RECOVERY-ORIENTED COMPUTING

Towards a theory of Undo

Aaron Brown
UC Berkeley

June 2002 ROC Retreat

Outline

- **Recap of Undo: motivation and the 3 R's**
- **First implementation attempt & lessons learned**
- **Towards a theory for undo**
  - foundation: logging of application-level "verbs"
  - modeling verbs and undo history
  - properties of undo-wrappable systems
- **Status and conclusions**

Motivation for undo

- **Human error is a major impediment to dependability**
  - largest single contributing factor to outages
- **Undo is a recovery mechanism well-matched to coping with human (and non-human) error**
  - tolerates inevitable errors
  - harnesses hindsight and provides retroactive repair
    - » ~70% of human errors are immediately self-detected
  - supports trial & error exploration of complex systems
    - » allow operators to learn from mistakes

The 3R undo model

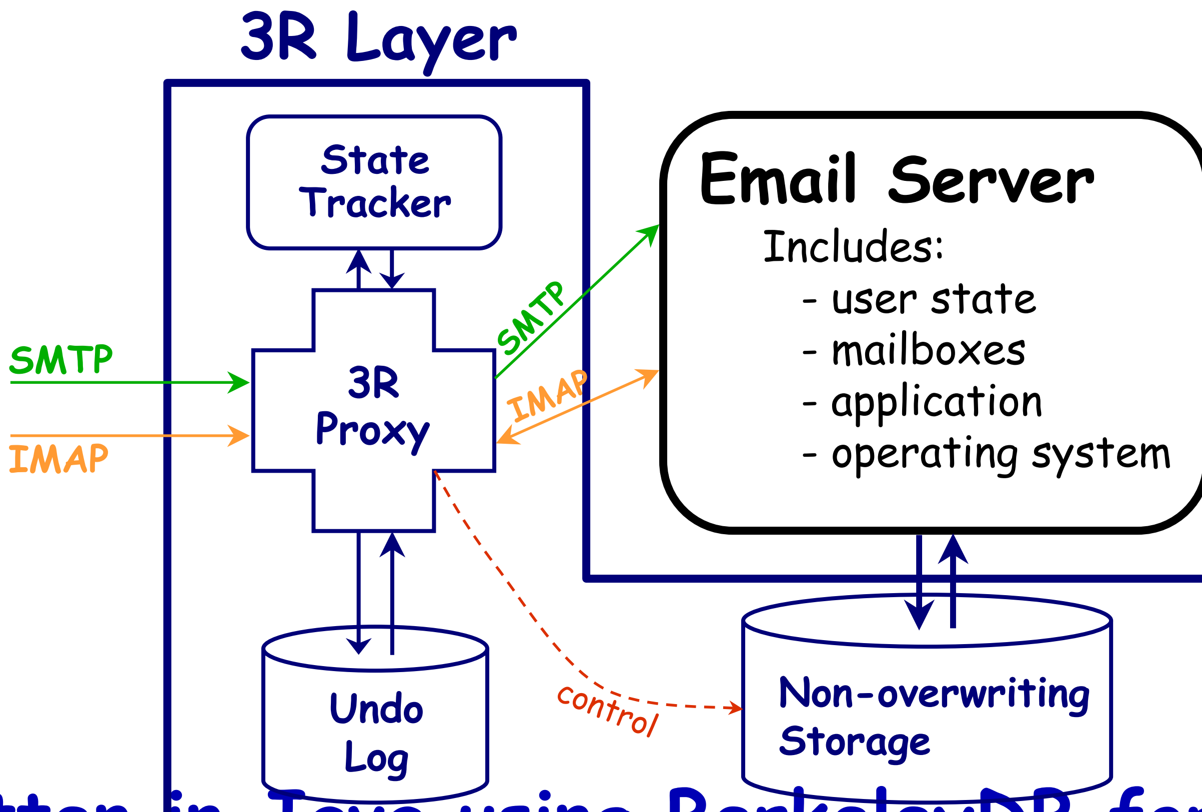
- **Undo == time travel for system operators**
- **Three R's for recovery**
  - **Rewind:** roll system state backwards in time
  - **Repair:** change system to prevent failure
    - » e.g., edit history, fix latent error, retry unsuccessful operation, install preventative patch
  - **Replay:** roll system state forward, replaying end-user interactions lost during rewind
- ***All three R's are critical***
  - rewind enables undo
  - repair lets user/administrator fix problems
  - replay preserves updates, propagates fixes forward

Challenges in 3R undo model

- **External consistency**
  - repair may alter state that's previously been seen by an external entity
- **Drawing the boundary of undo recovery**
  - want to recover content while allowing system state to change
- **Providing multiple-granularity undo**

First implementation attempt

- Undo wrapper for open source e-mail store



- Written in Java using BerkeleyDB for logging
  - partially completed: IMAP only, no integration w/FS

