring a strong experiential bias to the experiment. We saw this effect in early pilot studies for our evaluation of undo, where participants that were experts with traditional e-mail servers balked at using undo, even when instructed that the new but unfamiliar tools would offer more effective recovery than the traditional, familiar ones. The alternative approach of starting with technically competent but non-expert participants and training them to a common knowledge base avoids the problem of mental inertia and gives new tools and approaches the opportunity to compete on a level playing field.

With a class of participants selected, the next issue is how to recruit actual participants. For academic researchers like us seeking non-expert participants, recruiting is relatively easy— we can simply draw on the student population of our departments once granted the needed approvals from human subject protection committees. We found that a broadcast e-mail request accompanied by a promise of a reward to selected participants was sufficient to attract a substantial subject pool—between 15 and 20 subjects. Recruiting experts in an academic environment will likely be more difficult, requiring a more substantial incentive and a broader recruiting effort. Those in industrial environments may be able to leverage internal staff to obtain both experts and non-experts, although we have no first-hand experience with recruiting in such environments.

Regardless of how recruiting is accomplished, we found it extremely valuable to screen volunteers according to their background before selecting them as participants. We chose to conduct the screening by having prospective participants fill out a web form as part of the recruiting process. We devised a screening form containing the following components:

- an overview of the experiment and what would be expected of the participants
- a self-evaluation section where prospective participants reported their background in various systems operations tasks on several different platforms
- a simple test of relevant knowledge or skills; in our case study, for example, we verified that participants could properly identify a collection of e-mail protocols and server components.

In our experience, we found that we could obtain participants with reasonably equivalent skills and background by choosing all respondents with an aggregate self-evaluation score above a fixed cutoff and no errors on the knowledge/skill test.

Finally, with participants recruited and screened, the final issue is how to train them. Training plays a significant role controlling variance, particularly when using non-expert participants. To provide participants with the background they need to complete the benchmarks, training should include some general background on the type of system used in the benchmark experiments, a presentation of the experimental setup (machine names, software versions, network names, log locations, and so on), and a tutorial on the recovery tools and resources available during the benchmark.

The most important concern with training is that it be consistent across participants. We found that using printed training materials is the easiest way to get consistency. Since the training has to span the experience range of the participant pool, the materials should include as much detail as is needed to get the least-experienced participant up to speed. Accordingly, the training process should be self-paced to allow more-knowledgeable participants to skip over familiar details. We also found it helpful to encourage participants to ask any questions they wanted during the training period, allowing us to fill in unanticipated gaps in their knowledge.

If the benchmark involves a system with which the participant is unfamiliar (such as a new recovery tool), the training process should include a walkthrough of a simple task using that system to familiarize the participant with its interface and operation. While some might argue that this biases the participant toward using the new tool, we found in our pilot studies that it was essential to build at least basic familiarity with new tools for the benchmark to be useful. When we tried benchmarks without this pre-familiarization, participants were less likely to use the new tools or even to remember that they were available. In real-world deployed situations, operators and administrators will have at least basic awareness and knowledge of the recovery tools available to them, and the goal of the pre-familiarization during the benchmark training period is to bring the participants to at least that level of awareness.

10.2.4 Experiment design

A recovery benchmark can be used for one of three purposes: standalone evaluation of a single system, comparative evaluation of a new or improved system relative to a baseline system, or comparative evaluation of two or more systems. In this section, we consider the experimental design for each of these cases. In all cases we assume a standard dependability benchmark framework like that described in [16], with a workload generator that applies a representative workload to the test system and collects dependability metrics arising from how that workload is serviced.

The most basic approach to recovery benchmark experiment design is to use the format of a randomized trial. In this format, each participant is randomly assigned one of a fixed set of scenarios, each corresponding to one of the failure scenarios that comprise the faultload. For standalone single-system evaluation, each participant uses the same system; for comparative evaluations, participants are randomly assigned to one of the available systems. During each experiment, error injection is used to simulate the appropriate failure scenario, and the participant is then asked to recover the system to normal operation. The workload generator measures the system's dependability throughout the recovery interval.

While simple, this basic format suffers from the curse of variability. Since the only comparisons that can be made are between different participants, it will only work for very homogeneous or large participant pools—where the inherent variability between participants can be averaged out—or when the dependability effect being measured is so large as to swamp the human variability. During our own pilot studies, we found such extreme variation in approach and behavior across participants that we do not believe this simple approach is practical for dependability benchmarking of server systems. Part of the problem is that the typical server system offers so many different tools and possible avenues of recovery (especially command-line-oriented systems like UNIX) that large variability is unavoidable. Indeed, even trained expert system administrators will often disagree on their approaches to a particular problem, and often more than one solution is possible and effective.

The obvious way to deal with the variability problem is to avoid comparisons between participants. In comparative evaluations (either of different systems or of an improved system against a baseline), this can be accomplished by comparing each participant only to him- or herself. This approach allows the experimenter to draw independent conclusions for each participant as to whether a particular system is more dependable than another, in essence shifting the random variable from a continuous measure of the system's absolute dependability to a binary measure of whether one system is more dependable than the other. Variability still remains across participants, but this variability is useful to gauge how dependent the dependability advantage (or disadvantage) is on the human operator's background and skill.

To implement the self-comparison approach, participants are repeatedly given the same failure scenario in multiple consecutive sessions, with a different randomly-selected test system provided each time. For comparative evaluations that pit a new or improved system against a baseline, a simpler experiment design is possible: each participant carries out two sessions using the same failure scenario, with the improved system used in the first session and the baseline system in the second. This approach is simpler and needs fewer participants than the randomized approaches described above, but it introduces a systematic bias against the improved system, since the subject may gain insights in the first session that can be applied in the second session. However, because the bias is against the improved system, a positive conclusion can still be drawn from the benchmark if the improved system demonstrates better dependability than the baseline; only if the baseline outperforms the improved system is a more complex randomized trial needed. The trade-off toward simplicity at the cost of known bias may well be worth it for benchmarks intended simply to verify whether a new recovery tool or technique improves dependability, like those we intend to use to evaluate undo.

10.3 Tailoring the Benchmark Methodology for Undo: Faultload

Now that we have established a practical methodology for human-aware recovery benchmarking and analyzed the number of participants needed, we can use it to benchmark the dependability of Three-R's undo. This section describes how we set up a recovery benchmark to evaluate undo; the next section presents the results of that evaluation. To evaluate undo, we used the Java-based implementation of Three-R's undo for an e-mail store service, as described in Chapters 5 and 6. Recall that our undoable e-mail store prototype is a first proof-of-concept; the results of our evaluation therefore provide only a lower bound on the dependability benefits of undo.

Our goal in benchmarking undo was to understand the dependability consequences of having undo-based recovery, and to test the hypothesis that an e-mail server with undo provides higher dependability than a similar server without undo. We thus chose an experiment design optimized for comparative evaluation versus a baseline, allowing us to use the optimization discussed in Section 10.2.4 of reducing our subject pool at the cost of a systematic anti-undo bias; our baseline was the same e-mail server with the undo tool disabled and its components bypassed.

We followed the methodology of Section 10.2 in performing the benchmarks, beginning with a survey of practicing e-mail administrators to establish a faultload, and culminating in a user study in which we recruited human participants from the student population of the U.C. Berkeley computer science department and carried out experiments following the procedures of Section 10.2. We describe the survey and faultload in this section, then describe the study design and its results in the following sections. All of the materials that we used for our evaluation of undo (survey, questionnaires, and training materials) are reproduced in Appendix A.

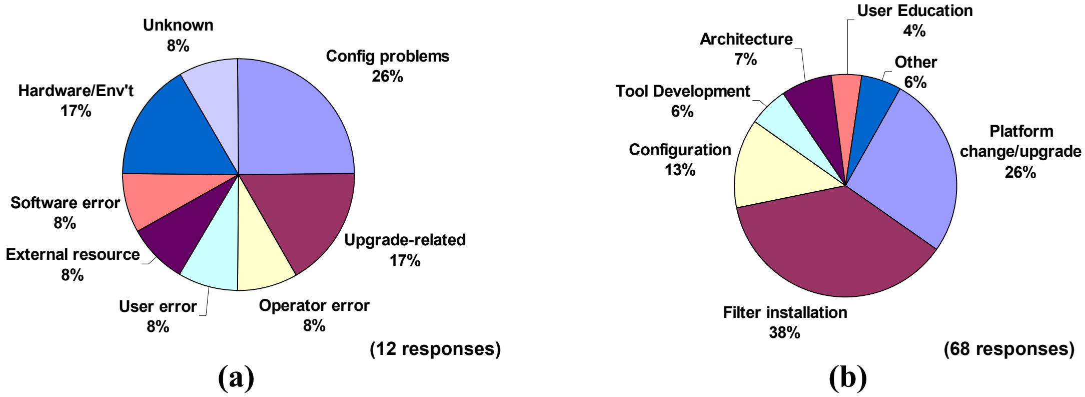


Figure 10-1: Survey of e-mail administrators. The graphs show the breakdown of responses given by practicing e-mail system administrators when asked to describe (a) the causes for actual situations where e-mail was lost, and (b) challenging e-mail management tasks they had performed recently. The data suggest that a benchmark faultload for e-mail should include configuration errors involving SPAM filters, upgrades, simple restart-repairs, and hardware failures.

10.3.1 Survey of e-mail administrators

We began our case study with a survey of practicing e-mail administrators, to establish a faultload as described in Section 10.2.2. We developed a web-based survey and e-mailed a request for participation to the mailing list of SAGE, a membership organization dedicated to the profession of system administration [106]. We received 68 responses to the survey, and collected descriptions of 151 common e-mail administration tasks, 68 challenging e-mail management tasks, and 12 descriptions of scenarios where e-mail data was lost. We began our analysis by categorizing the responses, starting with the data-loss scenarios. Figure 10-1a shows our breakdown of these scenarios. The most common failure scenarios involved configuration errors (typically involving SPAM/virus filtering software), failed upgrades of the e-mail server platform, and hardware or environmental failures.

To get a broader perspective on the human component of the faultload, we turned to the set of challenging management tasks. As Figure 10-1b shows, the dominant management challenges involved upgrades, SPAM/virus filter installation, and configuration management. An analysis of the 151 reported common tasks provided similar results, with configuration management dominating at 56% of the reported tasks and upgrades also significant at 12%. The other significant category of common task was simple repairs like restarting crashed server processes, which accounted for 15% of the reported tasks.

10.3.2 From survey to faultload

Taken together, the survey data suggests that our faultload include scenarios mimicking configuration errors involving SPAM filters, upgrades, simple restart-repairs, and hardware failures. Because of time limitations, the limited number of participants available to us, and the desire to have each scenario tested by multiple participants, we were forced to restrict our faultload to three distinct failure scenarios. We chose one each of the above categories, omitting the hardware failures; hardware failures are not repairable via undo, and thus contribute little to a comparative benchmarking approach.

Unfortunately, the survey results were not detailed enough to use as error-injection recipes, so we had to perform additional analysis to create a practical faultload. To do this, we configured an e-mail server in our lab and manually performed a SPAM filter installation and configuration and an upgrade of the mail server software. Using a process similar to a cognitive walkthrough [56], we identified specific errors that would produce the symptoms reported in the survey results. We then devised scripts that would automatically generate those errors, thus creating the desired faultload. In the end, our faultload consisted of the following three scenarios:

1. **SPAM filter configuration error:** this scenario mimics a human error made by the operator while configuring a SPAM filter based on the MIMEDefang and SpamAssassin mail filtering tools. The fault is a mistyped configuration line in the MIMEDefang filter script. The resulting syntax error causes all incoming e-mail that is less than 200KB in size to be silently rejected.

2. **Failed e-mail software upgrade:** this scenario mimics the results of a bad upgrade to the e-mail server platform. The fault occurs when, in the process of upgrading the Sendmail mail server software from version 8.12.9 to version 8.12.10, the operator forgets to activate the compile-time option needed to enable mail filtering. The symptoms of the resulting failure are that once the upgrade is installed, all mail filtering ceases.

3. **Simple software crash:** this scenario simulates a crash in the e-mail server software, either due to a hardware fault, software bug, or external attack. We do not inject the fault directly, but rather simulate the resulting error by terminating the Sendmail server process. The symptoms of the resulting failure are that no incoming e-mail is accepted by the mail server.

Note that the first two scenarios are ones for which undo-based recovery is potentially useful; in the third scenario, undo is unnecessary since no system state has been damaged. Including this scenario is important, however, as it will show whether our participants can tell when it is appropriate to use the undo-based recovery tool, and whether simply having the tool available (and unused) has any dependability impact.

10.4 Tailoring the Benchmark Methodology for Undo: Setup

With the failure scenarios in place, we set up and performed a dependability benchmark following the design and procedures of Section 10.2.4. We recruited non-expert, technically-savvy participants from the student population of the U.C. Berkeley computer science department by sending out a mass e-mail solicitation accompanied by an online screening questionnaire. The questionnaire asked respondents to self-report their own system management experience with Linux, Windows, and other systems, and also included a skill test in which we asked respondents to match protocol and service names to their uses. We received 18 responses to our solicitation, 14 of which met our screening criteria of having at least 60% of the maximum possible self-reported experience and no more than one error on the skill test. Of these 14 respondents, 13 agreed to participate in the benchmark experiments, and 12 successfully completed the experiments. Participants were offered a $50 gift certificate to online retailer Amazon.com to compensate them for their time.

10.4.1 Protocol

We constructed each benchmark experiment in three phases. In the first, the participant completed informed consent paperwork, then spent as much time as they wanted reviewing a set of training materials that introduced the setup of the e-mail server system and described the undo-based recovery tool. Participants were not told at this point which failure scenario(s) they would be given. Also during the training period, the participants were encouraged to follow a simple task walkthrough that familiarized them with the e-mail server setup: they were asked to verify that the Sendmail e-mail server was running, to edit one of its configuration files, and to restart it. They were further given the opportunity to experiment with the undo-based recovery tool and its user interface.

The remaining two experiment phases constituted the actual dependability benchmark, constructed as two trials. In each trial, the participant was given a symptom report describing one of the scenarios in the faultload, and was asked to recover the e-mail system to normal operation. The scenarios were randomly assigned to participants, but each received the same scenario in both trials, following the self-comparison style of experimental design discussed in Section 10.2.4. Each trial had a 30-minute time limit, although participants could choose to end the session early if they felt they had completed the recovery. Participants had access to the Internet, a book describing the Sendmail mail server, a backup consisting of a snapshot of the system from the prior day, an e-mail client configured to send test messages, and all standard tools available on the e-mail server system. Undo functionality was available *only* during the first trial. While participants could ask any questions they wanted during training, we refused to answer questions during the benchmark trials, with one exception: each participant was allowed *one* question during each trial, just as in real-life an administrator might appeal to a guru for help. The goal of this “guru” resource was to prevent frustration should the participant get stuck; it was only used three times across the 28 benchmark trials we conducted.

10.4.2 Measurement workload

During each of the benchmark sessions, we applied a workload of simulated e-mail traffic to the server under test, including both a stream of incoming e-mail via the SMTP protocol and a stream simulating the actions of users checking mail via the IMAP protocol. Incoming e-mail was generated according to a Poisson process with a rate of 5 messages per minute and randomly-chosen message sizes based on the distribution used by the standard SPECmail2001 e-mail benchmark [118]. Each piece of incoming e-mail was hashed and stamped with a unique ID; at the end of each session the workload generator attempted to retrieve each message to verify whether it had been received, processed correctly, and filtered if appropriate.

We chose the incoming mail rate to balance the desire for frequent system measurement with the time cost to the participant of potentially having to rewind and replay large amounts of e-mail traffic. Given that the rewind and replay implementations in our e-mail undo prototype are suboptimal, that our test systems were running on resource-limited virtual machines as discussed below, and that each trial was limited to 30 minutes, we ended up choosing a lower rate than that used in the performance benchmarks discussed in Chapter 6; our 5 messages per minute are roughly equivalent to 1000 simulated SPECmail users.

Simulated user IMAP sessions were also generated using a Poisson arrival process with a rate of 5 sessions per minute; in each session, the simulated user logs in, lists unread messages, chooses a random (large) subset of new messages to download, then chooses a random (small) subset of those downloaded messages to delete. Both the IMAP and SMTP workload generators were designed to run open-loop, and, unlike most existing e-mail benchmarks like SPECmail2001 [118] and Netscape's Mailstone [81], were constructed to continue to function properly even if the IMAP or SMTP server processes were unavailable or misbehaving.

10.4.3 System configuration

All of our benchmark sessions used identically-configured e-mail servers. Each server ran Debian Linux with the 2.4.18 kernel, Sendmail 8.12.9 as an SMTP server, UW-IMAP 2001.315 as the IMAP server, and MIMEDefang 2.37 with SpamAssassin 2.55 as the system-wide SPAM filters. The servers were run on virtual machines under VMWare GSX Server 2.0.1 on a 4x500MHz Pentium-III-based server with 2GB of DRAM running Linux 2.4.18 (the same machine listed as "undo proxy" in Table 6-2 on page 89). All e-mail and mailspools were stored on a Network Appliance F760 filer connected via gigabit Ethernet (the "time-travel storage server" in Table 6-2). The workload was generated on a separate 667MHz Pentium-III-based machine with 256MB of DRAM, running Windows 2000, and also connected via gigabit Ethernet ("workload generator" in Table 6-2). The participants interacted with the virtual e-mail servers via a Windows 2000-based console using ssh and Outlook Express. The console machine had a second video display attached that allowed the experimenter to unobtrusively monitor the participant's progress from a location out of the participant's line of sight.

The decision to use virtual machines for our e-mail servers vastly simplified the logistics of the experiments. First, we could run multiple virtual machines in parallel, enabling us to overlap load generation and fault injection from one session while the participant completed the other session. We could configure a separate virtual machine image for each possible failure scenario, making the error injection stage simpler by not requiring that a single server support all possible error scenarios. More importantly, virtual machines can easily be restored to a clean state by simply restoring their disk images and rebooting them. This property allowed us to give our participants free reign to make any changes they desired, as we could easily discard those changes and start each experiment from the same known-good state.

But the most significant advantage to using virtual machines in a recovery benchmark is that they can be easily cloned from an existing non-virtual server, maximizing the realism of the benchmark environment. We took advantage of this capability by cloning our benchmark e-mail servers from a deployed system; doing so ensured that our participants saw a realistic server environment with all of the helpful tools and distracting vestigial state remnants that would be present in a real-world environment.

10.5 The Dependability Consequences of Three-R's Undo

Using the setup described in the previous section, we were able to expose and evaluate the dependability impact of providing Three-R's undo to the human operator of an e-mail store service. We measured dependability in terms of **availability** and **correctness**. An e-mail system is available if it accepts incoming SMTP connections and if users can connect to it to retrieve mail via IMAP. It is correct if, *at the end of the benchmark*, all incoming e-mail has been processed correctly, without any lost or mishandled messages. Our availability metric is instantaneous, producing a binary value for each generated e-mail message (for SMTP) and simulated user session (for IMAP). Availability is measured in real time as the benchmark progresses.

Our correctness metric is a bit more complicated: it has the flavor of an instantaneous metric, as it produces a binary result for each piece of incoming e-mail that is generated, but it is evaluated at the end of the benchmark rather than in real-time. Defined in this manner, correctness more accurately reflects the semantics of e-mail and of the end-user experience: the SMTP e-mail protocol guarantees eventual delivery, not instantaneous delivery, and delays in e-mail delivery are commonplace even in properly-functioning e-mail systems. By defining the e-mail system as correct if all incoming e-mail is eventually delivered correctly (within, in this case, the 30-minute trial period), we match the protocol guarantees and allow leeway for recovery approaches (like undo) that take advantage of e-mail's relaxed synchronicity semantics. Note that when we plot correctness against time, we will treat it as if it is instantaneous; the time coordinate of a correctness result indicates the time that a particular e-mail message was generated, but the correctness value for that time point (as well as all others) is actually computed at the end of the benchmark. Note also that correctness implies SMTP availability, but not vice versa.

Our benchmarking infrastructure allowed us to collect two different flavors of dependability results from our experiments: per-participant longitudinal data showing the time-varying dependability behavior of the system as the recovery process takes place, and cross-participant summary data useful for direct system-to-system comparison. The summary data can be used for hypothesis testing—in our case, to validate the hypothesis that undo-based recovery improves net dependability—while the longitudinal data provides the details explaining why and how the hypothesis holds.

10.5.1 Longitudinal results

We begin our study the dependability impact of undo-based recovery by looking at the longitudinal data. Figure 10-2 plots correctness and availability over time for one particular participant's benchmark session. This participant was asked to recover from the first scenario in our faultload (the misconfigured SPAM filter), and the results are typical of other participants. The two sets of graphs correspond to the two experiment trials: in the right-hand set, the participant used the undo tool, whereas in the left-hand set the undo tool was not made available.

Figure 10-2 illustrates both the power of undo-based recovery to improve dependability and the power of our human-aware benchmarking approach to reveal that dependability impact. First, we can see that the undo-driven recovery process significantly improves the e-mail system's correctness under this failure scenario by reducing the number of incorrectly-dropped messages compared to the non-undo-based recovery. Furthermore, the benchmark illustrates how the undo facility achieves this advantage: it shows how, with undo, recovery extends retroactively to the point where the fault occurred, whereas the non-undo recovery at best can only correct errors following the point where recovery begins. The results thus validate the benefit of the retroactive repair capability made possible by Three-R's undo.

