Embracing Failure: Availability via Recovery-Oriented Computing (ROC)

Aaron Brown

ROC Research Group
Computer Science Division, UC Berkeley
abrown@cs.berkeley.edu
Outline

• Motivation for ROC
• Principles of ROC design
• Initial ROC implementation target
• Evaluating ROC: availability benchmarks
• Summary
Motivation for a new philosophy

• Internet service availability is a big concern
  - outages are frequent
    » 65% of IT managers report that their websites were unavailable to customers over a 6-month period
    • 25%: 3 or more outages
  - outages costs are high
    » NYC stockbroker: $6,500,000/hr
    » EBay: $ 225,000/hr
    » Amazon.com: $ 180,000/hr
    » social effects: negative press, loss of customers who “click over” to competitor

• Why?

Source: Internet Week 4/3/2000
Traditional HA vs. Internet reality

• Traditional HA env’t
  - stable
    » functionality
    » software
    » workload and scale
  - high-quality infrastructure designed for high availability
    » robust hardware: fail-fast, duplication, error checking
    » custom, well-tested, single-app software
    » single-vendor systems
  - certified maintenance
    » phone-home reporting
    » trained vendor technicians

• Internet service env’t
  - dynamic and evolving
    » weekly functionality changes
    » rapid software development
    » unpredictable workload and fast growth
  - commodity infrastructure coerced into high availability
    » cheap hardware lacking extensive error-checking
    » poorly-tested software cobbled together from off-the-shelf and custom code
    » multi-vendor systems
  - ad-hoc maintenance
    » by local or co-lo. techs
Facts of life

• Realities of Internet service environment:
  - hardware and software failures are inevitable
    » hardware reliability still imperfect
    » software reliability thwarted by rapid evolution
    » Internet system scale exposes second-order failure modes
  - unanticipated failures are inevitable
    » commodity components do not fail cleanly
    » black-box system design thwarts models
    » seemingly-obscure failure modes are normal
  - human operators are imperfect
    » human error accounts for ~50% of all system failures
    » human error probability is 10%-100% under stress

• Traditional HA doesn't address these realities!

Sources: Gray86, Hamilton99, Kuhn97, Menn99, Murphy95, Perrow99, Pope86
Recovery-Oriented Computing (ROC)

“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

• Failures are a fact, and recovery/repair is how we cope with them

• Hypothesis: improving recovery will improve availability
  
  - availability = \( \frac{MTTF}{(MTTF + MTTR)} \)
ROC systems

- A recovery-oriented system
  - uses recovery and repair to tolerate failures of hardware, software, and humans
  - provides rapid recovery
    » efficiently detects and diagnoses failures
  - provides effective recovery
    » proactively verifies efficacy and speed of repair procedures
  - provides robust recovery
    » tolerates errors during repair and maintenance
Context: ROC design

• Vs. traditional fault-tolerance approaches
  - different philosophy
    » traditional: focus on HW; assume good software, operators
      • build good SW by controlling development, modeling
    » repair-centric: assume that any HW, SW, operator can fail
      • assume environment too dynamic to control or model
  - some shared techniques
    » testing, checkpoints, fault-injection, diagnosis
    » but applied differently: online, system-wide, without models

• Other existing recovery-oriented approaches
  - restartable systems
    » Recursive Restartability, soft-state worker frameworks
  - application-level checkpoint recovery

Sources: Candea01, Fox97, Lowell98, Lowell00, Ninja01
Outline

• Motivation for ROC

• Principles of ROC design

• Initial ROC implementation target

• Evaluating ROC: availability benchmarks

• Summary
Approaching ROC design

• Tentative principles of ROC design
  1) isolation and redundancy: fault containment
     » prevent failure propagation and enable proactive testing
  2) online verification: fully-integrated online testing
     » detect failures quickly to expedite repair
     » provide trust in repair mechanisms and human operators
  3) undo: the ultimate repair mechanism?
     » tolerate human error and repair unanticipated failures
  4) diagnosis: dependency and fault tracking
     » assist operator in pinpointing failures to expedite repair
(1) Isolation and redundancy

- **System is redundant**
  - sufficient HW redundancy/data replication => part of system down but satisfactory service still available
  - enough to survive 2\textsuperscript{nd} failure or more during recovery

- **System is partitionable**
  - to isolate faults
  - to enable online repair/recovery
  - to enable online HW growth/SW upgrade
  - to enable operator training/expand experience on portions of real system
Approaches to isolation

- **Shared-nothing cluster design**
  - no shared storage between nodes
  - total physical partitioning of nodes possible via network disconnection
  - system versions can coexist: easy expansion, upgrades