Lessons learned during 1st try

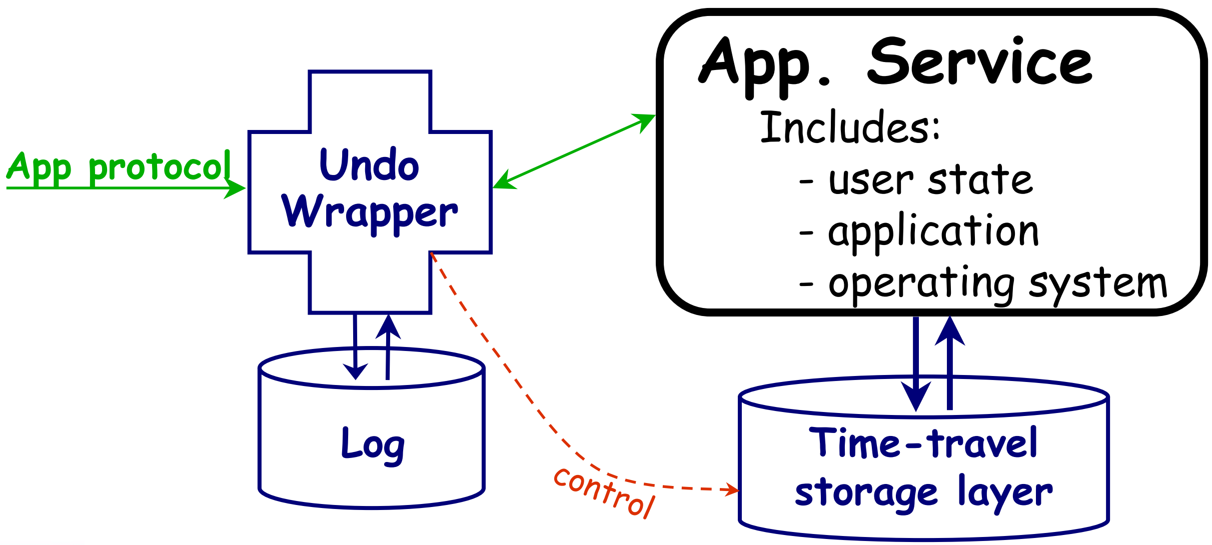
- **Undo wrapper is complex and error-prone**
  - deciding what to log is a challenge
  - have to anticipate all possible external inconsistencies
  - mechanics of log management & state tracking are ugly
- **Ad-hoc approach doesn't work**
  - bottom-up design => policy expressed procedurally
    - » hard to reason about, change, debug
  - no framework for making policy decisions
- **E-mail protocols are not conducive to undo-wrapping**
  - no GUIDs, incomplete command set, ...

A theory for undo

- **Goals:**
  - framework to reason about external inconsistencies generated by an undo cycle
  - framework to reason about correctness of undo implementation
  - template for undo-wrappable applications/services
  - guide to a more general implementation
- **Approach:**
  - model undo system structure and applications
  - map example apps (e-mail) onto model
  - build implementation following model

Foundation: undo system structure

- **An undoable system consists of:**
  - an application with a well-defined, non-procedural user interface (a *service*)
  - a stable storage layer supporting time travel
    - » snapshots, backups, non-overwriting/log-structured FS
  - an undo wrapper that logs and replays user/operator interactions with the application



Undo logging

- **Logging must capture user intent, not actual state changes**
  - software may be buggy => state changes may be wrong
  - repair, history deletions may invalidate physical logs
  - easier to reason about consistency with intentional logs
- **Undo system logs at a high semantic level**
  - user/operator application-level actions (*verbs*)
  - higher-level than DBMS logical logging
- **Fringe benefit: easy georeplication**
  - log shipping of high-level undo logs to remote site(s)
  - undo system provides all mechanisms, including resync
    - » and vice versa: georeplicated systems easy to undo?

Modeling undo logging

- **Application-client interface is specified as a set of *verbs***
  - verbs define actions on logically-named state entities
  - e-mail examples:
    - » deliver, fetch, set flags, delete, refile, create folder, ...
- ***Operations* are instances of verbs**
  - reflect actual user/operator interaction
- **The undo log is a *history* of operations**
  - during repair, the history may be modified
  - and other changes may be made to the system that aren't reflected in the history

Modeling operations

- **Each logged operation is modeled by:**
  - a verb specifying the action
  - a set of state entities needed to carry out the action
  - a set of preconditions over the state entities
    - » if satisfied, operation will produce same results as previous execution

→ used to classify operation as *safe* or *unsafe*

  - an indication of which state is modified
  - an indication of which state is externalized
  - a time specifying when results are externalized
    - » allows for delayed responses and "undo windows"