Furthermore, the benchmark results reveal that, despite its correctness benefits, undo-based recovery still has some weaknesses in terms of overall dependability: unlike non-undo recovery, it causes a temporary drop in IMAP and SMTP service availability, and still allows a number of messages to be handled incorrectly. This drop in dependability was anticipated and occurs while the e-mail server's virtual machine is being rebooted during the rewind process; the reboot is necessary in order for the server to refresh its state from the rewound copy of hard state stored on the Network Appliance

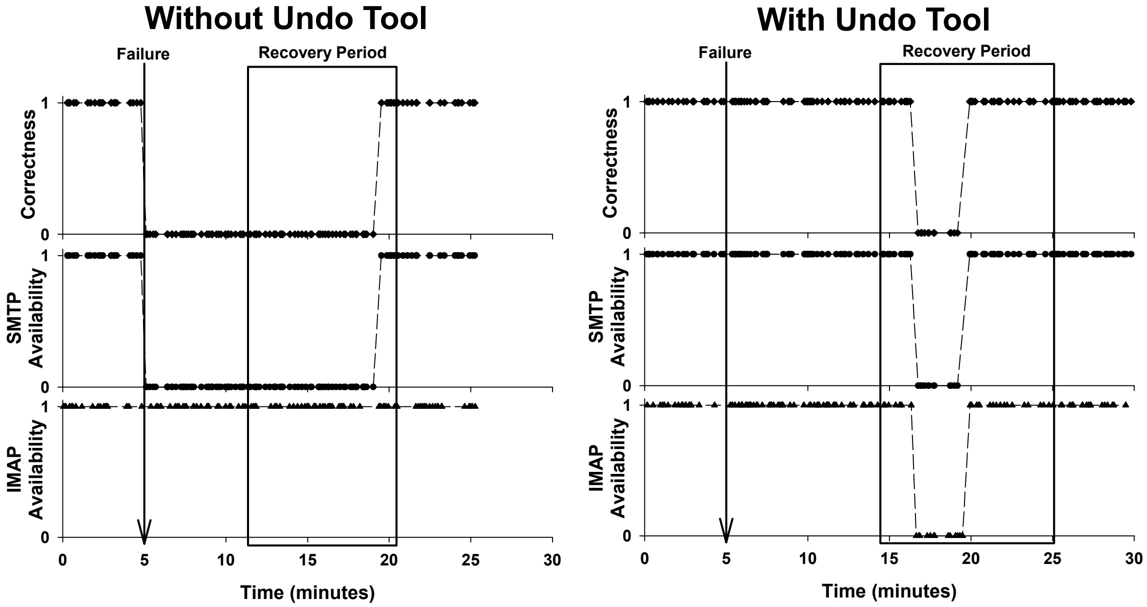


Figure 10-2: Time-varying dependability during recovery for one benchmark experiment. The graphs plot instantaneous correctness and availability over time for a benchmark using failure scenario #1 (filter configuration error). The recovery period begins when the participant is informed of the failure, and ends after 30 minutes or when the participant declares success. Note that while availability is measured as the benchmark progresses, correctness is measured at the end: the system is correct at time *t* if a message originally sent at time *t* has been properly received, handled, and stored on disk by the end of the benchmark.

filer. During the reboot, the undo proxy refuses to service SMTP or IMAP connections, resulting in unavailability and, consequently, incorrect behavior. In retrospect, we could have fixed the SMTP unavailability and correctness drops relatively easily, by modifying the undo proxy to queue incoming SMTP DELIVER verbs during the reboot process, much as they are queued and deferred while the system is rewound during an undo cycle.

The period of IMAP unavailability during reboot is harder to eliminate, since IMAP sessions are synchronous and interactive and cannot be easily deferred at the proxy, unlike asynchronous SMTP sessions. One option to deal with the IMAP unavailability might be to accept IMAP connections at the proxy and simulate a single read-only Inbox mailbox containing an explanatory message that the system is currently recovering from a failure and will be available shortly. While this approach does not provide access to the user's stored e-mail, it does at least provide the user with information as to why the stored e-mail is unavailable, and potentially with an estimated time at which full service will be restored. Finally, a complementary approach would be to improve the e-mail store service's reboot time (perhaps by using a journaled or log-structured file system), or to modify the system to be able to refresh its state without a full reboot, as in recursive restartability [18].

As is evident from the results in Figure 10-2 and the discussion above, our human-aware recovery benchmarking methodology succeeds in exposing the dependability consequences of undo-based recovery, and provides a direct quantitative comparison of two different human-driven recovery approaches, one with undo and one without. Furthermore, it is powerful enough to expose the detailed dependability-related behavior of the system during the recovery process: in our case, it exposed the availability downsides of our reboot-based rewind implementation, quantified its effects,

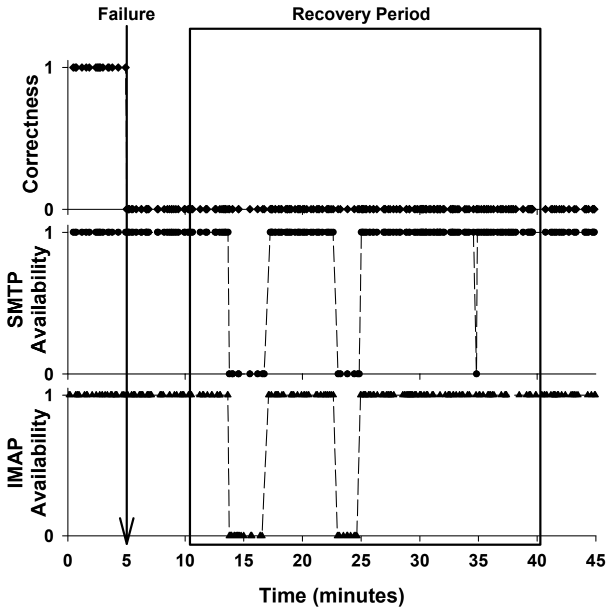


Figure 10-3: Benchmark in which user fails to recover. This figure, in the same format as Figure 10-2 but depicting failure scenario #2 (failed upgrade), illustrates the availability impact of a failed attempt to use the undo-based recovery tool, highlighting the fact that simply trying and abandoning the tool has dependability consequences.

and suggested where it might be worth spending more development effort to improve the undo tool's dependability behavior.

Figure 10-3 shows another example of longitudinal data that again provokes a new insight into the undo tool's behavior. In this case, the participant began using the undo tool, but got sidetracked by a spurious error message from the test e-mail client and ended up cancelling the undo process, in the end failing to recover from the problem scenario in the allotted time. During both the initial rewind and the later cancel operations, the undo manager rebooted the e-mail store server, temporarily dropping its availability just as discussed above.

What we learn from this example is that there are non-trivial availability consequences for simply *trying* the undo tool, without completing its recovery process. This is an insight that could be obtained only through a human-aware benchmark like the one we performed, as it depends on a practical usage pattern of the tool that was not anticipated in its design. The results suggest that we should again investigate ways to reduce the availability impact of service reboots, and perhaps even to rearchitect the undo cycle so that any availability impact comes only when its user completes the recovery, rather than when the recovery process is first invoked. Perhaps this could be accomplished by performing the undo cycle on a cloned copy of the original server, then swapping the cloned server for the original one when the operator chooses to commit the undo cycle.

10.5.2 Summary results

In order to compare dependability with undo-based recovery to that with the baseline system, we have to look at the aggregate results from all of our participants' sessions. To do this, we compute the total number of mishandled e-mail messages and the total number of failed attempts to contact the e-mail service (a measure of unavailability) for each benchmark session and each participant, starting five

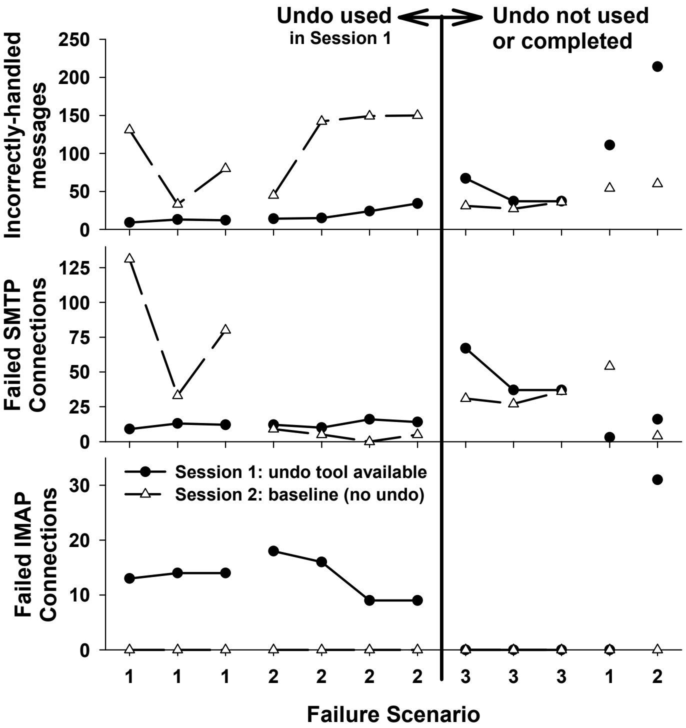


Figure 10-4: Benchmark summary results. The graphs plot overall correctness and availability metrics for all benchmark sessions, enabling a direct comparison of undo-based recovery to the baseline case where the undo tool was not available for use. Each point on the *x*-axis represents the results for an individual human participant. The results show a clear correctness benefit when undo-based recovery is used, but also point out that in some cases, undo-based recovery can reduce availability compared to the baseline.

minutes before the participant began recovery and ending five minutes after the participant signaled that the problem had been fixed. Figure 10-4 plots these results by participant, graphically showing the comparison between each participant's first session (with undo) and second session (the baseline, without undo). We have segregated the subjects by the failure scenarios they were given, and have split off the cases where the subjects chose not to use or complete the undo-based recovery process.

Correctness. Figure 10-4 shows the clear dependability benefit of undo-based recovery in terms of correctness compared to the baseline system. In the 7 cases where undo was used, the number of incorrectly-handled messages is always less than half of the corresponding result for the baseline system. Furthermore, our self-comparison approach lets us observe that variance is significantly reduced with undo-based recovery, indicating that the tool scales well across the expertise range of the participants, and suggesting that it can make effective recovery more accessible.

The 5 cases where undo was not used break down into two sets. The first three cases correspond to the third failure scenario listed above (crashed Sendmail process), for which the undo-based recovery tool is not useful. All of the participants realized this fact and none attempted to use undo-based recovery. Thus, the comparison results in Figure 10-4 show only the learning-curve effect, or the systematic bias introduced by our experiment design: the baseline data point was collected immediately after the undo data point using the same failure scenario, so the baseline result is better than the non-baseline, and likely reflects the best manual recovery that the participant could perform. It is interesting to note that even the best baseline result from these cases results in more incorrect messages than all but one case where undo was used for the other scenarios, and suggests extending undo-based recovery to cases like this third failure scenario. Finally, the remaining two cases correspond to scenarios where undo would have been useful, but where the participant chose not to use or complete the undo recovery process. In these cases, the baseline correctness results are significantly better than the non-baseline, again due to the learning curve effect, but are worse than the corresponding results from participants who did use and complete undo-based recovery.

A statistical analysis of the correctness results shows that our small participant pool was sufficient to produce statistically-conclusive results. In brief, we can model the benchmark experiments as a series of binomial trials, one for each participant [120]. In this model, each participant's sessions constitute a single binomial trial, like a coin toss, with either a positive result (if the system is more dependable in session 1, with undo available) or a negative result (if the system is more dependable in session 2, with undo not available). In the model, we assume that there is an unknown true probability p with which a participant achieves a positive result. We actually measure $\hat{p}$, an estimate of p, given by the fraction of participants that achieve a positive result on trials where undo could potentially be useful, namely scenarios #1 and #2.

To evaluate if our results provide statistically-conclusive evidence of a dependability benefit to undo, we must show that the true probability p is greater than 0.5—in other words, that the observed increased dependability is not simply due to chance. We do this by constructing a 95% confidence interval for p, and verifying that its lower bound is greater than 0.5 [120]. For a binomial model, the 95% lower confidence bound for p is given by:

$$p > \hat{p} \angle Z_{0.975}\mathrm{SE}(\hat{p})$$

where $Z_{0.975} = 1.96$ is the 0.975th quantile of the normal distribution and $\mathrm{SE}(\hat{p})$ is the sample standard deviation of $\hat{p}$, given by $\mathrm{SE}(\hat{p}) = \sqrt{\frac{\hat{p}(1 \angle \hat{p})}{n}}$. For our data from scenarios #1 and #2, we have $\hat{p} = 0.778$ and $n = 9$, since 7 of the 9 participants successfully improved dependability using undo. Note that this is a conservative analysis, since the two participants that achieved negative results with undo either chose not to use the tool or did not complete the undo process.

Plugging in, we find the lower bound of the confidence interval to be $p > 0.506$, indicating that we can make a statistically-significant conclusion at the 95% confidence level. Using the Wald statistic to test significance [120], we get a p-value of 0.045, indicating less than a 5% chance of error and confirming our analysis.

As an aside, it is interesting to analyze how many subjects would have been needed had we used the fully randomized approach instead of a self-comparison approach. To perform this analysis, we model a sequence of randomized trials using a normal 2-sample model [120], implying that the correctness values (count of incorrectly-handled messages) are normally distributed across subjects with a true mean value μ_1 when undo is available and a true mean value μ_2 when undo is not available, both with common standard deviation σ. Then we are interested in the parameter $\tau = \mu_2 \angle \mu_1$, which measures the average decrease in incorrectly-handled messages resulting from undo, and want to show that $\tau > 0$ with at least 95% confidence. To determine how many participants are needed to draw that conclusion, we can use the standard formula for power analysis at the 95% confidence level in a normal two-sample model [120]:

$$n \geq \left(\frac{0.95 \cdot \sigma \cdot (Z_{0.95} \angle Z_{0.05})}{\tau}\right)^2$$

Here, Z_q is the q'th quantile of the normal distribution. We approximate σ with the measured sample standard deviation S of the correctness metric, which works out to be 54.85 messages. We can approximate τ from our data, although in a true randomized design the learning-curve bias would not be present, making τ slightly larger; our estimate of n is therefore likely to be larger than the true value for a randomized experiment. With this approximation, τ is 44.22. So, the number n of participants needed is:

$$n \geq \left(\frac{0.95 \cdot 54.85 \cdot (1.645 + 1.645)}{44.22}\right)^2$$

$$n \geq 15.03$$

So at least 15 participants would have been needed just for scenarios #1 and #2 had we used a randomized trial; in comparison, our self-comparison-based experiment design needed only 9. Thus, we clearly see the benefit of our self-comparison-based experiment design in reducing subject pool size while still providing conclusive results.

Availability. Returning to the summary results in Figure 10-4, the data for availability illustrates one of the limitations of undo-based recovery. Except for one case (failure scenario #1), undo-based recovery does not offer an availability advantage over baseline recovery, and, especially for IMAP, can actually hurt it. In the one case of failure scenario #1, the undo-based recovery data shows an availability benefit; this is a side-effect of the way that the undo tool proxies incoming SMTP sessions, ensuring availability even when the SMTP server itself is misbehaving. In the end, the best conclusion we can draw here is that future work on the undo-based recovery approach should concentrate on improving the system's availability during the recovery process for both IMAP and SMTP protocols, as discussed above in Section 10.5.1, so that the obvious correctness benefits of the approach are not lost as a result of poorer availability.

10.6 Related Work

Dependability benchmarking has enjoyed a new-found spotlight in the literature in the past few years [14] [16] [57] [74] [130] [138], and our recovery benchmarks borrow heavily from the basic methodology that researchers, including us, have helped develop. However, there is only a limited amount of existing research work on benchmarks designed to measure the *human* components of dependability for server systems. Our work in this dissertation is the first that we know of to involve human participants in the dependability benchmarking process, and the first to adapt that idea into a practical methodology for measuring the dependability of human-assisted recovery.

While we are the first to directly involve human participants, others have begun to attack the challenge of incorporating other human aspects of dependability into benchmarks, primarily by attempting to simulate and inject the effects of errors made by operators while performing proactive maintenance tasks. In particular, Vieira and Madeira have developed a faultload for database management systems that mimics common human errors made by database operators, and have studied the recovery behavior of database management systems in response to those faults [128] [129] [130]. However, Vieira and Madeira's approach does not use human participants and therefore can only evaluate recovery mechanisms that work without human involvement.

Likewise, Zhu *et al.* have proposed benchmarks that can evaluate recovery mechanisms in general server systems, but their approach is limited to recovery processes instigated by crashes and hardware failures [137] [138]. Zhu *et al.* also do not address the human component of the recovery process, assuming that the system is capable of recovering on its own. Furthermore, neither Zhu's nor Vieira's methodology can provide the kind of insight that ours can offer into the dynamic, time-varying dependability consequences of the recovery process.

Turning to other communities, there is obviously a great deal of similarity between our methodology and behavioral research methodologies used to study human-computer interaction, and indeed our approach was heavily influenced by HCI techniques such as those described in Landauer's excellent survey [67]. However, unlike in HCI approaches where understanding the human's behavior is the main goal, the focus of our benchmarks is to quantify the *system*, with the human as a critical but indirect contributor to its behavior. In that sense our work is most similar to work in the security community on the effectiveness of security-related UIs, such as Whitten and Tygar's study of PGP [134].

10.7 Summary and Discussion

We began this dissertation by positing that undo-based recovery could improve the dependability of service systems. In this chapter, we have finally validated that claim. To do so, we had to first develop a benchmarking methodology that would allow us to evaluate the dependability of human-assisted recovery tools and processes, like undo. Our human-aware recovery benchmarks combine aspects of traditional dependability benchmarking with techniques from studies of human-computer interaction, incorporating human participants in the benchmark experiments yet still producing typical dependability-related metrics like availability and correctness as results. The methodology produces both summary results, useful for comparing the dependability of multiple systems or recovery approaches, and detailed longitudinal results that expose the time-varying dependability behavior of the recovery strategy being evaluated. Careful experiment design and controlled selection and training of participants allows us to achieve these useful results despite the variability introduced by human participants; in particular, our approach of comparing human participants to themselves allows us to trade a systematic bias for a better understanding of how variance affects the system under test, and lets us achieve statistically-conclusive results with a smaller subject pool than might otherwise be needed.

When we applied our newly-developed methodology to undo, as implemented in the prototype undoable e-mail store service discussed in Chapter 6, we found that it was able to verify the hypothesis that undo-based recovery improves dependability compared to traditional, non-undo-based recovery. The dependability benefit was primarily reflected in correctness: our human operator participants were able to use the retroactive repair capability of Three-R's undo to extend recovery all the way back to the failure point, restoring correct handling for e-mail messages that were lost or mishandled even before the recovery process began.

On the other hand, the benchmarks also revealed that the Three-R's undo cycle as implemented in our prototype has negative consequences for availability compared to non-undo-based recovery. The longitudinal data provided by the benchmarks enabled us to identify why this availability impact was occurring and suggested new directions for improvement of the undo facility's implementation. Of the negative availability consequences, all but the IMAP availability drops could be easily fixed by altering the undo proxy to queue incoming mail during service reboots (a minor change); fixing the IMAP availability drop requires either more significant architectural changes or taking the simpler approach of explaining the unavailability to end-users without correcting it. Note, however, that for many uses, correctness trumps availability in importance, and so our undo facility could provide significant dependability enhancements even as it stands today.

Finally, while our current methodology is effective, it is still only a first step into the nearly-unexplored domain of benchmarking the human-related aspects of server dependability. It is still a costly methodology, involving much of the overhead of a traditional user study, including surveying practicing administrators, recruiting participants, and deploying testbed systems that mimic a real-world operations environment. Unlike traditional benchmarking, where the experimenter controls the recovery tools, our approach also requires that tools be robust and usable enough that they can survive unsupervised use with inexpert participants; this can mean a significant amount of extra work to build decent user interfaces and to remove corner-case bugs that might otherwise be overlooked in a traditional benchmark.

On the other hand, our proposed experimental designs can produce conclusive results with only a handful of human participants, and even fewer may be needed if the goal is only to identify potential dependability effects without worrying about proof to statistical certainty. Furthermore, the cost of *not* performing human-aware recovery benchmarks can be as high or higher than carrying them out. We routinely deploy services into the field today without careful consideration of how they behave in response to human error and during human-driven recovery. As a result, dependability suffers, and companies can lose millions of dollars due to outages, reputation damage, and lost customer business. If human-aware dependability benchmarks can avoid some of these outages by identifying dependability problems and validating solutions beforehand, then the cost that they entail may well be worth paying.

It is our hope that future research can find ways to reduce the costs of human-aware dependability benchmarking, perhaps by teasing apart the benchmark components that truly require human intervention from those that can be adequately simulated. Perhaps there is a way to use mock-ups of recovery tools to get initial insight into their applicability without having to pay the cost of building complete, robust implementations. Further study is also needed into the issue of managing human variability: while our results with undo show that the comparative benchmarking approach can trade the detrimental effects of variability for a manageable systematic bias, variability will still prove a problem for single-system benchmarking and for comparative benchmarking where the dependability differences between systems are slight.

Despite these challenges, we believe that the benefits and significance of human-aware dependability benchmarks are evident, and we look forward to the day when they can be found in every dependability researcher's toolbox.

Chapter 11

Directions for Future Work

> "Every great work of art has two faces, one toward its own time and one toward the future, toward eternity."
>
> — *Daniel Barenboim*

The work described in this dissertation constitutes a first cut at system-wide undo for self-contained and nested services. Through our analysis, prototyping, and evaluation, we have demonstrated a feasible and effective mechanism for human operators to use to increase dependability. Despite our successes in developing and demonstrating system-wide undo, however, our work is no more than a first step across a new frontier. While we are the first to explore the concept of a full-system undo that preserves end-user work, and have clearly demonstrated that it is a concept worthy of pursuit, we have only illuminated a very small corner of a large design space, and even in that corner we have left many dark spots unexplored. Thus, we wrap up our exploration of system-wide undo with some ideas for future research, in the hope that greater understanding of its untapped design space can eventually be achieved.

11.1 Further Development, Optimization, and Analysis of Proxy-based Undo

The most apparent directions of future work are to continue the development of the proxy-based undo approach that we have started in this dissertation. Obvious initial extensions to our work would be to build verb sets and proxies for a wider variety of service applications, both to provide stronger evidence of its broad applicability and to start building up a library of common verb primitives; the latter could be used to help decrease the burden of developing the undo wrapper for new services, and could even provide a first step toward the goal of automatically-generated verb implementations based on machine-readable representations of services' external consistency policies.

Another obvious direction of future work would be to follow up on the many ideas for undo performance and availability enhancements that we have sprinkled throughout our discussions and analysis. For example, the negative availability consequences of undo that we identified in Chapter 10—particularly when the operator cancels an undesired invocation of undo—could be addressed by rearchitecting the undo manager to queue asynchronous verbs while the service system is rebooting/rewinding. Likewise, the internal architecture of the undo manager could be enhanced to provide better behavior under overload, potentially improving the maximum throughput of the undo system. Finally, there is a role for research into rewindable storage systems, both in exploring how to provide an efficient cancellable rewind operation, and in creating an API to support the hierarchically-structured rewindable storage needed for the hierarchical undo approach discussed in Chapter 9.

On the analysis front, a very interesting direction for future work is to explore the effects of paradox management on end-user satisfaction. While there is evidence from existing systems that users tolerate compensation willingly, it would be interesting to see the results of a user study designed to directly explore the question. These results could be invaluable to designers of undo functionality, as they would provide guidance as to what compensations are acceptable and what undo recovery guarantees are tolerable. Additionally, the study could provide an answer to the question of whether a global paradox-management policy—like the one used by our undo prototype—is sufficient. If not, then future researchers could explore having users define their own paradox-management policies, perhaps by selecting possible paradox scenarios from a list and indicating what compensation, if any, they would prefer for those scenarios. The verb framework would then be extended to incorporate those preferences into the recovery process.