- **HW support to limit scope of faults**
  - separate address spaces whenever possible
  - queue-based communication between processes
  - read/write protection of memory pages
  - physical (electrical) network partitioning

- **Geographic replication for last-resort isolation**
(2) Online verification

- System enables input insertion, output check of all modules (including fault insertion)
  - to check module operation to find failures faster
    » correctness and performance
  - to test correctness of recovery mechanisms
    » insert faults and known-incorrect inputs
    » also enables availability benchmarks
  - to discover if warning systems are broken
  - to expose and remove latent errors from each system
  - to train/expand experience of operator
More online verification

• Modules (HW and SW) perform redundant calculation to help discover errors
  - program checking analogy: if computation is $O(n^x)$, $(x > 1)$ and if check is $O(n)$, little cost to check
  - extension of assertion checking, checksums, ECC-like approaches to all software and hardware

• System proactively discovers its configuration
  - including interconnect and power supply topology, etc.
  - verifies available redundancy, thwarts human mistakes

• System continuously verifies global invariants
  - use “conservation law analysis” as in industrial plants to prevent loss, misdirection of data
Online verification of operators

- To expand operator experience beyond normal events, regular fault insertion on live system
  - provide training for new operators
  - familiarize operators with failure modes, repair tasks
    » reduce human error potential
  - test operator performance during repair
    » results reflected back to management to discover in advance if there is a people problem
  - use partitioning and isolation mechanisms to protect production data during testing/training
(3) Undo

• **ROC system should offer Undo**
  - to recover from operator errors
    » undo is ubiquitous in productivity apps
    » should have “undo for maintenance”
  - to recover from inevitable SW errors
    » restore entire system state to pre-error version
  - to recover from operator training via fault-insertion
  - to replace traditional backup and restore?

• **Implement using checkpoint and logging technology**
  - restrict semantics and granularity for simpler implementation, lower overhead
(4) Diagnosis

• System assists human in diagnosing problems
  - root-cause analysis to suggest possible failure points
    » track resource dependencies of all requests
    » correlate symptomatic requests with component dependency model to isolate culprit components
  - “health” reporting to detect failed/failing components
    » failure information, self-test results propagated upwards
  - unified status console to highlight improper behavior, predict failure, and suggest corrective action

• Log faults, errors, failures and recovery
  - to create a library of failures
    » for future diagnoses, training, fault-injection, and research
Outline

• Motivation for ROC

• Principles of ROC design

• Initial ROC implementation target

• Evaluating ROC: availability benchmarks

• Summary
First ROC implementation target

- **Hardware: ROC-I cluster**
  - 64-node PC cluster with integrated storage
  - special features for ROC-based high availability
    » support for hardware fault-injection
    » support for partitioning at the electrical level
    » support for topology discovery of network and power
    » highly instrumented hardware enables online HW verification
    » integrated diagnostic system: per-node diagnostic processors and independent diagnostic network
  - modular, cable-less “brick” design enables easy maintenance, reduces human-induced HW failures
ROC-I Brick Node

- Pentium-II/266
- 256 MB DRAM
- 18 GB SCSI (or IDE) disk
- 4x100Mb Ethernet
- m68k diagnostic processor & CAN diagnostic network
- Packaged in standard half-height RAID array canister
ROC-I system

- 64-node cluster of nodes, 1.1TB storage
  - cluster nodes are plug-and-play, intelligent, network-attached storage “bricks”
    » a single field-replaceable unit to simplify maintenance
  - more CPU per disk than NAS or cluster architectures

ROC-I Chassis
64 nodes, 8 per tray
2 levels of switches
  • 20 100 Mb/s
  • 2 1 Gb/s
Environment Monitoring:
  UPS, redundant PS, fans, heat and vibration sensors...

Storage-Oriented Node “Brick”
Portable PC CPU: Pentium II/266 + DRAM
Redundant NICs (4 100 Mb/s links)
Diagnostic Processor
Disk
Half-height canister
First ROC implementation target

- **Software application: Internet email service**
  - simple, but enough complexity to be interesting
    » hard state, rich data, relaxed consistency requirements

- techniques for email should generalize
  » but stronger consistency may add complexity

- proposed base email implementation: NinjaMail
  » research implementation from UCB Ninja group
  » provides needed infrastructure for investigating ROC
Outline

• Motivation for ROC

• Principles of ROC design

• Initial ROC implementation target

• Evaluating ROC: availability benchmarks

• Summary
Evaluating ROC systems

• Traditional benchmarks focus on performance
  - ignore availability
  - assume perfect hardware, software, human operators