→ used to determine if unsafe state is externalized

Operations & external inconsistency

- **An operation is *safe* upon replay iff:**
  - the operation existed, unmodified, in the pre-repair history
  - all associated state entities exist
  - all preconditions are met
  - informally, the operation can execute and produces the same results as the original execution
- **Unsafe operations represent potential external inconsistencies**
  - but only if the modified (unsafe) state is externalized later in the history
    - » determined by following dependencies in history

Classifying histories

- **A history is *replay-safe* if:**
  - it contains only safe operations, OR
  - no unsafe operation modifies state that is externalized by a later operation in the history
  - these histories cause no visible inconsistencies
  - all pre-repair histories are replay-safe
- **A history is *replay-acceptable* if:**
  - it contains unsafe or deleted operations
  - the history can be made replay-safe by inserting appropriate compensating actions
  - these histories have acceptable visible inconsistency
- **Undo requires replay-acceptable histories!**

Making histories replay-acceptable

- **Step 1: identify unsafe operations**
  - check preconditions and existence of needed state
  - done dynamically during replay
- **Step 2: insert compensating actions**
  - compensations are inherently application-specific
  - explanatory compensations explain unsafe operations to user
    - » ex: "this message was deleted because it had a virus"
  - repairing compensations alter state to reestablish preconditions
    - » ex: create "lost&found" to stand in for nonexistent or read-only e-mail folder

Example e-mail scenario

- **Before undo:**
  - virus-laden message arrives
  - user copies it into a folder without looking at it
- **Operator invokes undo to install virus filter**
- **During replay:**
  - message is redelivered and discarded by virus filter
  - copy operation is unsafe
    - » violated precondition: existence of source messsage
  - copy operation externalizes existence of message
    - » history is replay-unsafe
  - compensating action: insert placeholder for message
    - » now copy can be executed; history is replay-acceptable

Guaranteeing replay-acceptability

- **A dependable undo system must be able to make any history replay-acceptable**
  - operation templates (verbs) must be specified correctly
    - » all needed preconditions and no extraneous ones
  - compensations must exist for all precondition violations
    - » explicit compensations or dummy compensations that allow the inconsistency to pass through
  - precondition and compensation logic must be correct
    - » model identifies cases for exhaustive testing

Recap: model benefits

- **Simplifies reasoning about undo inconsistency**
  - expressed in terms of preconditions & compensations
- **Provides greater confidence in undo**
  - by construction, if preconditions are correct and compensations exist, all scenarios will produce acceptable external consistency
  - declarative specifications of verbs, preconditions, and compensations are easier to write and check
  - model provides guidance for exhaustive testing
- **Provides framework for general implementation**
  - can separate app-specific policy from undo mechanisms
- **Implicitly defines properties of applications that can be wrapped for undo**

Implications for applications

- **Model induces a set of properties for undo-wrappable applications**
  - a high-level, verb-structured interface/API for user, operator, and external actions
  - a state model where all state is nameable via the API and tagged with GUIDs
  - a "complete" API where each an inverse for each verb exists or can be constructed
  - external consistency semantics that permit compensation for non-commuting or non-replayable verbs

Implications for applications

- **Model induces a set of properties for undo-wrappable applications**
  + a high-level, verb-structured interface/API for user, operator, and external actions
  – a state model where all state is nameable via the API and tagged with GUIDs
  – a "complete" API where each an inverse for each verb exists or can be constructed
  + external consistency semantics that permit compensation for non-commuting or non-replayable verbs
- **Example: IMAP/SMTP-based e-mail**

Possible future benefits

- **Automated consistency analysis**
  - model allows identification of non-replay-safe histories
    - » as described, cannot be done statically since preconditions are dynamic
  - model could be extended to pre-compute expected inconsistencies before executing repair/replay
    - » "what-if" analysis of repair impact
    - » requires expanding verb definitions with specification of expected state changes
  - given buggy software and arbitrary repairs, automated analysis would be just a hint
    - » would provide "best-case" answer assuming perfect SW
    - » could compare with dynamic analysis to identify bugs?

Status and conclusions

- **Status**
  - continuing model development using e-mail as driver
    - » next step: try to better formalize compensations
  - restarting implementation to follow the model
    - » declarative specification of verbs and a general mechanism layer
- **Conclusions**
  - model-based approach to undo provides needed framework for reasoning about undo behavior
    - » simplifies specification of application policy
    - » enhances confidence in implementation
    - » may lead to automated "what-if" consistency analysis

Properties of operations

- **Two operations O_1 and O_2 *commute* if:**
  - O_1 and O_2 have disjoint state sets, OR
  - state modified by O_1 is not part of O_2's state set, OR
  - O_1's modifications to common state do not violate O_2's preconditions and are not externalized by O_2
  - essentially, O_2 isn't affected by changes to O_1
- **An operation is *replayable* if:**
  - all needed state exists at replay time
  - all preconditions are satisfied at replay time
  - the operation succeeded, or, if it failed, the time between failure and replay is less than the delay