11.2 Extensions and Alternatives to the Proxy-based Undo Model

If we take a step back from the corner of the design space holding our prototype architecture and implementation, we find many more speculative directions for future research. One entire area is to explore more complex structures of spheres of undo, using them to extend system-wide undo functionality to broader classes of applications and services. This work could start with implementing, refining, and analyzing the nested undo models from Chapter 9 for hierarchical services.

A more challenging direction to pursue is to extend the spheres-of-undo model to distributed services, like the example e-shopping service that Figure 11-1 depicts. While such a service could be wrapped in a single giant sphere of undo and treated using the techniques developed in this dissertation, a more natural approach is to divide it into multiple spheres of undo along organizational and functional boundaries. Figure 11-2 shows how this decomposition might be accomplished. Of course, once we introduce multiple spheres of undo, the service is no longer self-contained, and we need novel undo algorithms to handle the interactions between the spheres of undo. This is a challenging problem, akin to supporting distributed multilevel transactions in a database [47] or coordinated multicheckpoint rollback and recovery [36], but with the additional complexity of distributed paradox management. We have developed some initial thoughts and algorithms for how to support and implement interacting spheres of undo; a detailed discussion can be found in an accompanying technical report [15].

Returning to the more familiar realm of single-sphere undo, we find some more immediate directions for future research that extend the Three-R's undo model. For example, Three-R's undo

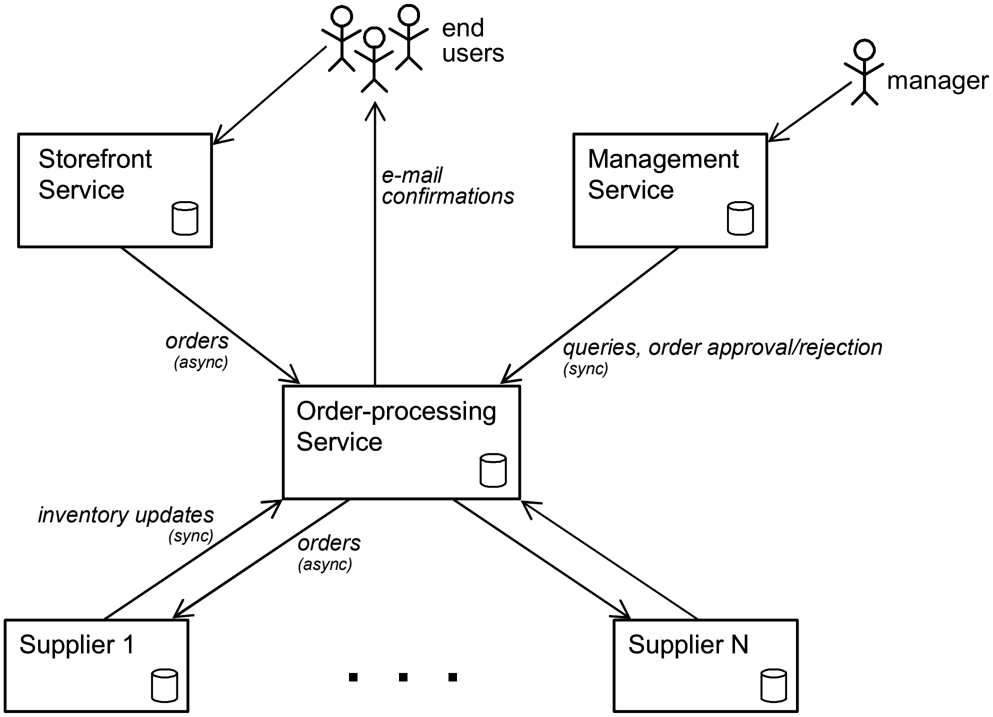


Figure 11-1: E-shopping service architecture. The diagram depicts the architecture of PetStore, an e-shopping system developed by Sun as a blueprint for online retailer applications [124]. PetStore is composed of three core services: a storefront web application used by customers, a management web application used by the system administrator to manage and approve orders, and a standalone order-processing application that accepts customer orders and fulfills them via a network of independent supplier services.

provides no way for operators to know ahead of time what the effects of an undo cycle will be in terms of number and degree of paradoxes; this limitation is seemingly unavoidable because of our replay-based approach where we discover and handle paradoxes dynamically as the system recovers. However, by leveraging virtual machine technology, it might be possible to simulate the results of the undo cycle before performing it on the actual system, thereby exposing the paradoxes and providing the operator with the desired feedback. Alternately, a summary of paradoxes could generated before the commit operation is invoked; this simpler approach provides the same feedback at the cost of decreased availability during an undo cycle.

An even more speculative direction might be to try to predict paradoxes without executing the rewind and replay processes at all. Doing so would require significant changes to the verb-based paradox-management framework, perhaps requiring pre-execution consistency checks rather than the simpler post-execution checks we use currently. Statistical sampling might play a role in reducing the cost and complexity of these checks as well, as might tighter integration between the undo facility and the service itself.

The latter point is one worthy of further investigation in its own right. In this dissertation, we built upon the assumption that the undo facility should be separate from the service itself, primarily for reasons of fault containment. Our implementation of undo gained significant complexity as a result—for example, we had to infer and manipulate execution ordering, extract context explicitly when not provided by the service protocol, and impose our own time-invariant unique names when the service failed to provide them. There is another entire region of the undo design space where

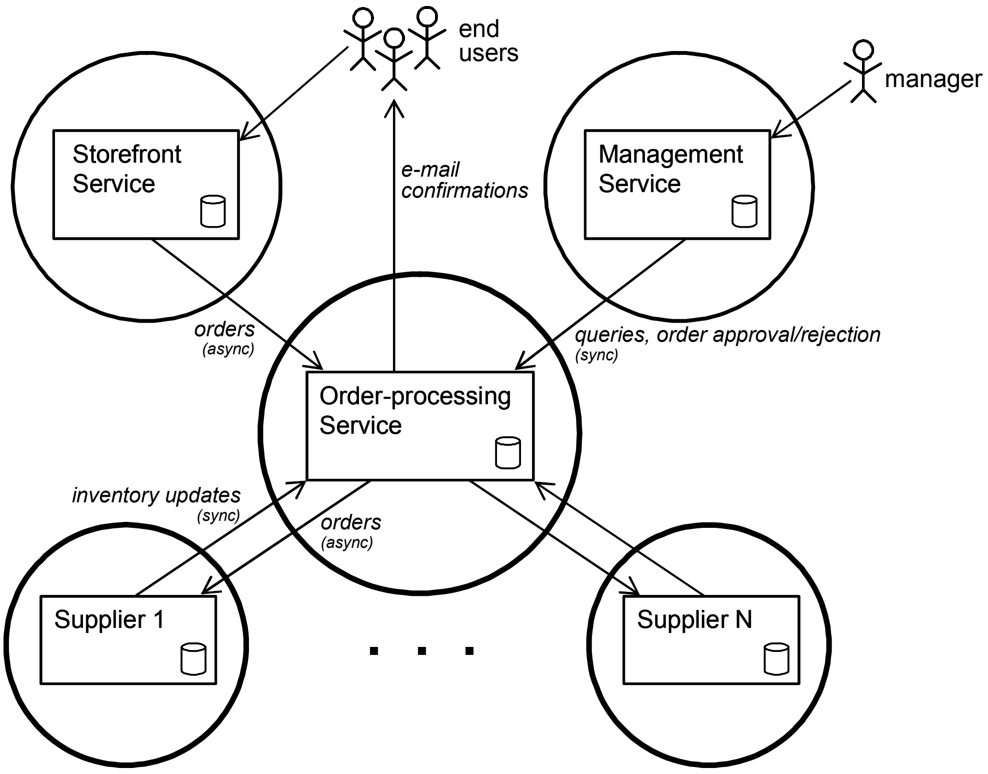


Figure 11-2: PetStore architecture as spheres of undo. Each subservice of the PetStore architecture naturally maps to its own sphere of undo. Providing undo across these independent spheres is a challenging research problem, as the unilateral undo of one sphere may cause paradoxes that affect other spheres. A multi-sphere undo system must choose to compensate for these paradoxes either by propagating the undo operation to all affected spheres—which may be difficult when the spheres are in different administrative domains—or by directly compensating for paradox-induced inconsistencies in a manner similar to that used in single-sphere undo.

Three-R's functionality is more tightly integrated into the service itself, and a promising direction of future work would be to explore that space. Intuitively, it seems that an integrated undo might sacrifice some recovery power for the performance and simplicity benefits of integration, but that sacrifice and its concomitant benefits need to be explored and quantified.

Another direction for future exploration is to extend the undo model from the simple state-based committable-cancellable undo that we chose in Chapter 2 to a more powerful model supporting branched timelines and administrative actions as first-class managed operations. As we discussed earlier in Chapter 2, branched timelines provide greater recovery flexibility by allowing the operator to choose amongst possible outcomes; including administrative actions as verbs would likewise increase flexibility by allowing operators to explicitly see what administrative changes had been made and later retroactively edit those changes. Both of these improvements to the undo model will require significant advances in verb representation, verb log storage organization, and verb generation; it would behoove future undo-system designers to perform a detailed user study to evaluate the effectiveness and comprehensibility of such advanced undo models—perhaps using a mock-up of the undo control interface—*before* investing the time and development cost into creating working implementations.

Finally, it would be interesting to explore a more sophisticated paradox management scheme, one that provided more than just the local compensations plus squashing that we currently offer. Ideally, it would be possible to extract the high-level intent of a sequence of user operations—checking new e-mail and deleting spam, as an example from the e-mail case—and use that understanding of intent to provide even more appropriate compensation should a paradox occur. The ultimate goal might be to produce a system with an intent-soundness recovery guarantee, relieving users of having to reconstruct their own intent should a paradox alter a piece of a previously-visible workflow.

11.3 Other Human-centric Methods for Improving Dependability

Finally, while we certainly believe that undo is an important mechanism for increasing dependability in human-operated service systems, it is by no means the only one. Surprisingly, there appears to be an entire unexplored space of tools and techniques that could significantly improve dependability by taking advantage of the human operator's presence rather than trying to eliminate it. Besides system-wide undo, we could provide operators with tools to help them understand how the system has changed over time, to identify and diagnose problems, to predict the impact of a change before it is made, and even to provide task guidance as the operator steps through a maintenance or recovery procedure. Some of these tools—like change analysis and problem detection—support undo by helping the operator understand when undo is necessary and to what time point it should be applied; the others are complimentary and help prevent problems that might require undo. All would significantly ease the human operator's burden in keeping service systems dependable.

We believe that many of these tools could be built using the same primitives that underlie the system-wide undo mechanism that we have developed over the course of this dissertation. In particular, verbs provide a context-independent encapsulation of work, and their associated paradox-management predicates provide a way of testing if that work was performed correctly. Verbs could therefore play a key role in impact analysis, problem detection, and diagnosis; if extended to cover administrative tasks they could also prove invaluable for change analysis and task guidance.

Thus, while our work on system-wide undo might seem like just an initial step in a little-explored space, it may well prove to have defined many of the fundamental primitives needed to build the dependable human-operated service systems that our computing infrastructure so desperately lacks. While we have identified only a few tentative first steps in this chapter, we believe that there is a great deal of fruitful work yet to be built upon the foundations created here, and we look forward to seeing it materialize in the future.

Chapter 12

Conclusion

> "If a problem has no solution, it may not be a problem, but a fact; not to be solved, but to be coped with over time."
>
> — *Shimon Peres*

In the search for increased dependability for the enterprise and Internet services that are fast becoming the dominant paradigm of computing, we would do well to keep in mind the wisdom that Shimon Peres so succinctly articulates in the quotation above. As we established at the start of this dissertation, one of the grand challenges for service dependability today is the human operator: operator error is the most significant and unaddressed impediment to the dependability and trustworthiness of modern services. While most have treated human operators as a problem, in reality their existence is a fact: results from psychology practically guarantee that humans will always have a role to play in managing computing systems, and that they will continue to make errors while performing their management duties.

Following Peres's advice, we must therefore move past trying to "solve" the "problem" of human operators, but rather accept them as facts, and develop mechanisms that both cope with the negative aspects and leverage the positive aspects of their behavior. The system-wide undo mechanism that we have developed in this dissertation is a first step toward Peres-inspired tools that cope with human operators and their error-proneness. System-wide undo constitutes a tool that human operators can use to recover after-the-fact from their own inevitable errors. It further takes advantage of the unique human capability of hindsight, allowing the human operator to effectively repair and recover from a host of other system failures, including state corruption and external attack.

Our approach to system-wide undo is made possible by the confluence of several novel ideas. First is the construct of spheres of undo, which provide the framework needed to understand how the timeline manipulations of undo affect non-undone entities like end-users and external services. We demonstrated how spheres of undo provide temporal and state-induced boundaries around standalone undoable services, and how they provide a powerful construct for extending our basic Three-R's undo metaphor to more complex systems like hierarchically-structured desktop platforms and distributed collections of interacting services.

From spheres of undo comes the second key innovation in system-wide undo: paradox management. Spheres of undo lead to a natural definition of paradoxes, as points where temporally-inconsistent data escapes from the bubble of time and state enclosed by a sphere of undo. In most existing undo-like systems, paradoxes are catastrophic and are avoided even at the cost of reduced undo capability. We take a different tack, recognizing that most services that deal with human end-users already have ways of absorbing or compensating for exposed inconsistencies, either because of inherently-weak guarantees—like the lack of ordering or timeliness guarantees for e-mail—or because the services encompass processes that have always been error-prone and thus incorporate fallbacks for when inconsistencies occur—as when an e-commerce or auction service absorbs the cost of a mishandled transaction under the overhead cost of doing business.

Our framework of verb-based paradox detection, compensation, and squashing is another key contribution of our work. Verbs represent a distillation of the important user interactions with a service, in some sense playing the role of an instruction set architecture for undoable services. Crucially, verbs provide a locus for annotating key user-service interactions with their corresponding external consistency guarantees, thereby forming the linchpin of a mechanism for automatic paradox management during undo-based recovery. Verbs provide a flexible framework that gives service designers control over the degree and extent of paradox compensation; they can provide an array of different recovery guarantees, ranging from a simple local compensation guarantee to a guarantee that no end-user work will be lost. Through this framework, our paradox-management approach significantly expands the applicability of undo-based recovery and allows human operators to recover effectively from problems even after their symptoms have been exposed to external users.

We have demonstrated that these abstract concepts of spheres of undo, paradox management, and verb-based policy definition can be implemented in a workable toolkit for wrapping system-wide undo functionality around essentially-unmodified standalone services. Our implementation approach favors undo power at the cost of performance, trading the performance advantages of tight integration with the service for the ability to recover from problems that go to the core of the service itself, including operating system state corruption and upgrades to the OS and service application. Through our analyses of e-mail and auction services, we have demonstrated that our approach is practical, and have presented initial evidence that the approach may have enough broad applicability to cover many of the most-significant service classes. While our prototype implementation suffers from technological limitations, particularly in the capabilities of the underlying rewindable storage layer and in the ability to perform efficient bulk replay of protocol operations, it does demonstrate that system-wide undo is within the realm of feasibility.

The results from our human-aware recovery benchmark of an undoable e-mail store service also demonstrate feasibility, in this case the feasibility of human-aware dependability evaluation in general. Our methodology illustrates how shrewd experimental design can turn human variability from a curse to a benefit, simultaneously reducing the number of human participants needed while exposing the relationship between operator skill and system dependability. More importantly, though, our benchmark results highlight the significant dependability benefits that are possible with system-wide undo, in particular pointing out the correctness improvements that result when it is possible to retroactively repair failures using system-wide undo.

Thus, we have achieved the goal we set out at the beginning of this dissertation: we have developed a means of improving service dependability through a recovery process that takes into account the human operations factor. As human operators will inevitably remain an integral part of large-scale service installations, at least for the foreseeable future, our work in this dissertation represents a pioneering and desperately-needed step towards greater dependability in the services that are increasingly mediating modern economic and social interaction. Now is the opportunity for progress; we look forward to seeing the positive changes that will arise as this work and future work that builds upon its foundations reach their full potential.

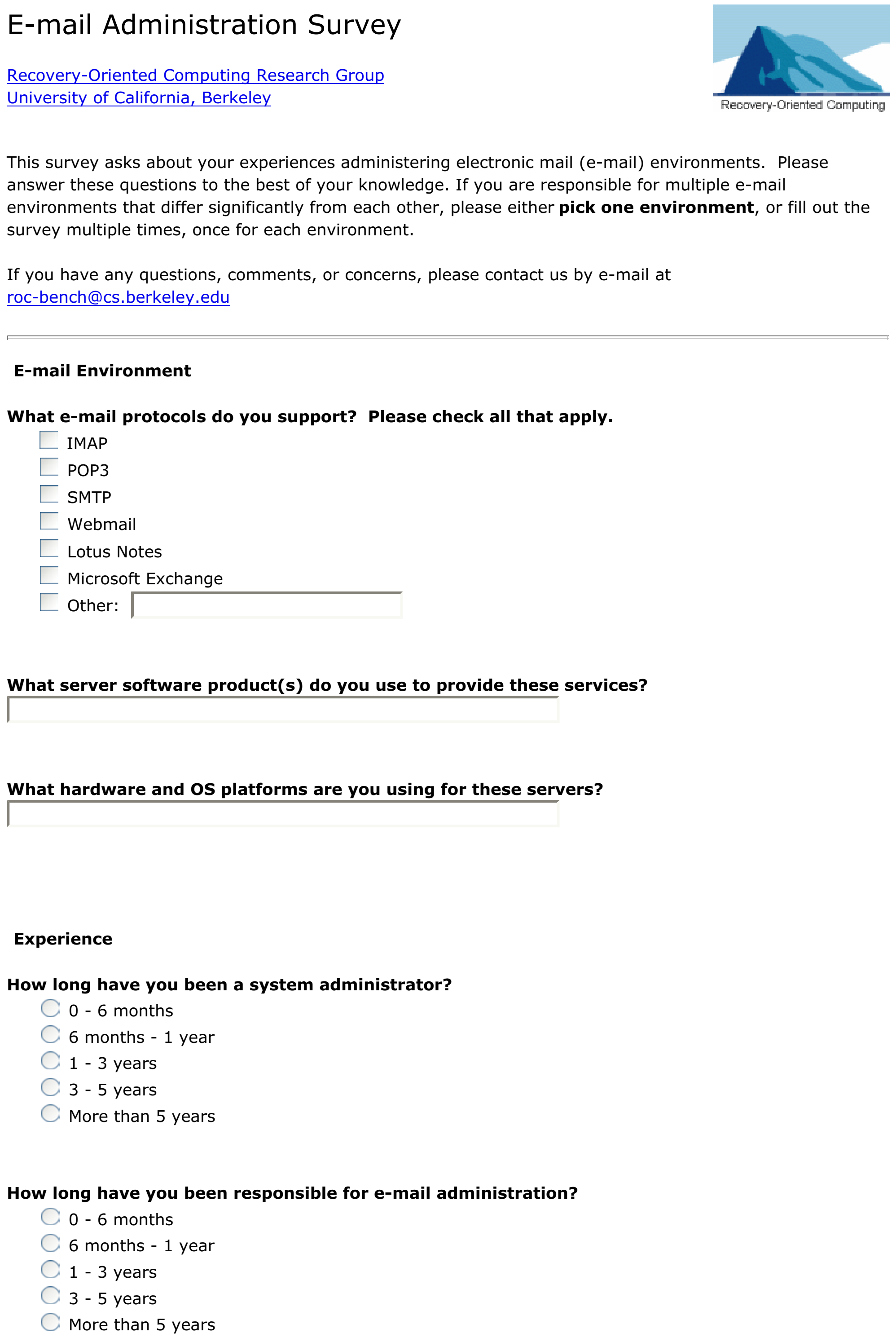
Appendix A

Materials For Recovery Benchmark of Undo

This appendix contains facsimiles of all of the materials we created and used during our human-aware recovery benchmark of our prototype undo tool, described in Chapter 10.

A.1 Survey of E-mail Administrators

The following pages reproduce the online survey form used to collect data from practicing e-mail system administrators.



E-mail Administration Survey

Recovery-Oriented Computing Research Group
University of California, Berkeley

This survey asks about your experiences administering electronic mail (e-mail) environments. Please answer these questions to the best of your knowledge. If you are responsible for multiple e-mail environments that differ significantly from each other, please either **pick one environment**, or fill out the survey multiple times, once for each environment.

If you have any questions, comments, or concerns, please contact us by e-mail at roc-bench@cs.berkeley.edu

E-mail Environment

What e-mail protocols do you support? Please check all that apply.

☐ IMAP
☐ POP3
☐ SMTP
☐ Webmail
☐ Lotus Notes
☐ Microsoft Exchange
☐ Other:

What server software product(s) do you use to provide these services?

What hardware and OS platforms are you using for these servers?