• Evaluating ROC requires evaluating availability gains from repair-oriented design techniques
  - requires availability benchmarking
    » a technique we developed in earlier work
Availability benchmarking 101

- Availability benchmarks quantify system behavior under failures and maintenance

- They require
  - a realistic workload for the system
  - quality of service metrics and tools to measure them
  - fault-injection to simulate failures
  - human operators to perform repairs
Example: email application

- **Workload**
  - SPECmail2001 industry-standard email benchmark

- **Quality of service metrics**
  - performance (SPECmail messages per minute)
  - error rate (lost or corrupted messages and mailboxes)
  - consistency (fraction of inconsistent mailboxes)
  - human maintenance time and error rate
Fault injection

- **Fault workload**
  - must accurately reflect failure modes of real-world Internet service environments
    - plus random tests to increase coverage, simulate Heisenbugs
  - but, no existing public failure dataset
    - we have to collect this data
    - a challenge due to proprietary nature of data
    - interest expressed by Microsoft, IBM, and Hotmail
  - major contribution will be to collect, anonymize, and publish a modern set of failure data

- **Fault injection harness**
  - build into system: needed anyway for online verification
Evaluating ROC: human aspects

• **Must include humans in availability benchmarks**
  - to verify effectiveness of undo, training, diagnostics
  - humans act as system administrators

• **Subjects should be admin-savvy**
  - system administrators
  - CS graduate students

• **Challenge will be compressing timescale**
  - i.e., for evaluating training

• **We have some experience with these trials**
  - earlier work in maintainability benchmarks used 5-person pilot study
Summary

• **ROC: Recovery-Oriented Computing**
  - a new approach to increasing availability by focusing on recovery and repair
  - based on realities of today’s Internet service env’t
  - tackles the universally-ignored problem of human error

• **A departure from traditional HA philosophy**
  - embracing failure, not attempting perfection
  - model of proactive testing/verification, on live systems

• **ROC offers the potential for unprecedented advances in availability**
Acknowledgements

• Dave Patterson

• Thesis committee
  - Dave Patterson, Armando Fox, James Hamilton, Kathy Yelick, John Chuang

• UC Berkeley ISTORE group
  - Eric Anderson, Jim Beck, Dan Hettena, Jon Kuroda, David Martin, David Oppenheimer, Noah Treuhaft

• Industry supporters
  - Bill Tetzlaff, Brendan Murphy, Gautam Kar

• Questions or comments?
  - email: abrown@cs.berkeley.edu
  - www: http://istore.cs.berkeley.edu/
End
Contributions

• New philosophy for high-availability design

• Definition of repair-centric design techniques
  - addressing hardware, software, and human failures

• Prototype repair-centric system implementation

• Quantitative, human-aware availability evaluation methodology
  - including collection and characterization of data on real-world system failure modes and maintenance tasks
Human error rate experiments

- Human error rates during simple RAID repair
  - 5 trained subjects repeatedly repairing disk failures
  - Aggregate error rate across subjects plotted over time
What causes un-availability?

- Many different factors are involved
  - human behavior during maintenance dominates

Source: Murphy95
How does ROC differ from Fault Tolerant Computing?

- Systems like Tandem, IBM mainframes concentrate on Hardware Failures
  - Mirrored disks, Redundant cross-checked CPUs, ...
  - Designed to handle 1 failure until repaired
- Also some work on Software failures: Tandem’s process pairs, transactions, ...
  - Rather than embracing failure, goal is SW perfection
- No attention to human failures
- FTC works on improving reliability vs. recovery/repair
- Generally ROC is synergistic with FTC
Traditional HA vs. repair-centric

- **Traditional HA system**
  - hardware-centric focus
  - assumes robust software
    » by controlling entire stack
  - assumes robust operator
    » by controlling maintenance
  - may not tolerate errors during repair/maintenance

- **Repair-centric system**
  - tolerates hardware, software, human errors
  - assumes black-box software stack
  - tolerates operator error
  - tolerates errors during maintenance/repair
Assumptions

• **Cluster-like environment**
  - replicated data and services
  - partitionable hardware
• **Single-application system**
• **Modular HW/SW design**
• **Availability trumps performance**
  - willing to sacrifice performance to increase availability
• **Extra resources are available**
  - willing to overprovision resources to improve availability
    » especially inexpensive disks and disk bandwidth
Undo

• **Undo definition**
  - undo restores modified system state to a previous snapshot while preserving externally-initiated updates
    » i.e., for email, it restores state while preserving mail delivery and user mailbox modifications