Experience

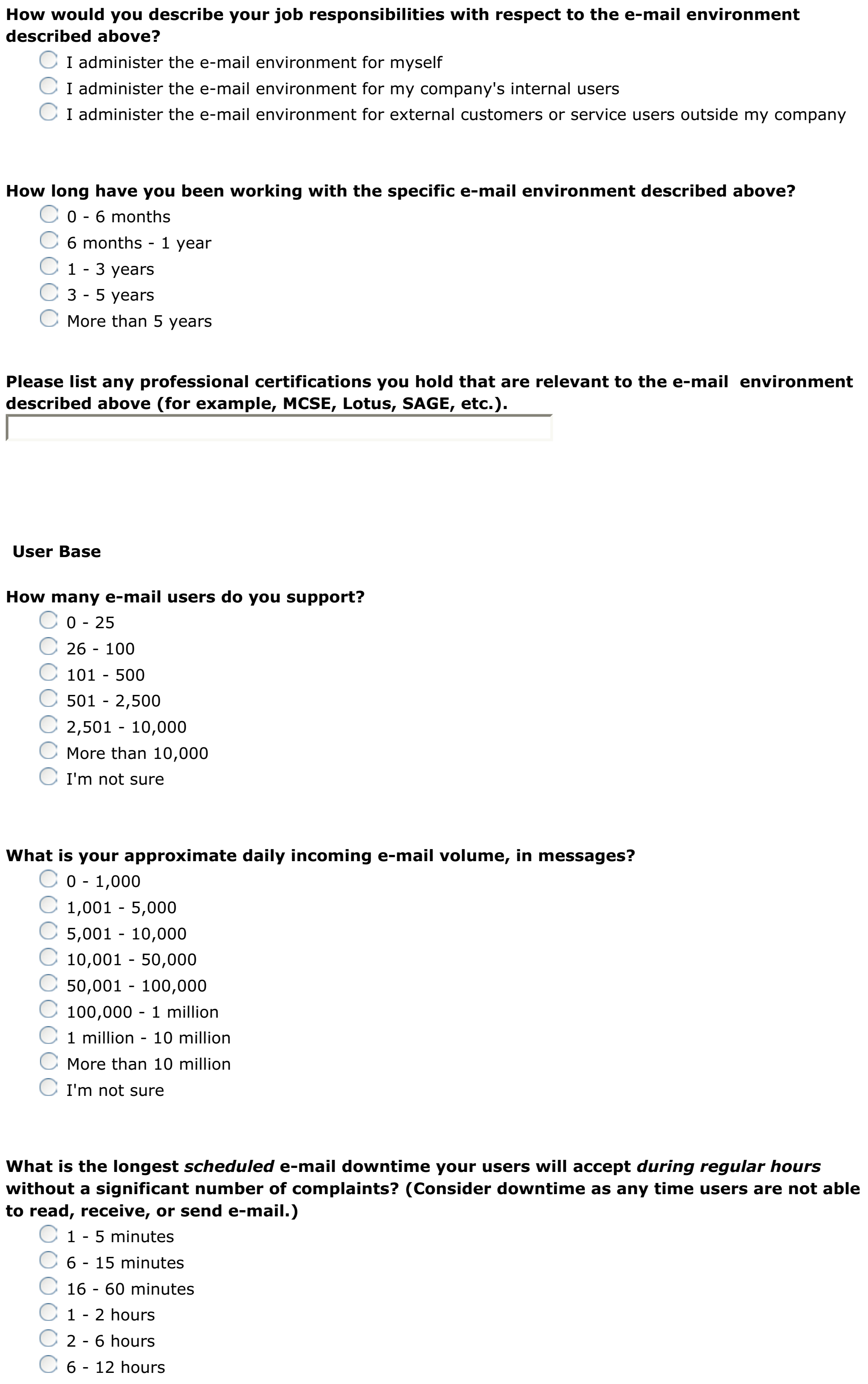
How long have you been a system administrator?

○ 0 - 6 months
○ 6 months - 1 year
○ 1 - 3 years
○ 3 - 5 years
○ More than 5 years

How long have you been responsible for e-mail administration?

○ 0 - 6 months
○ 6 months - 1 year
○ 1 - 3 years
○ 3 - 5 years
○ More than 5 years

Figure A-1: Survey of e-mail administrators. Page 1 of 7.



How would you describe your job responsibilities with respect to the e-mail environment described above?

○ I administer the e-mail environment for myself
○ I administer the e-mail environment for my company's internal users
○ I administer the e-mail environment for external customers or service users outside my company

How long have you been working with the specific e-mail environment described above?

○ 0 - 6 months
○ 6 months - 1 year
○ 1 - 3 years
○ 3 - 5 years
○ More than 5 years

Please list any professional certifications you hold that are relevant to the e-mail environment described above (for example, MCSE, Lotus, SAGE, etc.).

User Base

How many e-mail users do you support?

○ 0 - 25
○ 26 - 100
○ 101 - 500
○ 501 - 2,500
○ 2,501 - 10,000
○ More than 10,000
○ I'm not sure

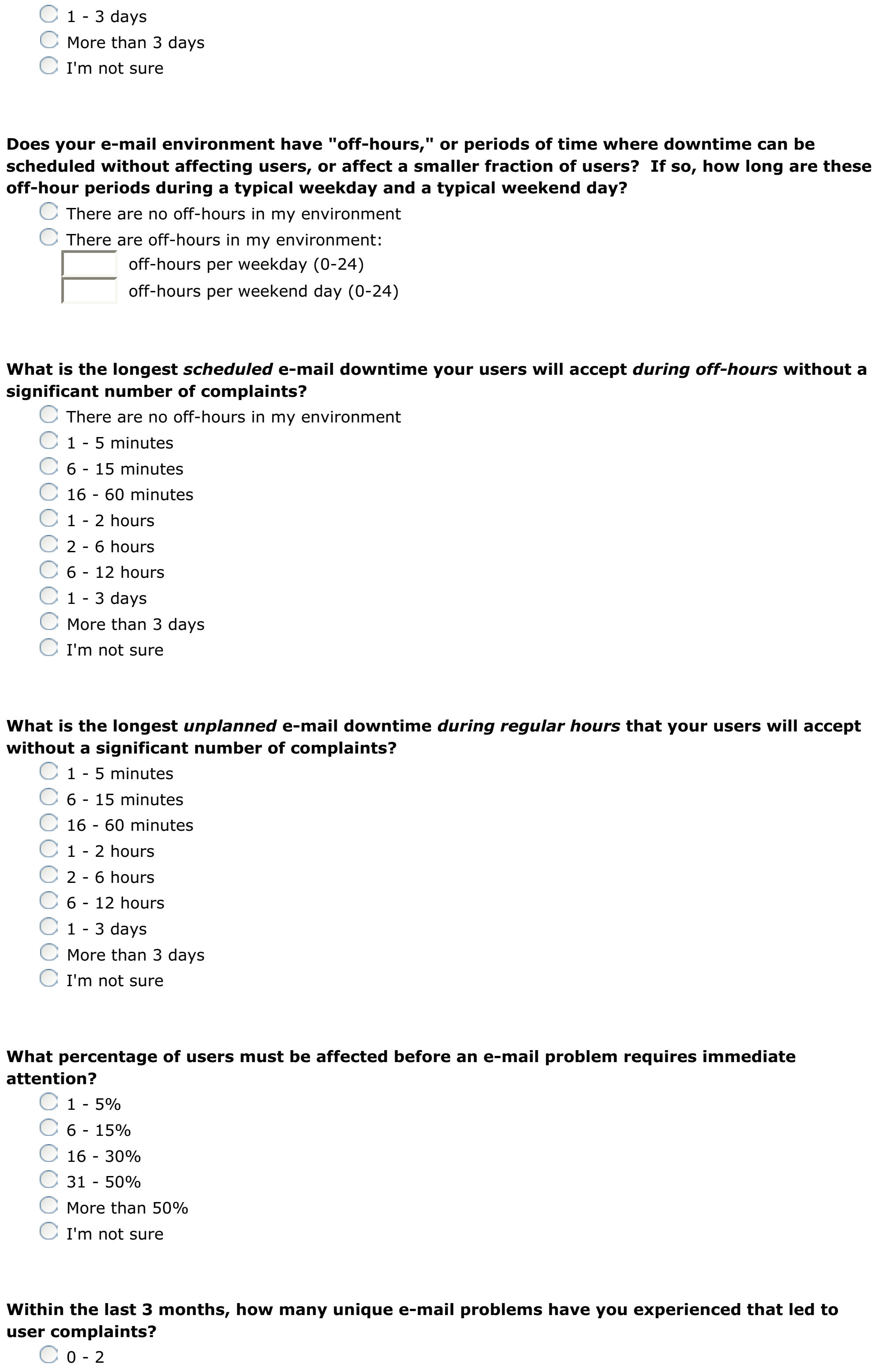
What is your approximate daily incoming e-mail volume, in messages?

○ 0 - 1,000
○ 1,001 - 5,000
○ 5,001 - 10,000
○ 10,001 - 50,000
○ 50,001 - 100,000
○ 100,000 - 1 million
○ 1 million - 10 million
○ More than 10 million
○ I'm not sure

What is the longest *scheduled* e-mail downtime your users will accept *during regular hours* without a significant number of complaints? (Consider downtime as any time users are not able to read, receive, or send e-mail.)

○ 1 - 5 minutes
○ 6 - 15 minutes
○ 16 - 60 minutes
○ 1 - 2 hours
○ 2 - 6 hours
○ 6 - 12 hours

Figure A-1: Survey of e-mail administrators. Page 2 of 7.



○ 1 - 3 days
○ More than 3 days
○ I'm not sure

Does your e-mail environment have "off-hours," or periods of time where downtime can be scheduled without affecting users, or affect a smaller fraction of users? If so, how long are these off-hour periods during a typical weekday and a typical weekend day?

○ There are no off-hours in my environment
○ There are off-hours in my environment:
□ off-hours per weekday (0-24)
□ off-hours per weekend day (0-24)

What is the longest *scheduled* e-mail downtime your users will accept *during off-hours* without a significant number of complaints?

○ There are no off-hours in my environment
○ 1 - 5 minutes
○ 6 - 15 minutes
○ 16 - 60 minutes
○ 1 - 2 hours
○ 2 - 6 hours
○ 6 - 12 hours
○ 1 - 3 days
○ More than 3 days
○ I'm not sure

What is the longest *unplanned* e-mail downtime *during regular hours* that your users will accept without a significant number of complaints?

○ 1 - 5 minutes
○ 6 - 15 minutes
○ 16 - 60 minutes
○ 1 - 2 hours
○ 2 - 6 hours
○ 6 - 12 hours
○ 1 - 3 days
○ More than 3 days
○ I'm not sure

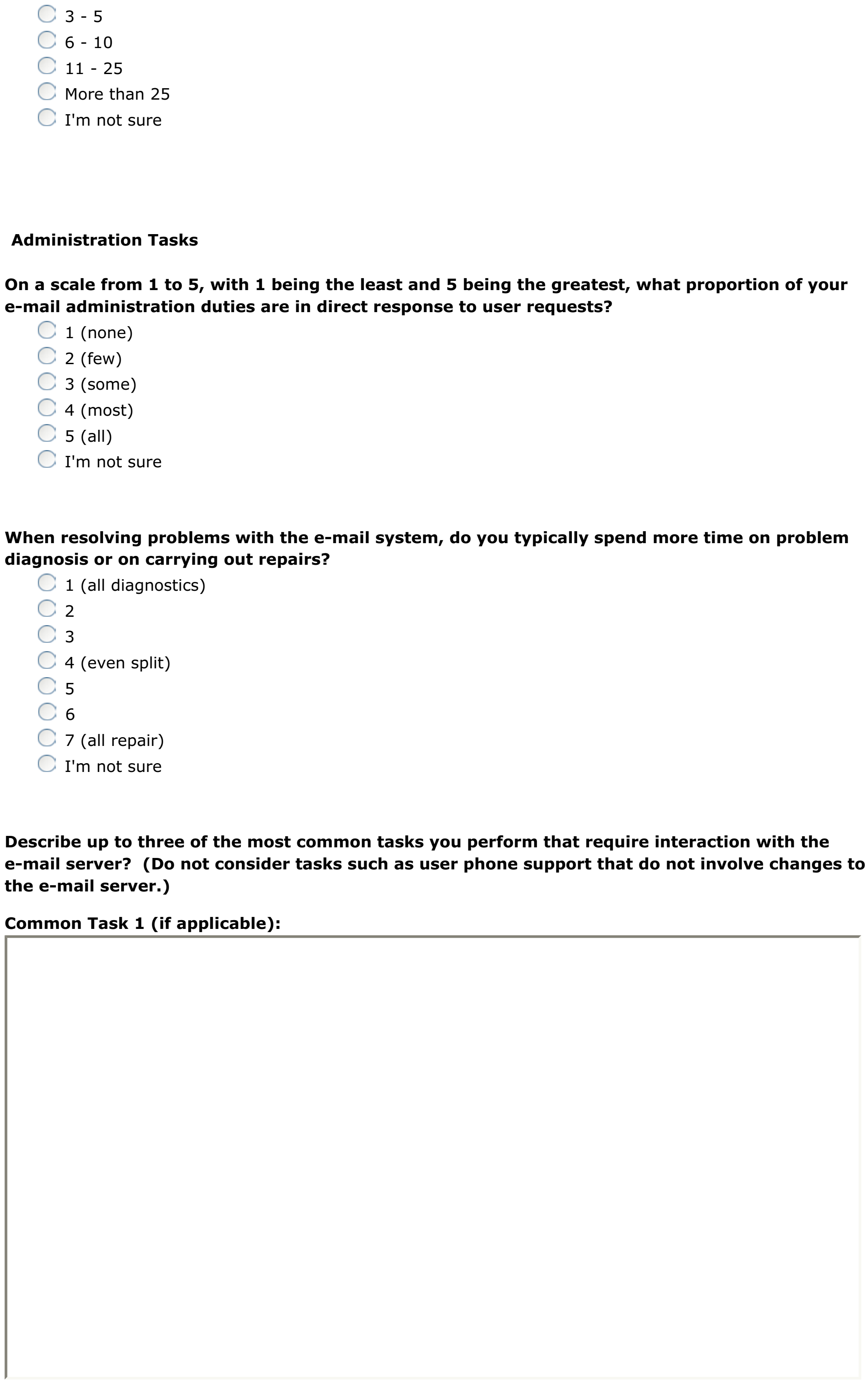
What percentage of users must be affected before an e-mail problem requires immediate attention?

○ 1 - 5%
○ 6 - 15%
○ 16 - 30%
○ 31 - 50%
○ More than 50%
○ I'm not sure

Within the last 3 months, how many unique e-mail problems have you experienced that led to user complaints?

○ 0 - 2

Figure A-1: Survey of e-mail administrators. Page 3 of 7.



○ 3 - 5
○ 6 - 10
○ 11 - 25
○ More than 25
○ I'm not sure

Administration Tasks

On a scale from 1 to 5, with 1 being the least and 5 being the greatest, what proportion of your e-mail administration duties are in direct response to user requests?

○ 1 (none)
○ 2 (few)
○ 3 (some)
○ 4 (most)
○ 5 (all)
○ I'm not sure

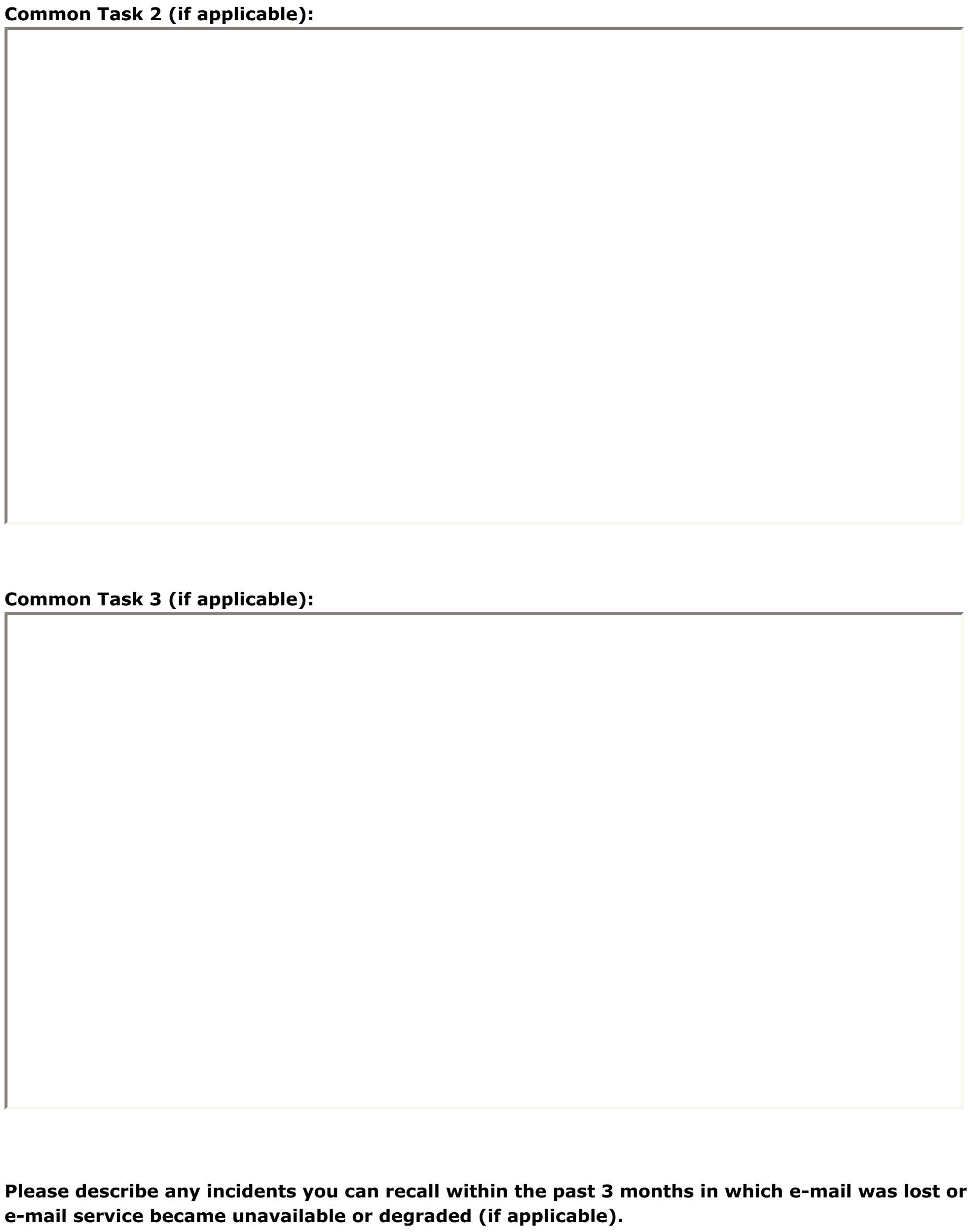
When resolving problems with the e-mail system, do you typically spend more time on problem diagnosis or on carrying out repairs?

○ 1 (all diagnostics)
○ 2
○ 3
○ 4 (even split)
○ 5
○ 6
○ 7 (all repair)
○ I'm not sure

Describe up to three of the most common tasks you perform that require interaction with the e-mail server? (Do not consider tasks such as user phone support that do not involve changes to the e-mail server.)

Common Task 1 (if applicable):

Figure A-1: Survey of e-mail administrators. Page 4 of 7.

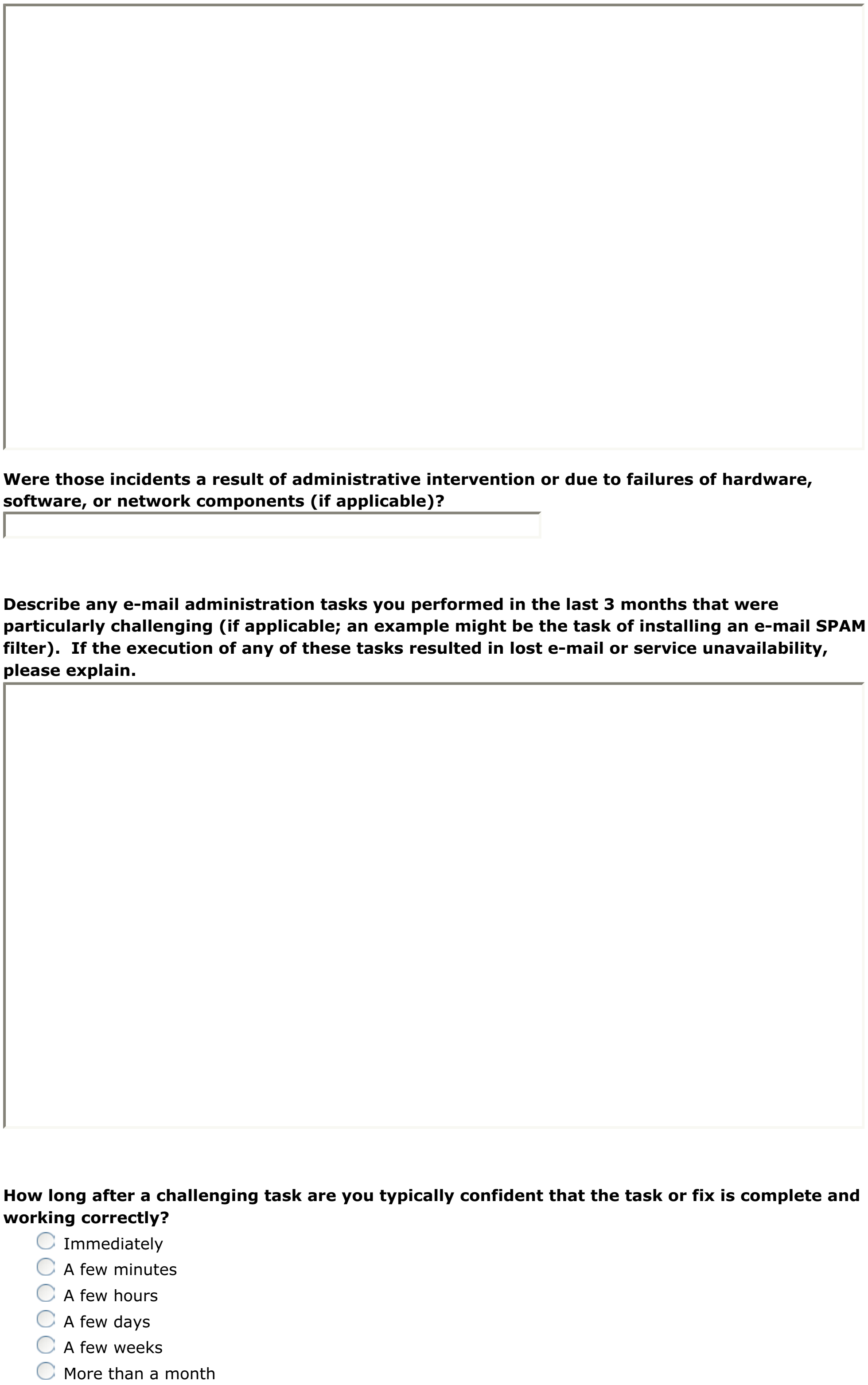


Common Task 2 (if applicable):

Common Task 3 (if applicable):

Please describe any incidents you can recall within the past 3 months in which e-mail was lost or e-mail service became unavailable or degraded (if applicable).

Figure A-1: Survey of e-mail administrators. Page 5 of 7.



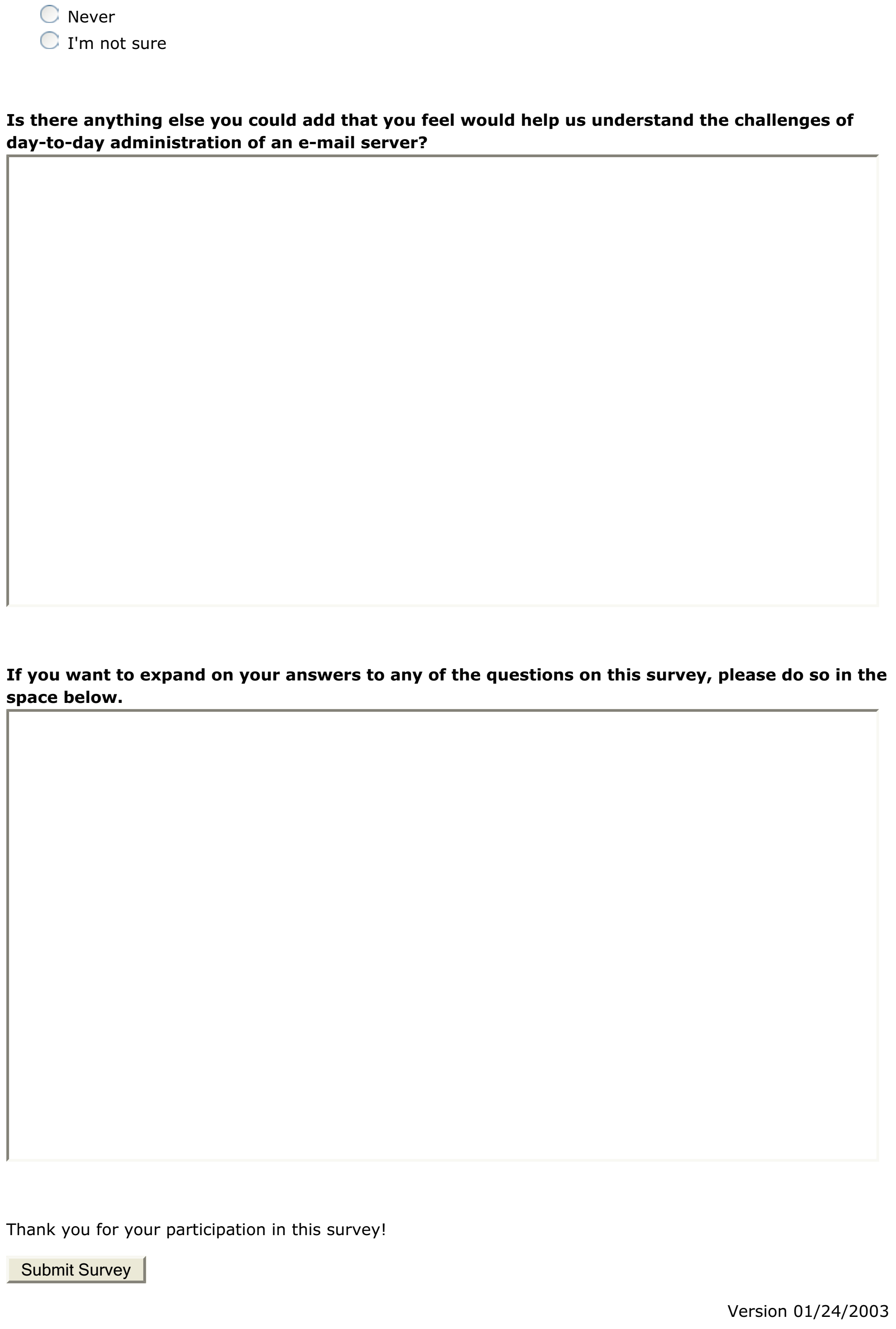
Were those incidents a result of administrative intervention or due to failures of hardware, software, or network components (if applicable)?

Describe any e-mail administration tasks you performed in the last 3 months that were particularly challenging (if applicable; an example might be the task of installing an e-mail SPAM filter). If the execution of any of these tasks resulted in lost e-mail or service unavailability, please explain.

How long after a challenging task are you typically confident that the task or fix is complete and working correctly?

○ Immediately
○ A few minutes
○ A few hours
○ A few days
○ A few weeks
○ More than a month

Figure A-1: Survey of e-mail administrators. Page 6 of 7.



○ Never
○ I'm not sure

Is there anything else you could add that you feel would help us understand the challenges of day-to-day administration of an e-mail server?

If you want to expand on your answers to any of the questions on this survey, please do so in the space below.

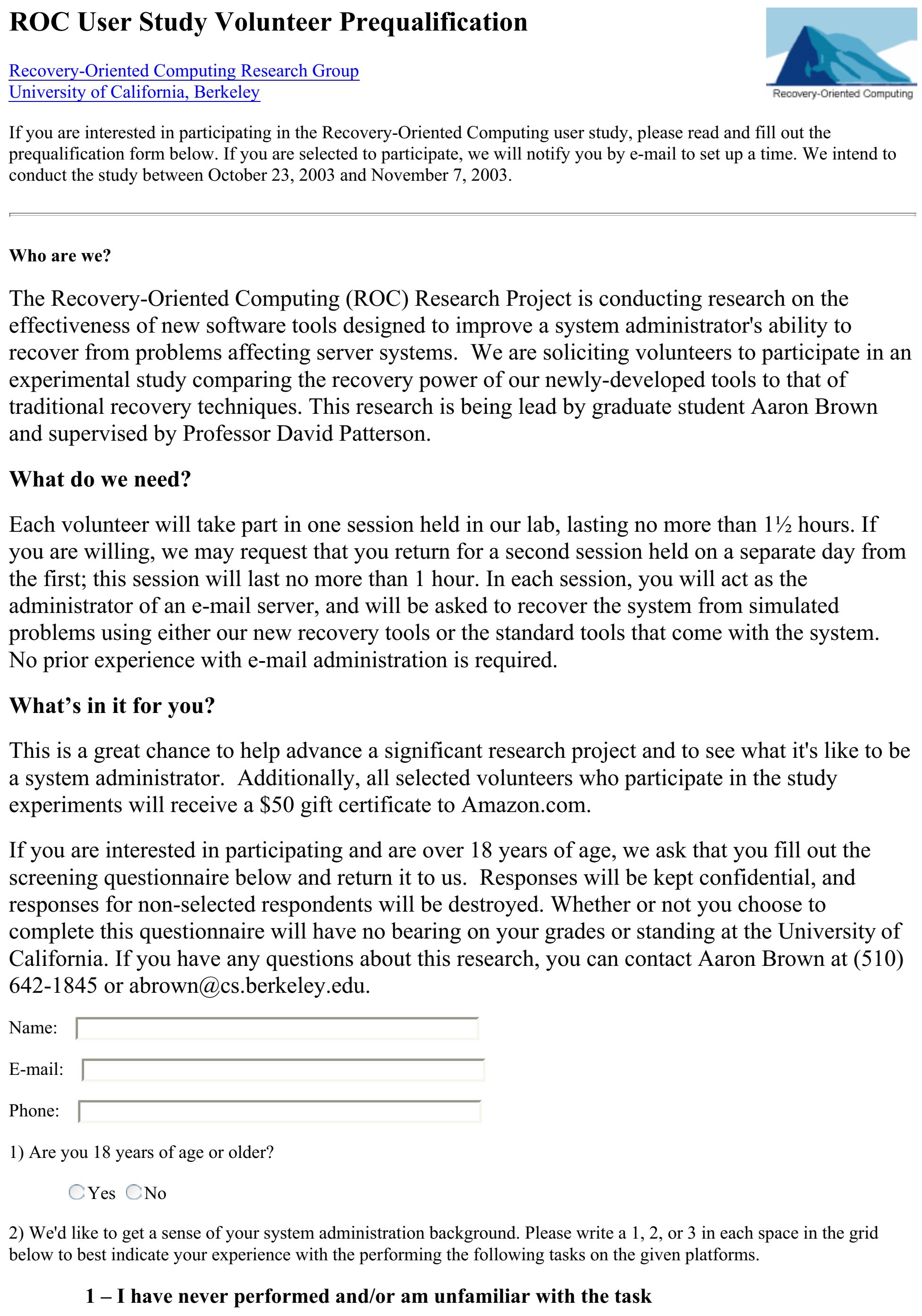
Thank you for your participation in this survey!

Version 01/24/2003

Figure A-1: Survey of e-mail administrators. Page 7 of 7.

A.2 Participant Screening Form

The following pages reproduce the online survey form used to collect data from practicing e-mail system administrators.



ROC User Study Volunteer Prequalification

Recovery-Oriented Computing Research Group
University of California, Berkeley

If you are interested in participating in the Recovery-Oriented Computing user study, please read and fill out the prequalification form below. If you are selected to participate, we will notify you by e-mail to set up a time. We intend to conduct the study between October 23, 2003 and November 7, 2003.

Who are we?

The Recovery-Oriented Computing (ROC) Research Project is conducting research on the effectiveness of new software tools designed to improve a system administrator's ability to recover from problems affecting server systems. We are soliciting volunteers to participate in an experimental study comparing the recovery power of our newly-developed tools to that of traditional recovery techniques. This research is being lead by graduate student Aaron Brown and supervised by Professor David Patterson.

What do we need?

Each volunteer will take part in one session held in our lab, lasting no more than 1½ hours. If you are willing, we may request that you return for a second session held on a separate day from the first; this session will last no more than 1 hour. In each session, you will act as the administrator of an e-mail server, and will be asked to recover the system from simulated problems using either our new recovery tools or the standard tools that come with the system. No prior experience with e-mail administration is required.

What's in it for you?

This is a great chance to help advance a significant research project and to see what it's like to be a system administrator. Additionally, all selected volunteers who participate in the study experiments will receive a $50 gift certificate to Amazon.com.

If you are interested in participating and are over 18 years of age, we ask that you fill out the screening questionnaire below and return it to us. Responses will be kept confidential, and responses for non-selected respondents will be destroyed. Whether or not you choose to complete this questionnaire will have no bearing on your grades or standing at the University of California. If you have any questions about this research, you can contact Aaron Brown at (510) 642-1845 or abrown@cs.berkeley.edu.

Name:

E-mail:

Phone:

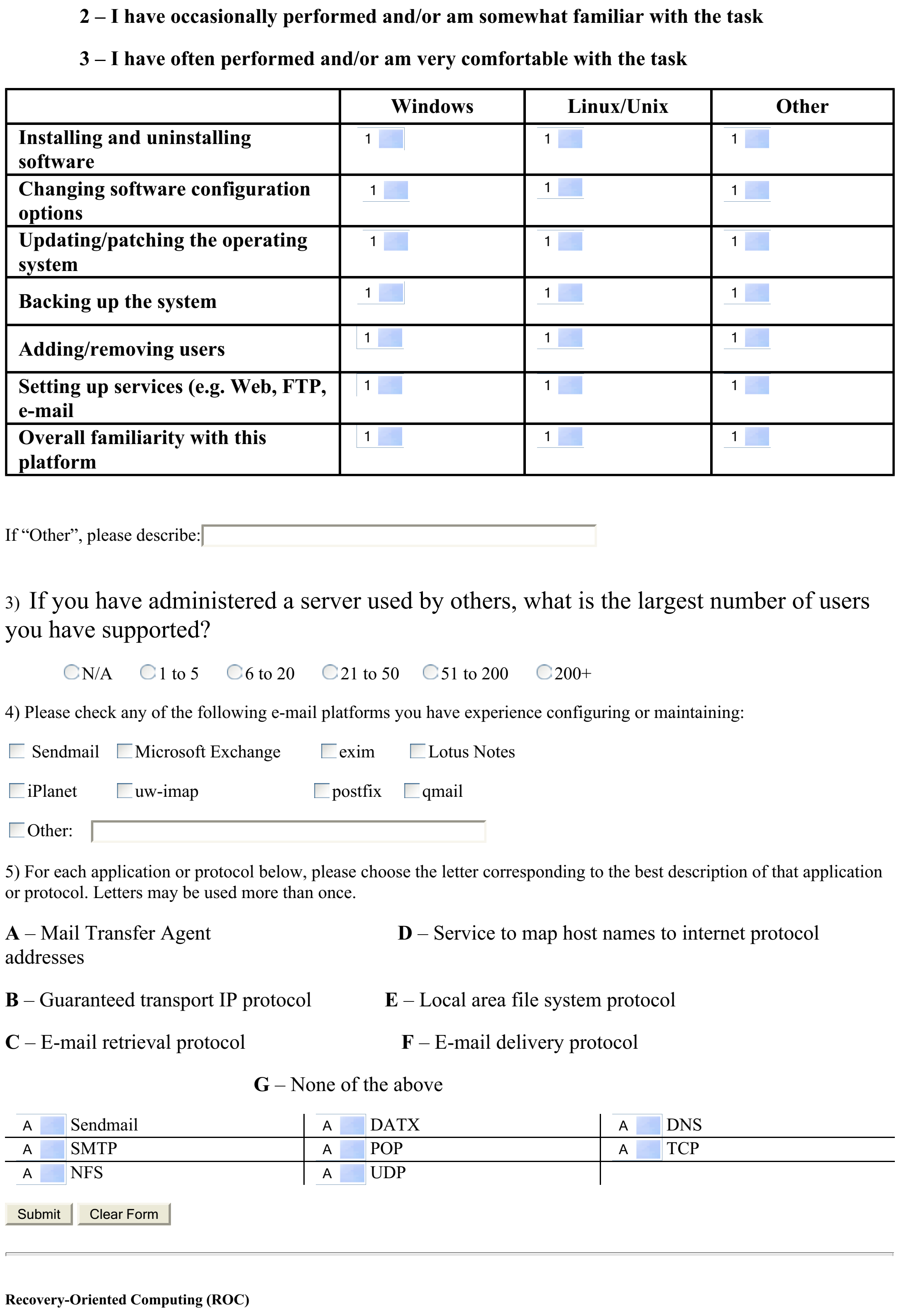
1) Are you 18 years of age or older?

○Yes ○No

2) We'd like to get a sense of your system administration background. Please write a 1, 2, or 3 in each space in the grid below to best indicate your experience with the performing the following tasks on the given platforms.

1 – I have never performed and/or am unfamiliar with the task

Figure A-2: Screening questionnaire. Page 1 of 2.



2 – I have occasionally performed and/or am somewhat familiar with the task

3 – I have often performed and/or am very comfortable with the task

	Windows	**Linux/Unix**	**Other**
Installing and uninstalling software	1	1	1
Changing software configuration options	1	1	1
Updating/patching the operating system	1	1	1
Backing up the system	1	1	1
Adding/removing users	1	1	1
Setting up services (e.g. Web, FTP, e-mail	1	1	1
Overall familiarity with this platform	1	1	1

If "Other", please describe:

3) If you have administered a server used by others, what is the largest number of users you have supported?

○N/A ○1 to 5 ○6 to 20 ○21 to 50 ○51 to 200 ○200+

4) Please check any of the following e-mail platforms you have experience configuring or maintaining:

□ Sendmail □Microsoft Exchange □exim □Lotus Notes

□iPlanet □uw-imap □postfix □qmail

□Other:

5) For each application or protocol below, please choose the letter corresponding to the best description of that application or protocol. Letters may be used more than once.

A – Mail Transfer Agent

D – Service to map host names to internet protocol addresses

B – Guaranteed transport IP protocol

E – Local area file system protocol

C – E-mail retrieval protocol

F – E-mail delivery protocol

G – None of the above

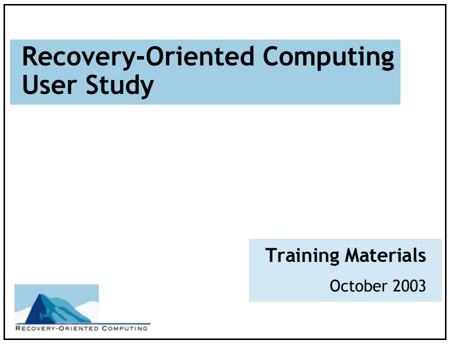
A	Sendmail	A	DATX	A	DNS
A	SMTP	A	POP	A	TCP
A	NFS	A	UDP		

Recovery-Oriented Computing (ROC)

Figure A-2: Screening questionnaire. Page 2 of 2.

A.3 Training Materials

The following slides were given to participants during the training period that constituted the first phase of each benchmark experiment.

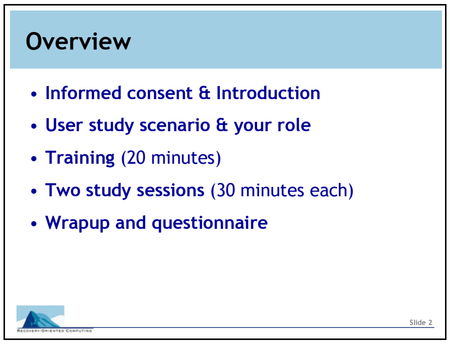


Recovery-Oriented Computing User Study

Training Materials

October 2003

Recovery-Oriented Computing

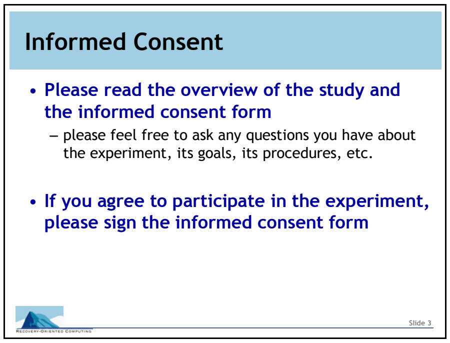


Overview

- **Informed consent & Introduction**
- **User study scenario & your role**
- **Training** (20 minutes)
- **Two study sessions** (30 minutes each)
- **Wrapup and questionnaire**

Recovery-Oriented Computing

Slide 2

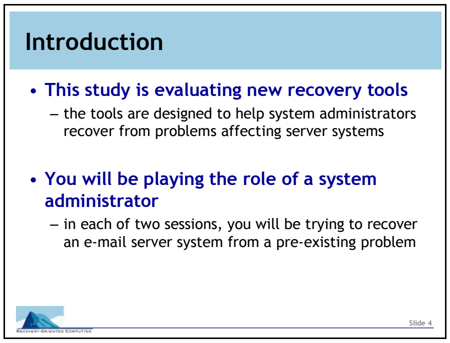


Informed Consent

- **Please read the overview of the study and the informed consent form**
  - please feel free to ask any questions you have about the experiment, its goals, its procedures, etc.
- **If you agree to participate in the experiment, please sign the informed consent form**

Recovery-Oriented Computing

Slide 3

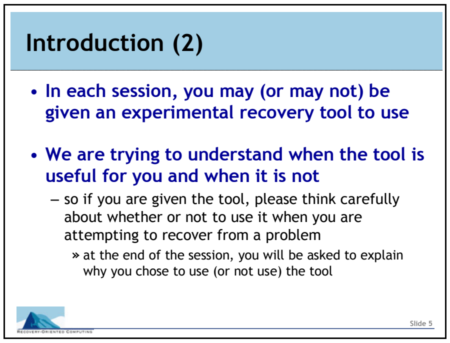


Introduction

- **This study is evaluating new recovery tools**
  - the tools are designed to help system administrators recover from problems affecting server systems
- **You will be playing the role of a system administrator**
  - in each of two sessions, you will be trying to recover an e-mail server system from a pre-existing problem

Recovery-Oriented Computing

Slide 4

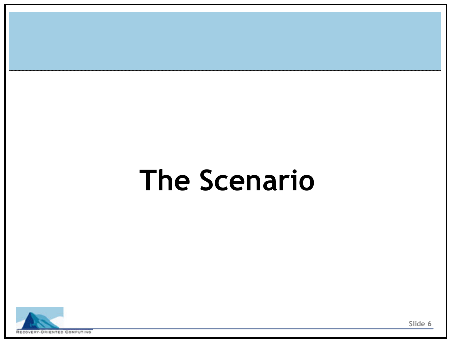


Introduction (2)

- **In each session, you may (or may not) be given an experimental recovery tool to use**
- **We are trying to understand when the tool is useful for you and when it is not**
  - so if you are given the tool, please think carefully about whether or not to use it when you are attempting to recover from a problem
    - » at the end of the session, you will be asked to explain why you chose to use (or not use) the tool

Recovery-Oriented Computing

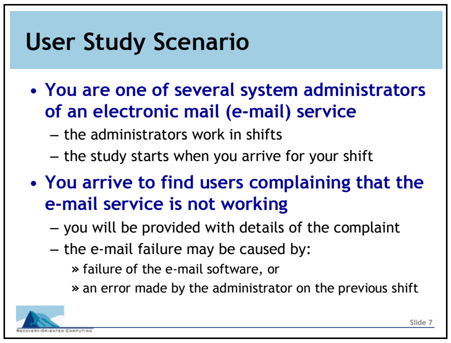
Slide 5



The Scenario

Recovery-Oriented Computing

Slide 6

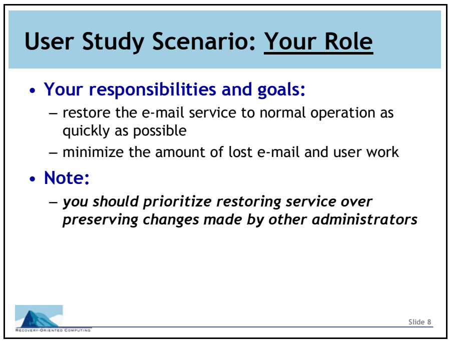


User Study Scenario

- **You are one of several system administrators of an electronic mail (e-mail) service**
  - the administrators work in shifts
  - the study starts when you arrive for your shift
- **You arrive to find users complaining that the e-mail service is not working**
  - you will be provided with details of the complaint
  - the e-mail failure may be caused by:
    - » failure of the e-mail software, or
    - » an error made by the administrator on the previous shift

Recovery-Oriented Computing

Slide 7

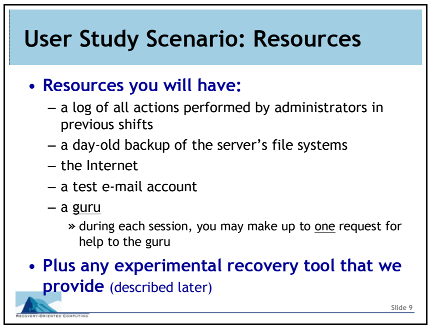


User Study Scenario: Your Role

- **Your responsibilities and goals:**
  - restore the e-mail service to normal operation as quickly as possible
  - minimize the amount of e-mail and user work
- **Note:**
  - ***you should prioritize restoring service over preserving changes made by other administrators***

Recovery-Oriented Computing

Slide 8

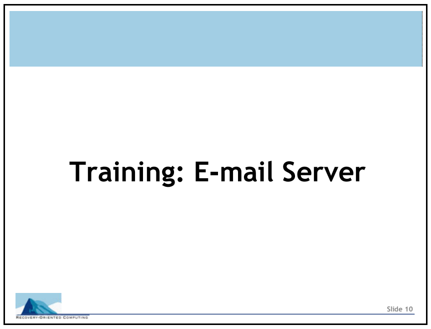


User Study Scenario: Resources

- Resources you will have:
  - a log of all actions performed by administrators in previous shifts
  - a day-old backup of the server's file systems
  - the Internet
  - a test e-mail account
  - a guru
    - » during each session, you may make up to one request for help to the guru
- Plus any experimental recovery tool that we provide (described later)

Recovery-Oriented Computing

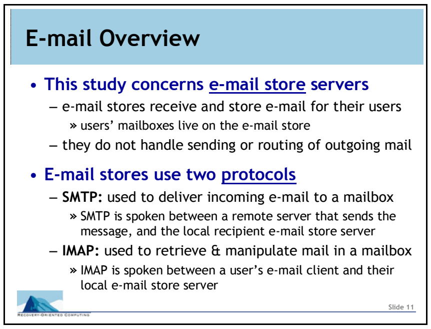
Slide 9



Training: E-mail Server

Recovery-Oriented Computing

Slide 10

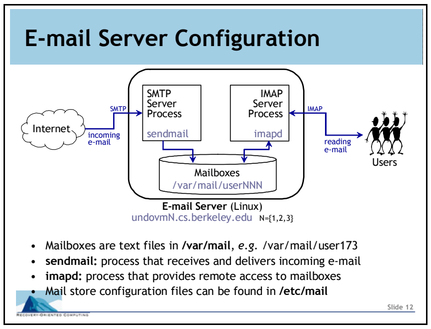


E-mail Overview

- This study concerns e-mail store servers
  - e-mail stores receive and store e-mail for their users
    - » users' mailboxes live on the e-mail store
  - they do not handle sending or routing of outgoing mail
- E-mail stores use two protocols
  - **SMTP:** used to deliver incoming e-mail to a mailbox
    - » SMTP is spoken between a remote server that sends the message, and the local recipient e-mail store server
  - **IMAP:** used to retrieve & manipulate mail in a mailbox
    - » IMAP is spoken between a user's e-mail client and their local e-mail store server

Recovery-Oriented Computing

Slide 11



E-mail Server Configuration

Internet

SMTP

incoming e-mail

SMTP Server Process

sendmail

IMAP Server Process

imapd

IMAP

reading e-mail

Users

Mailboxes

/var/mail/userNNN

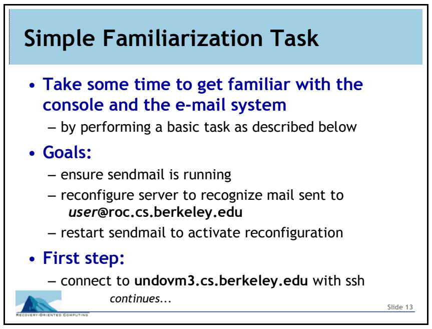
E-mail Server (Linux)

undovmN.cs.berkeley.edu N={1,2,3}

- Mailboxes are text files in **/var/mail**, *e.g.* /var/mail/user173
- **sendmail:** process that receives and delivers incoming e-mail
- **imapd:** process that provides remote access to mailboxes
- Mail store configuration files can be found in **/etc/mail**

Recovery-Oriented Computing

Slide 12



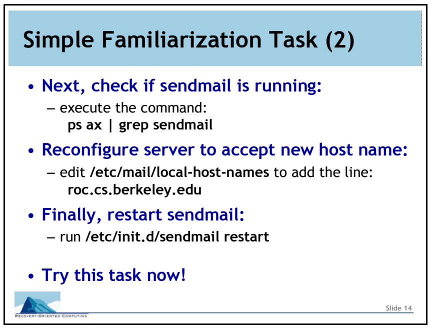
Simple Familiarization Task

- Take some time to get familiar with the console and the e-mail system
  - by performing a basic task as described below
- Goals:
  - ensure sendmail is running
  - reconfigure server to recognize mail sent to ***user*@roc.cs.berkeley.edu**
  - restart sendmail to activate reconfiguration
- First step:
  - connect to **undovm3.cs.berkeley.edu** with ssh

continues...

Recovery-Oriented Computing

Slide 13



Simple Familiarization Task (2)

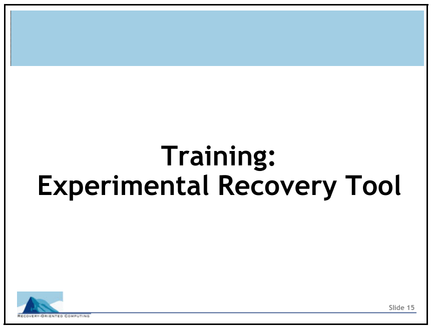
- Next, check if sendmail is running:
  - execute the command:

    ps ax | grep sendmail
- Reconfigure server to accept new host name:
  - edit **/etc/mail/local-host-names** to add the line:

    roc.cs.berkeley.edu
- Finally, restart sendmail:
  - run **/etc/init.d/sendmail restart**
- Try this task now!

Recovery-Oriented Computing

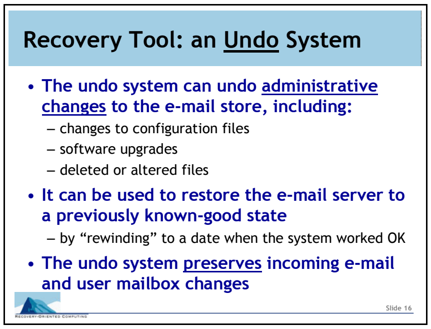
Slide 14



Training: Experimental Recovery Tool

Recovery-Oriented Computing

Slide 15

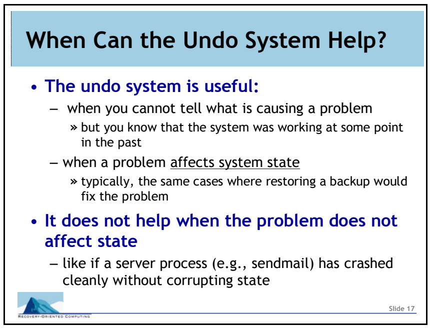


Recovery Tool: an Undo System

- The undo system can undo administrative changes to the e-mail store, including:
  - changes to configuration files
  - software upgrades
  - deleted or altered files
- It can be used to restore the e-mail server to a previously known-good state
  - by "rewinding" to a date when the system worked OK
- The undo system preserves incoming e-mail and user mailbox changes

Recovery-Oriented Computing

Slide 16

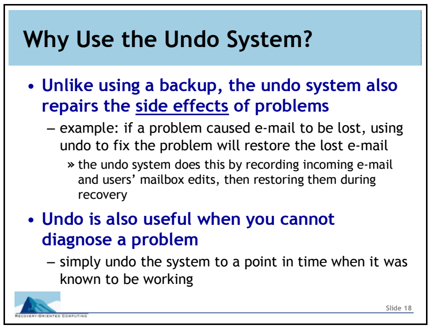


When Can the Undo System Help?

- The undo system is useful:
  - when you cannot tell what is causing a problem
    - » but you know that the system was working at some point in the past
  - when a problem affects system state
    - » typically, the same cases where restoring a backup would fix the problem
- It does not help when the problem does not affect state
  - like if a server process (e.g., sendmail) has crashed cleanly without corrupting state

Recovery-Oriented Computing

Slide 17

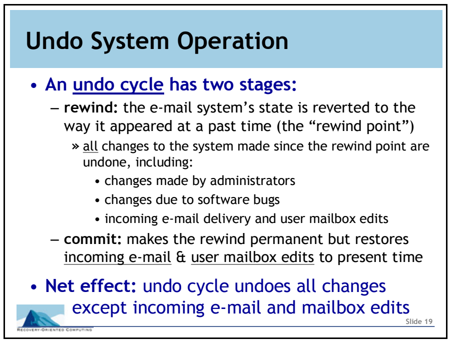


Why Use the Undo System?

- Unlike using a backup, the undo system also repairs the side effects of problems
  - example: if a problem caused e-mail to be lost, using undo to fix the problem will restore the lost e-mail
    - » the undo system does this by recording incoming e-mail and users' mailbox edits, then restoring them during recovery
- Undo is also useful when you cannot diagnose a problem
  - simply undo the system to a point in time when it was known to be working

Recovery-Oriented Computing

Slide 18



Undo System Operation

- **An undo cycle has two stages:**
  - **rewind:** the e-mail system's state is reverted to the way it appeared at a past time (the "rewind point")
    - » all changes to the system made since the rewind point are undone, including:
      - changes made by administrators
      - changes due to software bugs
      - incoming e-mail delivery and user mailbox edits
  - **commit:** makes the rewind permanent but restores incoming e-mail & user mailbox edits to present time
- **Net effect:** undo cycle undoes all changes except incoming e-mail and mailbox edits

Slide 19

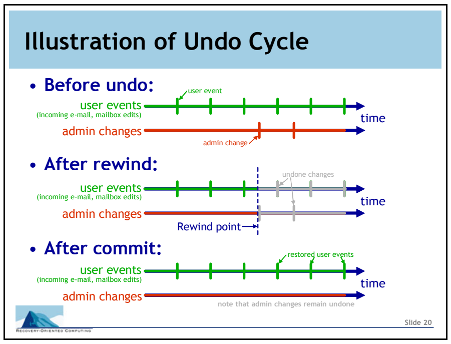
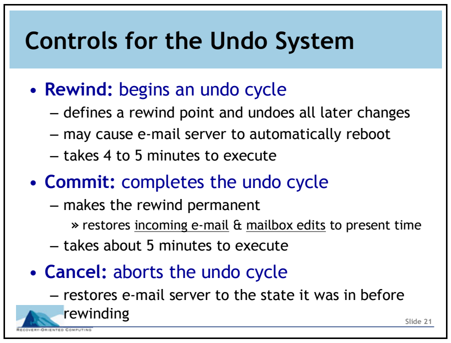


Illustration of Undo Cycle

- **Before undo:**
- **After rewind:**
- **After commit:**

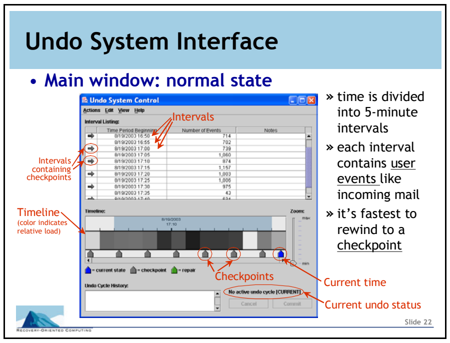
Slide 20



Controls for the Undo System

- **Rewind:** begins an undo cycle
  - defines a rewind point and undoes all later changes
  - may cause e-mail server to automatically reboot
  - takes 4 to 5 minutes to execute
- **Commit:** completes the undo cycle
  - makes the rewind permanent
    - » restores incoming e-mail & mailbox edits to present time
  - takes about 5 minutes to execute
- **Cancel:** aborts the undo cycle
  - restores e-mail server to the state it was in before rewinding

Slide 21



Undo System Interface

- **Main window: normal state**

Undo System Control

Interval Listing:

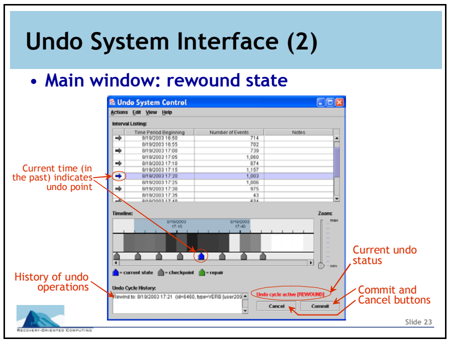
	Time Period Beginning	Number of Events	Notes
➡	8/19/2003 16:50	714	
	8/19/2003 16:55	702	
➡	8/19/2003 17:00	739	
	8/19/2003 17:05	1,060	
➡	8/19/2003 17:10	874	
	8/19/2003 17:15	1,157	
➡	8/19/2003 17:20	1,003	
	8/19/2003 17:25	1,006	
➡	8/19/2003 17:30	975	
	8/19/2003 17:35	43	
➡	8/19/2003 17:40	634	

Undo Cycle History:

No active undo cycle [CURRENT]

- » time is divided into 5-minute intervals
- » each interval contains user events like incoming mail
- » it's fastest to rewind to a checkpoint

Slide 22



Undo System Interface (2)

- **Main window: rewound state**

Undo System Control

Interval Listing:

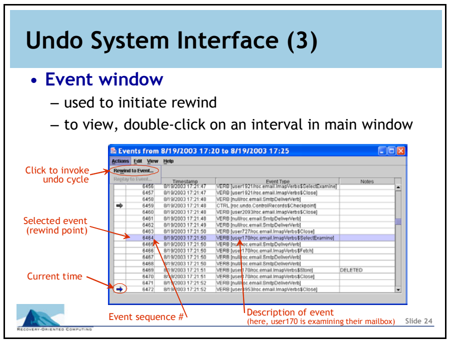
	Time Period Beginning	Number of Events	Notes
➡	8/19/2003 16:50	714	
	8/19/2003 16:55	702	
➡	8/19/2003 17:00	739	
	8/19/2003 17:05	1,060	
➡	8/19/2003 17:10	874	
	8/19/2003 17:15	1,157	
➡	8/19/2003 17:20	1,003	
	8/19/2003 17:25	1,006	
➡	8/19/2003 17:30	975	
	8/19/2003 17:35	43	
➡	8/19/2003 17:40	634	

Undo Cycle History:

Rewind to: 8/19/2003 17:21 (id=6460, type=VERB [user209

Undo cycle active [REWOUND]

Slide 23



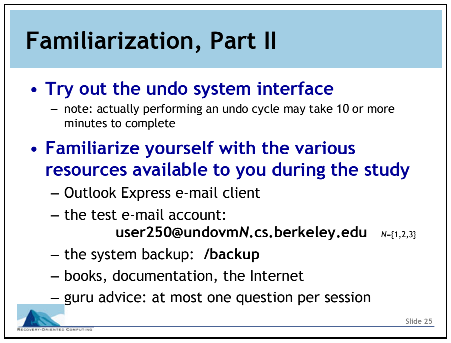
Undo System Interface (3)

- **Event window**
  - used to initiate rewind
  - to view, double-click on an interval in main window

Events from 8/19/2003 17:20 to 8/19/2003 17:25

		Timestamp	Event Type	Notes
	6456	8/19/2003 17:21:47	VERB [user1921/roc.email.ImapVerbs$SelectExamine]	
	6457	8/19/2003 17:21:47	VERB [user1921/roc.email.ImapVerbs$Close]	
	6458	8/19/2003 17:21:48	VERB [null/roc.email.SmtpDeliverVerb]	
➡	6459	8/19/2003 17:21:48	CTRL [roc.undo.ControlRecords$Checkpoint]	
	6460	8/19/2003 17:21:48	VERB [user2093/roc.email.ImapVerbs$Close]	
	6461	8/19/2003 17:21:48	VERB [null/roc.email.SmtpDeliverVerb]	
	6462	8/19/2003 17:21:49	VERB [null/roc.email.SmtpDeliverVerb]	
	6463	8/19/2003 17:21:50	VERB [user727/roc.email.ImapVerbs$Close]	
	6464	8/19/2003 17:21:50	VERB [user170/roc.email.ImapVerbs$SelectExamine]	
	6465	8/19/2003 17:21:50	VERB [null/roc.email.SmtpDeliverVerb]	
	6466	8/19/2003 17:21:50	VERB [user170/roc.email.ImapVerbs$Fetch]	
	6467	8/19/2003 17:21:50	VERB [null/roc.email.SmtpDeliverVerb]	
	6468	8/19/2003 17:21:50	VERB [null/roc.email.SmtpDeliverVerb]	
	6469	8/19/2003 17:21:51	VERB [user170/roc.email.ImapVerbs$Store]	DELETED
	6470	8/19/2003 17:21:51	VERB [user170/roc.email.ImapVerbs$Close]	
	6471	8/19/2003 17:21:52	VERB [null/roc.email.SmtpDeliverVerb]	
➡	6472	8/19/2003 17:21:52	VERB [user1953/roc.email.ImapVerbs$Close]	

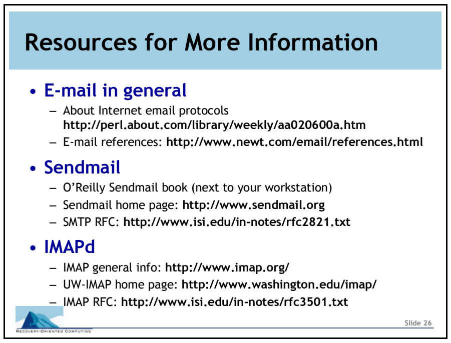
Slide 24



Familiarization, Part II

- **Try out the undo system interface**
  - note: actually performing an undo cycle may take 10 or more minutes to complete
- **Familiarize yourself with the various resources available to you during the study**
  - Outlook Express e-mail client
  - the test e-mail account: **user250@undovmN.cs.berkeley.edu** *N*={1,2,3}
  - the system backup: ***/backup***
  - books, documentation, the Internet
  - guru advice: at most one question per session

Slide 25



Resources for More Information

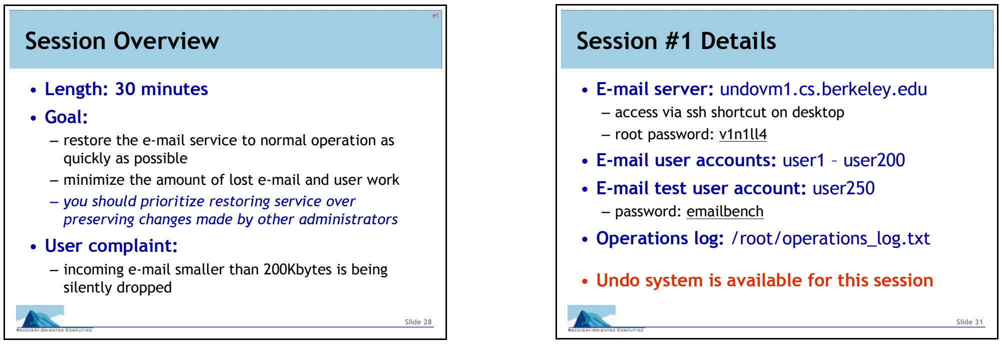
- **E-mail in general**
  - About Internet email protocols **http://perl.about.com/library/weekly/aa020600a.htm**
  - E-mail references: **http://www.newt.com/email/references.html**
- **Sendmail**
  - O'Reilly Sendmail book (next to your workstation)
  - Sendmail home page: **http://www.sendmail.org**
  - SMTP RFC: **http://www.isi.edu/in-notes/rfc2821.txt**
- **IMAPd**
  - IMAP general info: **http://www.imap.org/**
  - UW-IMAP home page: **http://www.washington.edu/imap/**
  - IMAP RFC: **http://www.isi.edu/in-notes/rfc3501.txt**

Slide 26

Figure A-3: Training materials.

A.4 Scenario Descriptions

The following slides were used to instruct each participant on the failure scenario they had to recover from in each benchmark trial. The slides break down into sets of two, one set for each pairing of failure scenario and trial number.



e1

Session Overview

- **Length: 30 minutes**
- **Goal:**
  - – restore the e-mail service to normal operation as quickly as possible
  - – minimize the amount of lost e-mail and user work
  - *– you should prioritize restoring service over preserving changes made by other administrators*
- **User complaint:**
  - – incoming e-mail smaller than 200Kbytes is being silently dropped

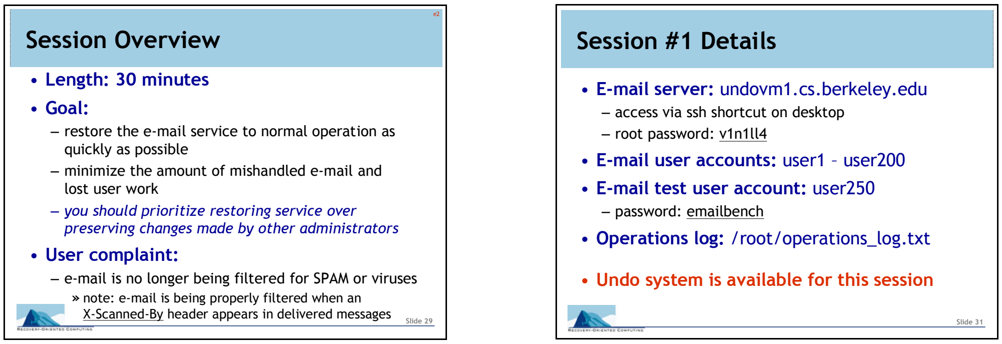
Recovery-Oriented Computing Slide 28

Session #1 Details

- **E-mail server:** undovm1.cs.berkeley.edu
  - – access via ssh shortcut on desktop
  - – root password: v1n1ll4
- **E-mail user accounts:** user1 – user200
- **E-mail test user account:** user250
  - – password: emailbench
- **Operations log:** /root/operations_log.txt
- **Undo system is available for this session**

Recovery-Oriented Computing Slide 31

Figure A-4: Scenario description: trial #1, scenario #1.



e2

Session Overview

- **Length: 30 minutes**
- **Goal:**
  - – restore the e-mail service to normal operation as quickly as possible
  - – minimize the amount of mishandled e-mail and lost user work
  - *– you should prioritize restoring service over preserving changes made by other administrators*
- **User complaint:**
  - – e-mail is no longer being filtered for SPAM or viruses
    - » note: e-mail is being properly filtered when an X-Scanned-By header appears in delivered messages

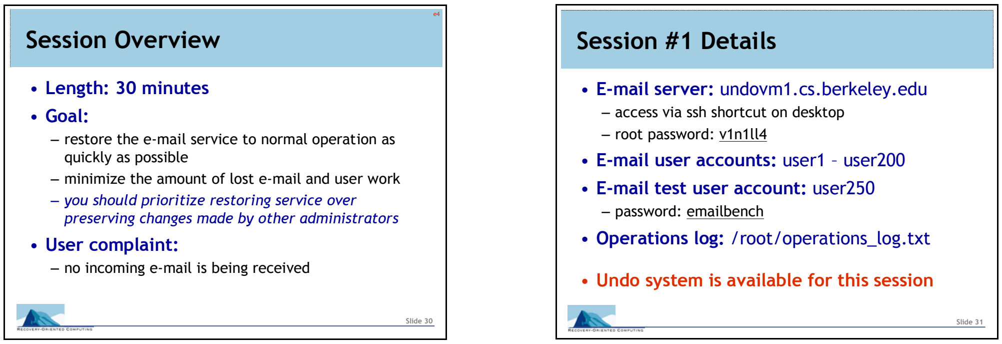
Recovery-Oriented Computing Slide 29

Session #1 Details

- **E-mail server:** undovm1.cs.berkeley.edu
  - – access via ssh shortcut on desktop
  - – root password: v1n1ll4
- **E-mail user accounts:** user1 – user200
- **E-mail test user account:** user250
  - – password: emailbench
- **Operations log:** /root/operations_log.txt
- **Undo system is available for this session**

Recovery-Oriented Computing Slide 31

Figure A-5: Scenario description: trial #1, scenario #2.



e4

Session Overview

- **Length: 30 minutes**
- **Goal:**
  - – restore the e-mail service to normal operation as quickly as possible
  - – minimize the amount of lost e-mail and user work
  - *– you should prioritize restoring service over preserving changes made by other administrators*
- **User complaint:**
  - – no incoming e-mail is being received

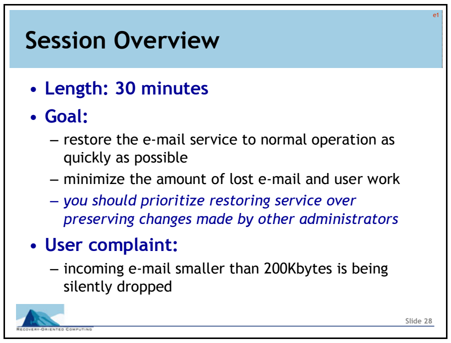
Recovery-Oriented Computing Slide 30

Session #1 Details

- **E-mail server:** undovm1.cs.berkeley.edu
  - – access via ssh shortcut on desktop
  - – root password: v1n1ll4
- **E-mail user accounts:** user1 – user200
- **E-mail test user account:** user250
  - – password: emailbench
- **Operations log:** /root/operations_log.txt
- **Undo system is available for this session**

Recovery-Oriented Computing Slide 31

Figure A-6: Scenario description: trial #1, scenario #3.

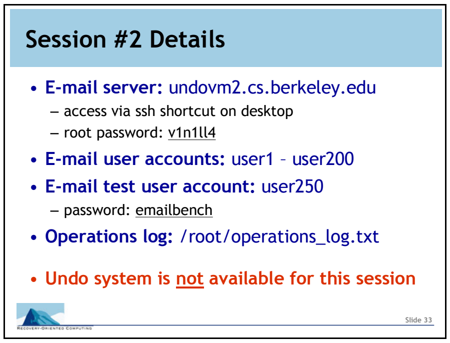


e1

Session Overview

- **Length: 30 minutes**
- **Goal:**
  - restore the e-mail service to normal operation as quickly as possible
  - minimize the amount of lost e-mail and user work
  - *you should prioritize restoring service over preserving changes made by other administrators*
- **User complaint:**
  - incoming e-mail smaller than 200Kbytes is being silently dropped

Slide 28

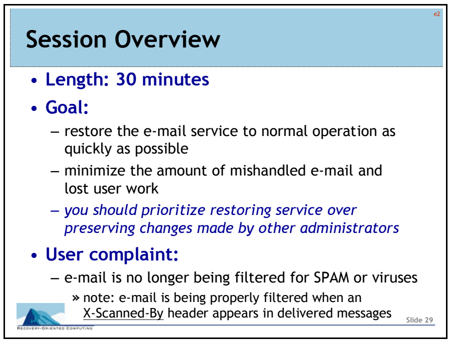


Session #2 Details

- **E-mail server:** undovm2.cs.berkeley.edu
  - access via ssh shortcut on desktop
  - root password: v1n1ll4
- **E-mail user accounts:** user1 - user200
- **E-mail test user account:** user250
  - password: emailbench
- **Operations log:** /root/operations_log.txt
- Undo system is not available for this session

Slide 33

Figure A-7: Scenario description: trial #2, scenario #1.

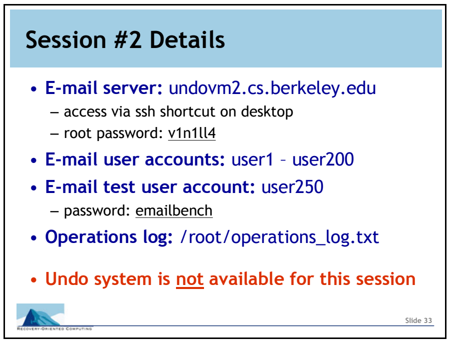


e2

Session Overview

- **Length: 30 minutes**
- **Goal:**
  - restore the e-mail service to normal operation as quickly as possible
  - minimize the amount of mishandled e-mail and lost user work
  - *you should prioritize restoring service over preserving changes made by other administrators*
- **User complaint:**
  - e-mail is no longer being filtered for SPAM or viruses
    - » note: e-mail is being properly filtered when an X-Scanned-By header appears in delivered messages

Slide 29

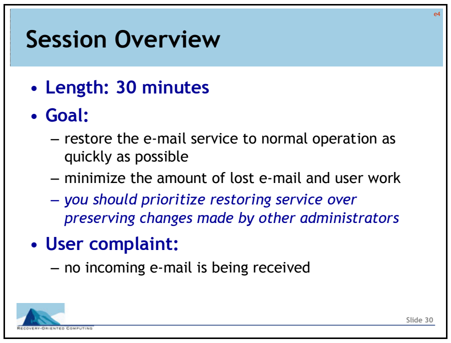


Session #2 Details

- **E-mail server:** undovm2.cs.berkeley.edu
  - access via ssh shortcut on desktop
  - root password: v1n1ll4
- **E-mail user accounts:** user1 - user200
- **E-mail test user account:** user250
  - password: emailbench
- **Operations log:** /root/operations_log.txt
- Undo system is not available for this session

Slide 33

Figure A-8: Scenario description: trial #2, scenario #2.

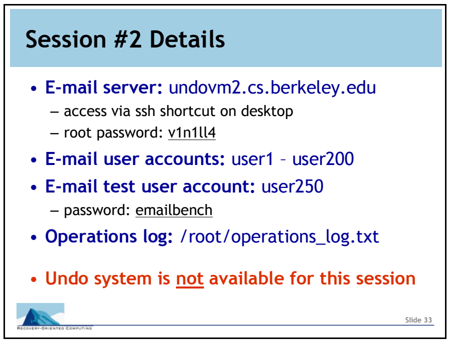


e4

Session Overview

- **Length: 30 minutes**
- **Goal:**
  - restore the e-mail service to normal operation as quickly as possible
  - minimize the amount of lost e-mail and user work
  - *you should prioritize restoring service over preserving changes made by other administrators*
- **User complaint:**
  - no incoming e-mail is being received

Slide 30



Session #2 Details

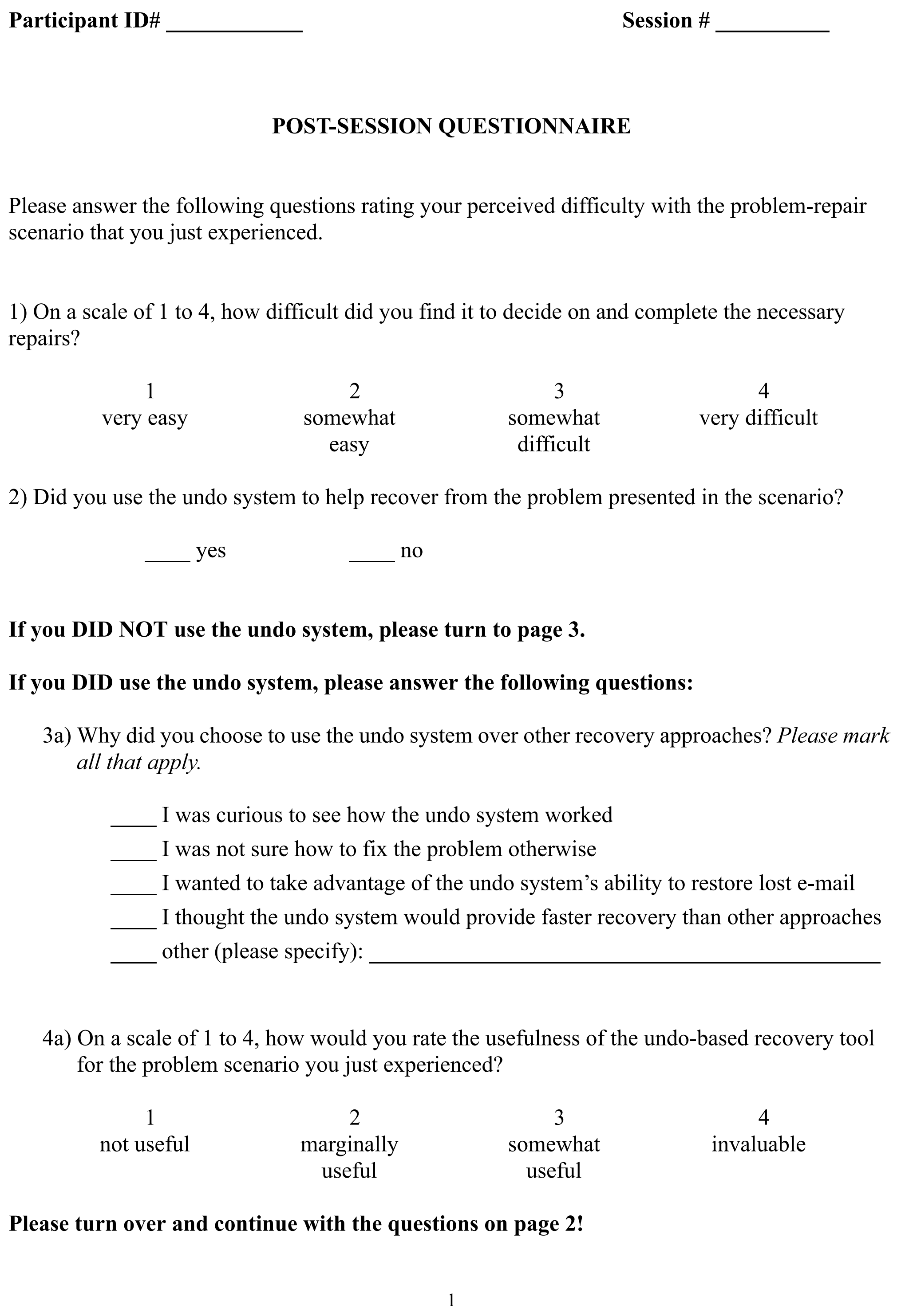
- **E-mail server:** undovm2.cs.berkeley.edu
  - access via ssh shortcut on desktop
  - root password: v1n1ll4
- **E-mail user accounts:** user1 - user200
- **E-mail test user account:** user250
  - password: emailbench
- **Operations log:** /root/operations_log.txt
- Undo system is not available for this session

Slide 33

Figure A-9: Scenario description: trial #2, scenario #3.

A.5 Questionnaires

After each benchmark trial, participants were given a questionnaire to probe their experiences during the trial. The following pages show the questionnaire used after the first trial, in which the undo facility was available:



Participant ID# ____________ **Session # __________**

POST-SESSION QUESTIONNAIRE

Please answer the following questions rating your perceived difficulty with the problem-repair scenario that you just experienced.

1) On a scale of 1 to 4, how difficult did you find it to decide on and complete the necessary repairs?

1	2	3	4
very easy	somewhat easy	somewhat difficult	very difficult

2) Did you use the undo system to help recover from the problem presented in the scenario?

____ yes ____ no

If you DID NOT use the undo system, please turn to page 3.

If you DID use the undo system, please answer the following questions:

3a) Why did you choose to use the undo system over other recovery approaches? *Please mark all that apply.*

____ I was curious to see how the undo system worked
____ I was not sure how to fix the problem otherwise
____ I wanted to take advantage of the undo system's ability to restore lost e-mail
____ I thought the undo system would provide faster recovery than other approaches
____ other (please specify): ____

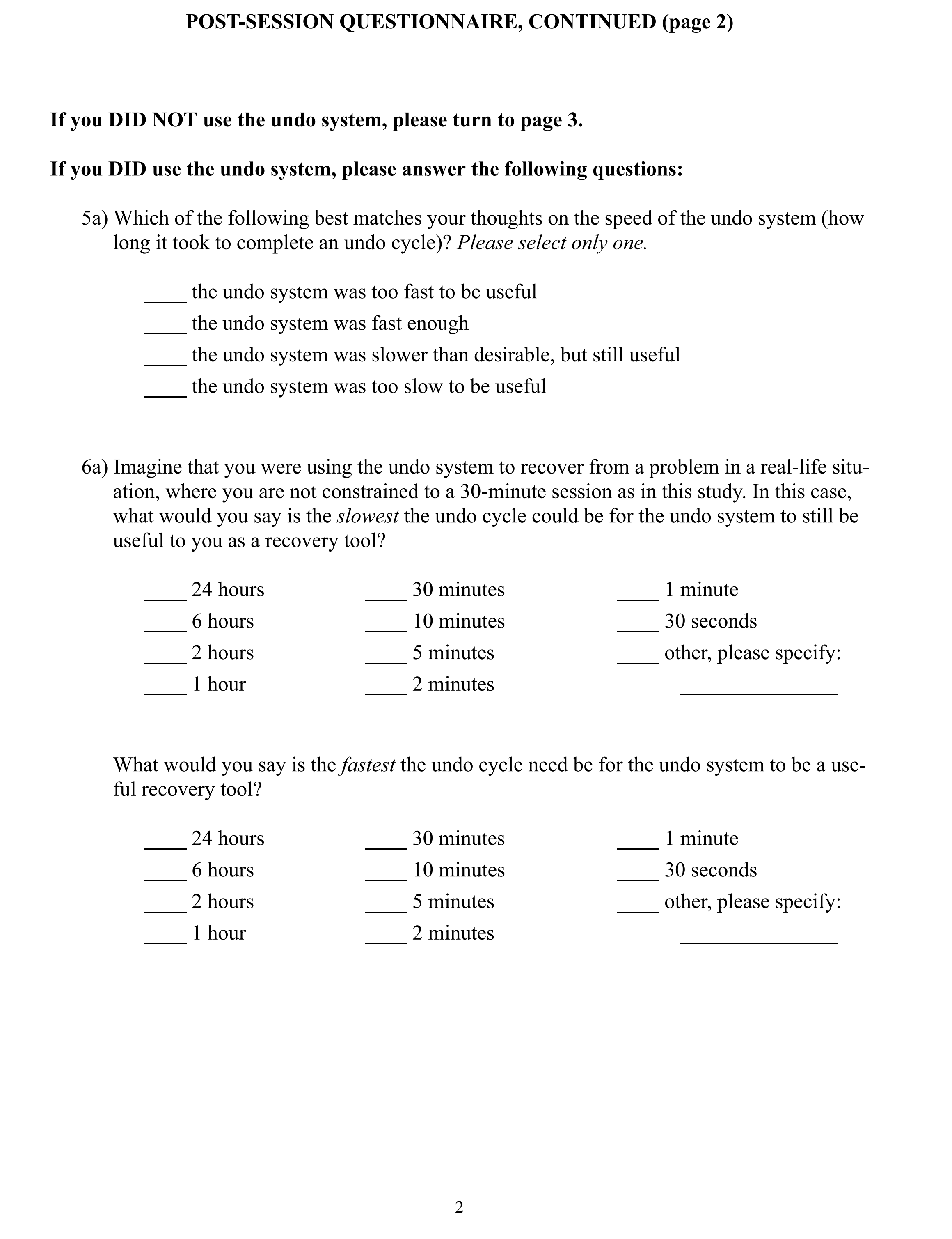
4a) On a scale of 1 to 4, how would you rate the usefulness of the undo-based recovery tool for the problem scenario you just experienced?

1	2	3	4
not useful	marginally useful	somewhat useful	invaluable

Please turn over and continue with the questions on page 2!

1

Figure A-10: Questionnaire used after first benchmark trial. Page 1 of 3.



POST-SESSION QUESTIONNAIRE, CONTINUED (page 2)

If you DID NOT use the undo system, please turn to page 3.

If you DID use the undo system, please answer the following questions:

5a) Which of the following best matches your thoughts on the speed of the undo system (how long it took to complete an undo cycle)? *Please select only one.*

____ the undo system was too fast to be useful
____ the undo system was fast enough
____ the undo system was slower than desirable, but still useful
____ the undo system was too slow to be useful

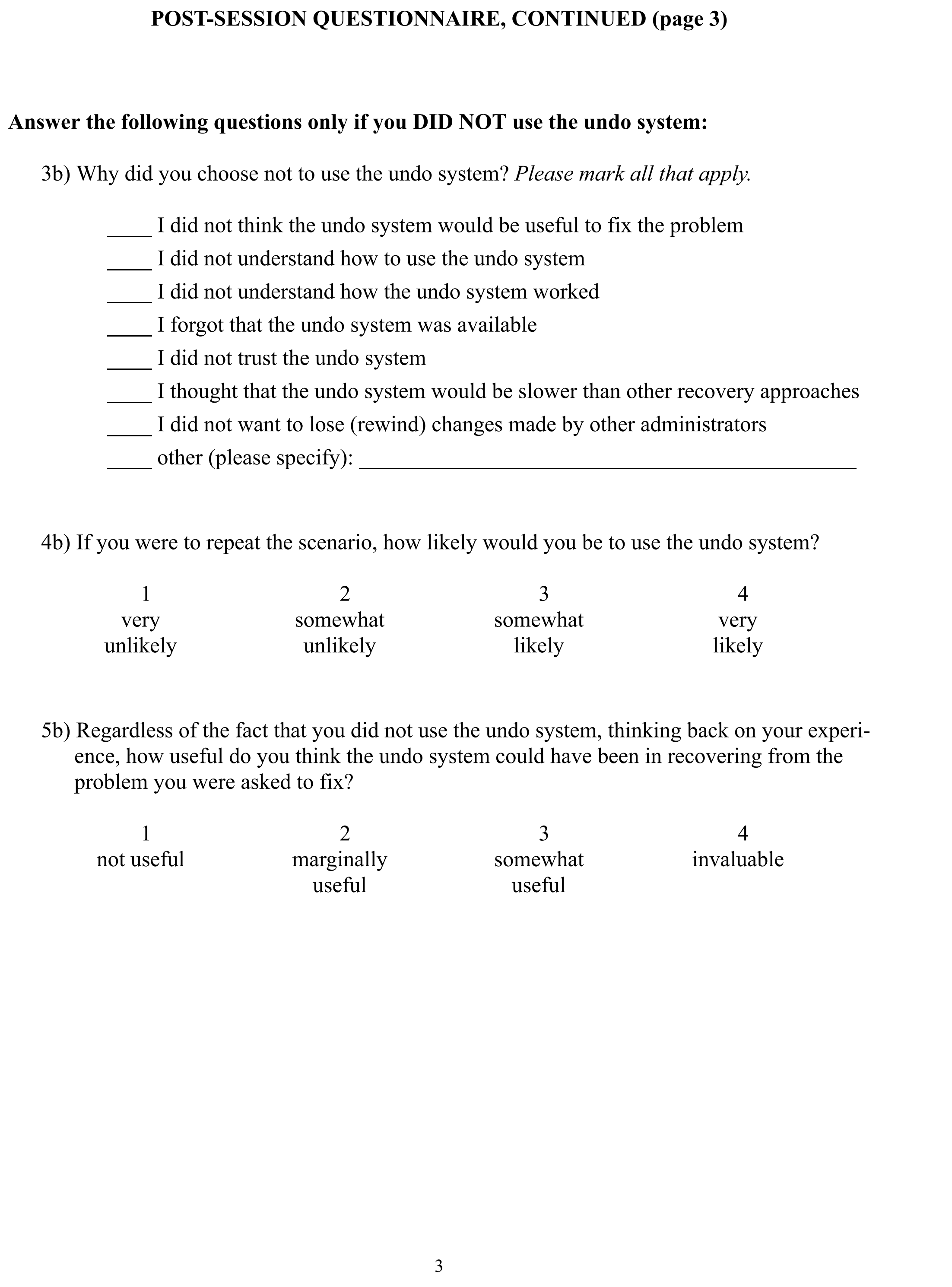
6a) Imagine that you were using the undo system to recover from a problem in a real-life situation, where you are not constrained to a 30-minute session as in this study. In this case, what would you say is the *slowest* the undo cycle could be for the undo system to still be useful to you as a recovery tool?

____ 24 hours
____ 6 hours
____ 2 hours
____ 1 hour
____ 30 minutes
____ 10 minutes
____ 5 minutes
____ 2 minutes
____ 1 minute
____ 30 seconds
____ other, please specify:
____

What would you say is the *fastest* the undo cycle need be for the undo system to be a useful recovery tool?

____ 24 hours
____ 6 hours
____ 2 hours
____ 1 hour
____ 30 minutes
____ 10 minutes
____ 5 minutes
____ 2 minutes
____ 1 minute
____ 30 seconds
____ other, please specify:
____

Figure A-10: Questionnaire used after first benchmark trial. Page 2 of 3.



POST-SESSION QUESTIONNAIRE, CONTINUED (page 3)

Answer the following questions only if you DID NOT use the undo system:

3b) Why did you choose not to use the undo system? *Please mark all that apply.*

____ I did not think the undo system would be useful to fix the problem
____ I did not understand how to use the undo system
____ I did not understand how the undo system worked
____ I forgot that the undo system was available
____ I did not trust the undo system
____ I thought that the undo system would be slower than other recovery approaches
____ I did not want to lose (rewind) changes made by other administrators
____ other (please specify): ____

4b) If you were to repeat the scenario, how likely would you be to use the undo system?

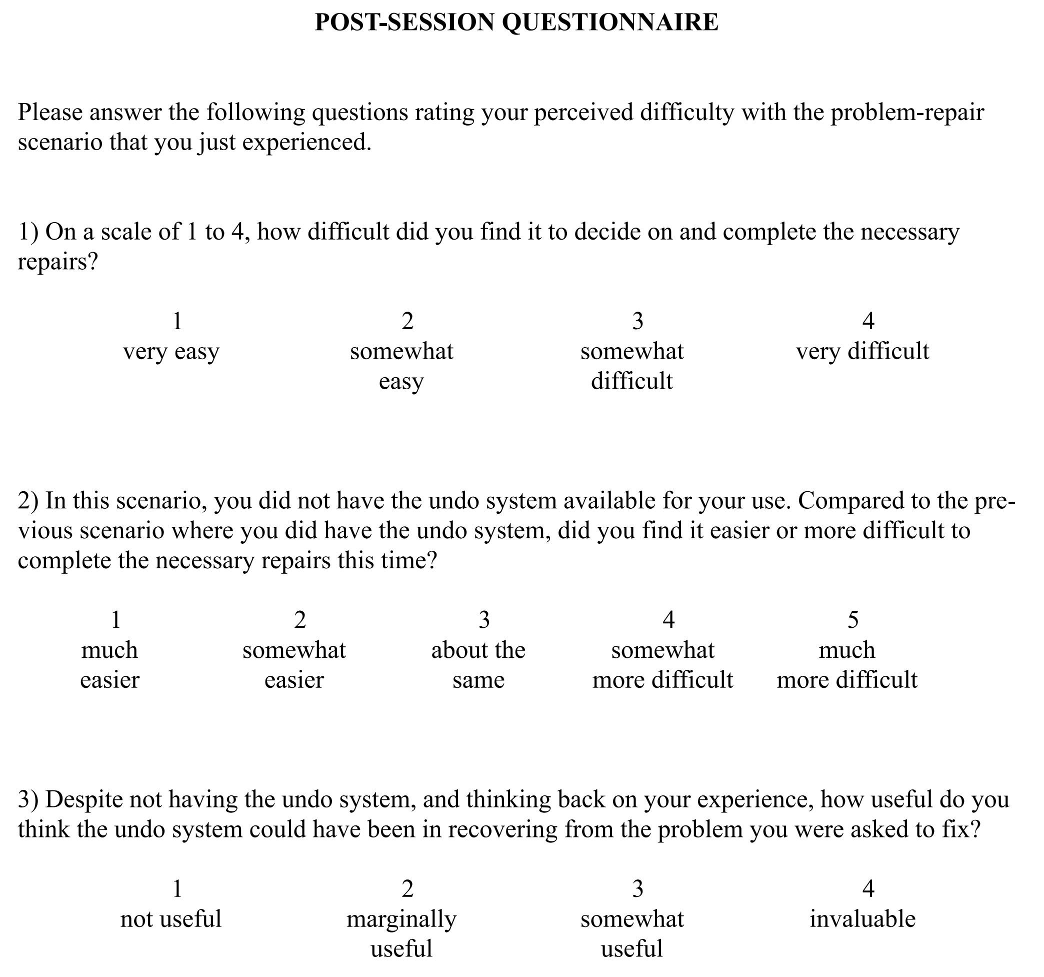
1	2	3	4
very unlikely	somewhat unlikely	somewhat likely	very likely

5b) Regardless of the fact that you did not use the undo system, thinking back on your experience, how useful do you think the undo system could have been in recovering from the problem you were asked to fix?

1	2	3	4
not useful	marginally useful	somewhat useful	invaluable

Figure A-10: Questionnaire used after first benchmark trial. Page 3 of 3.

The following questionnaire was used after the second trial, in which the undo facility was *not* available:



POST-SESSION QUESTIONNAIRE

Please answer the following questions rating your perceived difficulty with the problem-repair scenario that you just experienced.

1) On a scale of 1 to 4, how difficult did you find it to decide on and complete the necessary repairs?

1	2	3	4
very easy	somewhat easy	somewhat difficult	very difficult

2) In this scenario, you did not have the undo system available for your use. Compared to the previous scenario where you did have the undo system, did you find it easier or more difficult to complete the necessary repairs this time?

1	2	3	4	5
much easier	somewhat easier	about the same	somewhat more difficult	much more difficult

3) Despite not having the undo system, and thinking back on your experience, how useful do you think the undo system could have been in recovering from the problem you were asked to fix?

1	2	3	4
not useful	marginally useful	somewhat useful	invaluable

Figure A-11: Questionnaire used after second benchmark trial.

References

[1] G. D. Abowd and A. J. Dix. Giving Undo Attention. *Interacting with Computers*, 4(3):317–342, 1992.

[2] C. Amza, E. Cecchet, A. Chanda et al. Specification and Implementation of Dynamic Web Site Benchmarks. *Proceedings of the 5th IEEE 5th Annual Workshop on Workload Characterization (WWC-5)*. Austin, TX, November 2002.

[3] E. Anderson and D. A. Patterson. A Retrospective on Twelve Years of LISA Proceedings. *Proceedings of the 13th Systems Administration Conference (LISA XIII)*. Seattle, WA, 1999.

[4] J. E. Archer, R. Conway, and F. B. Schneider. User Recovery and Reversal in Interactive Systems. *ACM Transactions on Programming Languages and Systems*, 6(1):1–19, 1984.

[5] A. Avizienis, J.-C. Laprie, and B. Randell. Fundamental Concepts of Dependability. *Proceedings of the Third Information Survivability Workshop (ISW-2000)*. Boston, MA, October 2000.

[6] L. Bainbridge. The ironies of automation. In J. Rasmussen, K. Duncan, and J. Leplat (Eds.), *New Technology and Human Error*. London: Wiley, 1987.

[7] R. Barga and D. Lomet. Phoenix Project: Fault-Tolerant Applications. *SIGMOD Record*, 31(2):94-100, June 2002.

[8] R. Barga, D. Lomet, S. Paparizos et al. Persistent Applications via Automatic Recovery. *Proceedings of the 7th International Database Engineering and Applications Symposium*. Hong Kong, July 2003.

[9] R. Barrett. System Administrators are Users Too. *Stanford University Seminar on People, Computers, and Design*, Stanford, CA, May 30, 2003. Details at http://hci.stanford.edu/cs547/abstracts/02-03/030530-barrett.html. Video available at http://murl.microsoft.com/LectureDetails.asp?1026.

[10] T. Berlage. A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects. *ACM Transactions on Computer-Human Interaction*, 1(3):269–294, 1994.

[11] B. Berliner. CVS II: Parallelizing software development. *Proceedings of the 1990 USENIX Conference*, Washington, D.C., January 1990, 341–352.

[12] L. A. Bjork. Recovery Scenario for a DB/DC System. *Proceedings of the 1973 ACM Annual Conference*. Atlanta, GA, 1973, 142–146.

[13] J. B. Bowles, J. G. Dobbins. High-Availability Transaction Processing: Practical Experience in Availability Modeling and Analysis. *Proceedings of the 1999 Annual Reliability and Maintainability Symposium*, 268–273, 1999.

[14] A. B. Brown. Towards Availability and Maintainability Benchmarks: A Case Study of Software RAID Systems. *University of California, Berkeley Computer Science Division Technical Report UCB/CSD-01-1132*. Berkeley, CA, January 2001.

[15] A. B. Brown. Toward System-Wide Undo for Distributed Services. *University of California, Berkeley Computer Science Division Technical Report UCB//CSD-03-1298*. Berkeley, CA, December 2003.

[16] A. B. Brown and D. A. Patterson. Towards Availability Benchmarks: A Case Study of Software RAID Systems. *Proceedings of the 2000 USENIX Annual Technical Conference.* San Diego, CA, June 2000.

[17] A. B. Brown and D. A. Patterson. Rewind, Repair, Replay: Three R's to Dependability. *Proceedings of the 10th ACM SIGOPS European Workshop*. St. Emilion, France, September 2002.

[18] G. Candea and A. Fox. Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel. *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*. Schloss Elmau, Germany, May 2001.

[19] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[20] M. Chen, E. Kiciman, et al. Pinpoint: Problem Determination in Large, Dynamic, Internet Services. *Proceedings of the International Conference on Dependable Systems and Networks (IPDS Track)*. Washington D.C., 2002.

[21] R. Choudhary and P. Dewan. A General Multi-User Undo/Redo Model. *Proceedings of the Fourth European Conference on Computer-Supported Cooperative Work (ECSCW '95)*. Stockholm, Sweden, September 1995, 231–246.

[22] J. M. Christensen and J. M. Howard. Field Experience in Maintenance. *Human Detection and Diagnosis of System Failures: Proceedings of the NATO Symposium on Human Detection and Diagnosis of System Failures*, J. Rasmussen and W. Rouse (Eds.). New York: Plenum Press, 1981, 111–133.

[23] Compiere ERP + CRM Business Solution. http://sourceforge.net/projects/compiere.

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

[25] C. T. Davies. Data Processing Spheres of Control. *IBM Systems Journal*, 17(2):179–198, 1978.

[26] C. T. Davies. Recovery Semantics for a DB/DC System. *Proceedings of the 1973 ACM Annual Conference*. Atlanta, GA, 1973, 136–141.

[27] Digital. *VAX/VMS System Software Handbook*. Bedford, MA, 1985.

[28] B. Dijker. *A Day in the Life of System Administrators*. http://sageweb.sage.org/jobs/thefield/ditl.pdf, 1998.

[29] A. Dix, R. Mancini, and S. Levialdi. Alas I am Undone—Reducing the Risk of Interaction? *HCI '96 Adjunct Proceedings*. London, 1996, 51–56.

[30] G. Dunlap, S. King, et al. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. *Proceedings of the 5th Symposium on Operating System Design and Implementation (OSDI 2002)*. Boston, MA, December 2002.

[31] *eBay Inc. Announces Second Quarter 2003 Financial Results*. eBay, Inc. Press Release, 24 July 2003. http://www.shareholder.com/ebay/releases-earnings.cfm.

[32] W. K. Edwards. Flexible Conflict Detection and Management in Collaborative Applications. *Proceedings of the 10th ACM Symposium on User Interface Software and Technology (UIST)*. Banff, Canada, October 1997.

[33] W. K. Edwards, T. Igarashi et al. A Temporal Model for Multi-Level Undo and Redo. *Proceedings of the 13th ACM Symposium on User Interface Software and Technology (UIST)*. San Diego, CA, November 2000.

[34] W. K. Edwards and E. D. Mynatt. Timewarp: Techniques for Autonomous Collaboration. *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI)*. Atlanta, GA, March 1997.

[35] Electronic News. *EBay Outages Cast Clouds on Sun*. Electronic News, 21 June 1999. http://www.e-insite.net/electronicnews/index.asp?layout=article&articleid=CA70797&.

[36] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-passing Systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

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

[38] A. Fox and D. A. Patterson. When Does Fast Recovery Trump High Reliability? *Proceedings of the Second Workshop on Evaluating and Architecting Systems for Dependability (EASY)*. San Jose, CA, October 2002.

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

[40] I. Foster, C. Kesselman, and S. Tueckle. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.

[41] A. Fox, Personal communication, June 2003.

[42] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison Wesley, 1995.

[43] H. Garcia-Molina and K. Salem. Sagas. *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, San Francisco, CA, 1987, 249–259.

[44] R. F. Gordon, G. B. Leeman, and C. H. Lewis. Concepts and Implications of Undo for Interactive Recovery. *Proceedings of the 1985 ACM Annual Conference*, 150–157, 1985.

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

[46] J. Gray. A census of Tandem system availability between 1985 and 1990. *Tandem Computers Technical Report 90.1*, 1990.

[47] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. San Francisco: Morgan Kaufmann, 1993.

[48] R. E. Griswold, D. R. Hanson, and J. T. Korb. Generators in Icon. *ACM Transactions on Programming Languages and Systems*, 3(2):144–161, 1981.

[49] V. Grover and J. T. C. Teng. E-commerce and the Information Market. *Communications of the ACM*, 44(4):79–86, 2001.

[50] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. *Proceedings of the 11th ACM Symposium on Operating Systems Principles*. November 1987, 155–162.

[51] B. Harder. Microsoft Windows XP System Restore. *Microsoft MSDN Library.* http://msdn.microsoft.com/library/en-us/dnwxp/html/windowsxpsystemrestore.asp.

[52] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 3e*. San Francisco: Morgan Kaufmann, 2003.

[53] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS Server Appliance. *Proceedings of the 1994 USENIX Winter Technical Conference*. San Francisco, CA, January 1994.

[54] INPO. *An Analysis of Root Causes in 1983 Significant Event Reports*. INPO 84-027. Atlanta, GA: Institute of Nuclear Power Operations, 1984, as cited in [97].

[55] INPO. *A Maintenance Analysis of Safety Significant Events*. Nuclear Utility Management and Human Resources Committee, Maintenance Working Group. Atlanta, GA: Institute of Nuclear Power Operations, 1985, as cited in [97].

[56] M. Ivory and M. Hearst. The State of the Art in Automating Usability Evaluation. *ACM Computing Surveys*, 33(4):470–516, December 2001.

[57] K. Kanoun and H. Madeira. A Framework for Dependability Benchmarking. *Proceedings of the 2002 Workshop on Dependability Benchmarking,* in *DSN 2002 Supplement*. Washington, D.C., June 2003, F-7–8.

[58] B. H. Kantowitz and R. D. Sorkin. *Human Factors: Understanding People-System Relationships*. New York: Wiley, 1983.

[59] Kembel, R. *Fibre Channel: A Comprehensive Introduction*. Tucson, AZ: Northwest Learning Associates, 2000, p. 8.

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

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

[62] Kolstad, R. 1992 LISA Time Expenditure Survey. *;login:, the USENIX Association Newsletter*, 1992.

[63] Kolstad, R. Sysadmin Book of Knowledge. http://ace. delos.com/taxongate.

[64] H. F. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. *Proceedings of the 16th VLDB Conference*. Brisbane, Australia, 1990, 95–106.

[65] D. R. Kuhn. Sources of Failure in the Public Switched Telephone Network. *IEEE Computer* 30(4), April 1997.

[66] D. Kurlander and S. Feiner. Editable Graphical Histories. *Proceedings of the IEEE 1988 Workshop on Visual Languages*. Pittsburgh, PA, October 1988, 127–134.

[67] T. K. Landauer. Research Methods in Human-Computer Interaction. In *Handbook of Human-Computer Interaction, 2e*, M. Helander et al. (ed), Elsevier, 1997, 203–227.

[68] G. B. Leeman. A Formal Approach to Undo Operations in Programming Languages. *ACM Transactions on Programming Languages and Systems*, 8(1):50–87, 1986.

[69] P. Liu, P. Ammann, and S. Jajodia. Rewriting Histories: Recovering from Malicious Transactions. *Distributed and Parallel Databases*, 8:7–40, 2000.

[70] D. B. Lomet. Persistent Applications Using Generalized Redo Recovery. *Proceedings of the Fourteenth IEEE International Conference on Data Engineering*. Orlando, FL, 1998, 154–163.

[71] D. Lomet and M. Tuttle. A Theory of Redo Recovery. *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, San Diego, CA, June 2003.

[72] D. E. Lowell, S. Chandra, and P. Chen. Exploring Failure Transparency and the Limits of Generic Recovery. *Proceedings of the 4th Symposium on Operating System Design and Implementation (OSDI 2000)*. San Diego, CA, October 2000.

[73] A. P. Madden. After The PC. *Red Herring*. December 1998.

[74] H. Madeira and P. Koopman. Dependability Benchmarking: making choices in an n-dimensional problem space. *Proceedings of the 1st Workshop on Evaluating and Architecting System dependabilitY (EASY '01),* Göteborg, Sweden, July 2001.

[75] R. Mancini, A. J. Dix, and S. Levialdi. Dealing with Undo. *Proceedings of Interact '97*. Sydney, Australia, 1997.

[76] N. Mann. Position Paper. *Workshop "System Administrators are Users Too", ACM CHI 2003 Conference on Human Factors in Computing Systems*. Ft. Lauderdale, FL, April 2003. Available at http://www.cs.berkeley.edu/~mikechen/chi2003-sysadmin/papers/ NancyMann_Sun_PositionPaper.pdf.

[77] J. Menn. Prevention of Online Crashes is No Easy Fix. *Los Angeles Times,* 2 December 1999, C-1.

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

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

[80] 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.

[81] Netscape, Inc. *Mailstone Utility*. http://docs.iplanet.com/ docs/manuals/messaging/nms41/mailston/stone.htm.

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

[83] D. A. Norman. Categorization of action slips. *Psychological Review* 88:1–15, 1981.

[84] D. A. Norman. The Psychology of Everyday Things. New York: Basic Books, 1988.

[85] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS' 03)*. Seattle, WA, March 2003.

[86] Oracle. *Oracle 9i Flashback Query*. Oracle, Inc. White Paper, March 2002. http://otn.oracle.com/deploy/availability/pdf/FlashbackQueryBWP.pdf.

[87] Osage: Persistence Plus XML. http://sourceforge.net/projects/osage/.

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

[89] D. A. Patterson, A. B. Brown, et al. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. *University of California, Berkeley Computer Science Division Technical Report UCB//CSD-02-1175*, 15 March 2002.

[90] H. Pehlivan and I. Holyer. A Recovery Mechanism for Shells. *Computer Journal*, 43(3):168–176. Oxford University Press for the British Computer Society, 2000.

[91] R. H. Pope. Human Performance: What Improvement from Human Reliability Assessment. *Reliability Data Collection and Use in Risk and Availability Assessment: Proceedings of the 5th Eure-Data Conference*, H.-J. Wingender (Ed.). Berlin: Springer-Verlag, April 1986, 455–465.

[92] A. Prakash and M. J. Knister. A Framework for Undoing Actions in Collaborative Systems. *ACM Transactions on Computer-Human Interaction*, 1(4):295–330, 1994.

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

[94] L. Press. Personal computing: the post-PC era. *Communications of the ACM*, 42(10):21–24, 1999.

[95] Quartz Enterprise Job Scheduler. http://sourceforge.net/projects/quartz/.

[96] J. Rasmussen. *Information Processing and Human-Machine Interaction*. Amsterdam: North-Holland, 1986.

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

[98] J. Rekimoto. Time-Machine Computing: A Time-centric Approach for the Information Environment. *Proceedings of the 12th ACM Symposium on User Interface Software and Technology (UIST '99)*, 1999.

[99] M. Ressel and R. Gunzenhäuser. Reducing the Problems of Group Undo. *Proceedings of ACM GROUP '99*. Phoenix, AZ, 1999, 131–139.

[100] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using Abstraction to Improve Fault Tolerance. *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*. Banff, Alberta, Canada, October 2001.

[101] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

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

[103] W. B. Rouse. Models of human problem solving: Detection, diagnosis and compensation for system failures. *Proceedings of IFAC Conference on Analysis, Design and Evaluation of Man-Machine Systems*. Baden-Baden, Germany, September 1981.

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

[105] RUBiS: Rice University Bidding System. http://rubis.objectweb.org/.

[106] SAGE. http://www.sage.org.

[107] Y. Saito and M. Shapiro. Replication: Optimistic Approaches. *Hewlett-Packard Labs Technical Report HPL-2002-33*. Palo Alto, CA, February 2002.

[108] D. S. Santry, M. J. Feeley, N. C. Hutchinson et al. Deciding when to forget in the Elephant file system. *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*. Kiawah Island, SC, December 1999, 110–123.

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

[110] W. Sawyer. Case Studies from HP's 5nines Program: The Cost of Moving from 99% to 99.999% Availability. *Presentation at the Second Workshop of the High-Dependability Computing Consortium (HDCC), "Dependability in Real Life"*. Santa Cruz, CA, May 8, 2001.

[111] E. Schneider, C. Ferril et al. *Method, software and apparatus for saving, using and recovering data.* US Patent 6,016,553, January 2000.

[112] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Systems: Design and Evaluation, 3e.* Natick, MA: A K Peters, 1998.

[113] Sleepycat Software. *Berkeley DB Data Store*. http://www.sleepycat.com/products/data.shtml.

[114] T. J. Slegel, R. M. Averill III, M. A. Check, et al. IBM's S/390 G5 Microprocessor Design. *IEEE Micro*, 19(2):12–23, 1999.

[115] D. J. Sorin, M. M. K. Martin, et al. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA-29)*. Anchorage, Alaska, May 2002.

[116] L. Spainhower and T. Gregg. G4: A Fault-Tolerant CMOS Mainframe. *Proceedings of the 1998 International Symposium on Fault-Tolerant Computing*, 432–440, 1998.

[117] L. Spainhower, J. Isenberg, R. Chillarege et al. Design for Fault-Tolerance in System ES/9000 Model 900. *Proceedings of the 1992 International Symposium on Fault-Tolerant Computing*, 38–47, 1992.

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

[119] R. M. Stallman. *GNU Emacs Manual, 15th Ed*. Boston, MA: GNU Press, 2002.

[120] C. J. Stone. *A Course in Probability and Statistics*. Belmont, CA: Duxbury Press, 1996.

[121] R. E. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[122] C. Sun. Undo Any Operation at Any Time in Group Editors. *Proceedings of ACM CSCW.* Philadelphia, PA, 2000, 191–200.

[123] C. Sun and C. Ellis. Operational Transform in Real-Time Group Editors: Issues, Algorithms, and Achievements. *Proceedings of ACM CSCW '98*. Seattle, WA, 1998, 59–68.

[124] Sun Microsystems. *Java BluePrints: Java Pet Store Demo 1.3.1*. http://java.sun.com/blueprints/code/jps131/docs/index.html, 2003.

[125] T. Sweeney. No Time for DOWNTIME—IT Managers feel the heat to prevent outages that can cost millions of dollars. *InternetWeek*, n. 807, 3 April 2000.

[126] D. B. Terry, M. M. Theimer, et al. Managing Update Conflicts in a Bayou, a Weakly Connected Replicated Storage System. *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*. Copper Mountain, CO, December 1995.

[127] H. Thimbleby. *User Interface Design*. New York: ACM Press, 1990.

[128] M. Vieira and H. Madeira. Recovery and Performance Balance of a COTS DBMS in the Presence of Operator Faults. *Proceedings of the 2002 International Conference on Dependable Systems and Networks*. Washington, D.C., June 2002, 615–624.

[129] M. Vieira and H. Madeira. Definition of Faultloads Based on Operator Faults for DBMS Recovery Benchmarking. *Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing (PRDC2002)*. Tsukuba, Japan, 2002.

[130] M. Vieira and H. Madeira. Benchmarking the Dependability of Different OLTP Systems. *Proceedings of the 2003 International Conference on Dependable Systems and Networks*. San Francisco, CA, June 2003, 305–310.

[131] J. S. Vitter. US&R: A New Framework for Redoing. *IEEE Software*, 1(4):39–52, 1984.

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

[133] H. Wächter and A. Reuter. The ConTract Model. In *Database Transaction Models for Advanced Applications*, A. Elmagarmid (Ed.). San Francisco: Morgan Kaufmann, 1991, 219–263.

[134] A. Whitten and J. D. Tygar. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. *Proceedings of the 9th USENIX Security Symposium*, August 1999.

[135] Xpoint Technologies. XPoint Rapid Restore Server. http://www.xpointdirect.com/en/IBMR-RPC/RRServer.asp.

[136] Y. Yang. Undo support models. *International Journal on Man-Machine Studies*, 28:457–481, 1988.

[137] J. Zhu, J. Mauro, and I. Pramanick. System Recovery Benchmarking. *Proceedings of the 2002 Workshop on Dependability Benchmarking,* in *DSN 2002 Supplement*. Washington, D.C., June 2003, F-27–28.

[138] J. Zhu, J. Mauro, and I. Pramanick. Robustness Benchmarking for Hardware Maintenance Events. *Proceedings of the 2003 International Conference on Dependable Systems and Networks*. San Francisco, CA, June 2003, 115–122.