• **Undo is the most fundamental repair-centric design mechanism**
  - provides a way to tolerate human errors
    » undo is ubiquitous in productivity apps
    » should have “undo for maintenance”
  - allows recovery from inevitable HW/SW errors
    » restore entire system state to pre-error version
  - subsumes traditional backup and restore
Undo examples

- **Tolerating human maintenance errors**
  - operator disconnects wrong component during repair
    » undo: replace component, system continues normally
  - operator installs software upgrade that corrupts data or performs poorly (E*Trade, EBay)
    » undo: roll-back upgrade, restore uncorrupted data, replay interim requests
  - operator overwrites data store or critical config file
    » undo: restore data store, config state; replay lost requests

- **Tolerating failures**
  - hardware or software failure corrupts data
    » undo: restore snapshot and replay interim requests
  - system destabilizes when new hardware is added
    » undo: revert system configuration state to disable hardware
**Undo context**

- Similar to existing checkpoint techniques...
  - file system snapshots (e.g., NetApp)
  - DBMS log-based recovery
  - application checkpointing for failure recovery

- …but with some new twists
  - use for tolerating human mistakes
  - use at system level as well as application level
    » mandatory for tolerating errors during repair/maintenance
  - preservation of externally-initiated updates
    » logging/replay at external interfaces and full state restoration avoid inherent save-work/lose-work conflict

*Sources: Hitz95, Lowell98, Lowell00, Mohan92*
Undo implementation

• As a repair mechanism, undo must be simple
  - no complex fine-grained distributed checkpoints, etc.

• Two types of simple undo
  1) allow replacement of incorrectly-removed components
     » enforce queuing in front of all removable resources
     » spill queues to disk to allow reasonable replacement window
     » Ninja’s queue-based communication model should match well
  2) coarse-grained maintenance-undo of system state
     » provide cluster-wide hard state rollback mechanism with
       preservation of external updates (like mail delivery)
     » leverage properties of email service to simplify
       implementation
Undo implementation (2)

- **Coarse-grained maintenance undo**
  - use standard snapshot and logging techniques
  - restrict semantics to simplify implementation
    » coarse-grained in space: undo affects entire cluster partition
    » coarse-grained in time: undo rolls back to a previous snapshot
    » undo restores only system hard-state
      • software, config. files, mail store contents
      • updates preserved by logging and replaying at external interfaces
      • enabled by Ninja design of stateless workers
  - these semantics are sufficient
    » coarse granularity is appropriate for a repair mechanism
    » email can tolerate inconsistencies during undo/rollback
Undo issues

• Open issues in implementing undo
  - defining undo points
    » simplest: a special “undo mode” for tolerating human error
    » but periodic snapshots are needed for repairing unanticipated failures
  - snapshot and logging mechanisms
    » overhead affects granularity of undo points
    » with cheap disks and disk bandwidth, are simple but high-overhead schemes acceptable?
  - protecting undo from failures
    » snapshots, external request logs must be independent
    » undo should be tested like any repair mechanism: stage 3
Stage 2: Online verification

• **Goal:** expedite repair
  - expose latent problems for repair
  - reduce failure propagation with faster detection

• **Techniques**
  - continuously verify HW & SW component operation
    » check correctness to detect bugs and hard failures
    » check performance to detect bottlenecks and soft failures
    » use real test inputs, not heartbeats
  - add verification at all component interfaces
    » check received data against specifications, checksums
  - check global system properties
    » use “conservation law analysis” as in industrial plants [Lind81] to prevent loss, misdirection of data
Issues in online verification

• **Standard testing issues**
  - input selection, result verification, coverage analysis

• **Online testing challenges**
  - ensuring non-destructive operation
    » perform testing on an isolated partition of the cluster
    » use hardware isolation and existing Ninja partitioning and node-reincorporation mechanisms
  - detecting dynamic performance problems
    » check all tests against running statistical estimates of range of normal performance

• **Developing global conservation laws for email**
  - example: rate of incoming messages must equal sum of rates of additions to user mailboxes
Stage 3: Exercising repair

- Repair mechanisms are often untrustworthy
  - buggy automatic recovery code
  - humans unfamiliar with system repair procedures

- **Goal:** proactively verify repair mechanisms by exercising them in realistic environment
  - detect broken recovery code so it isn’t relied on
  - provide framework for testing recovery code
  - familiarize operators with failure modes and repair procedures, and test them

- **Basic technique:** fault-injection
  - performed in online, production system!
Exercising repair: approach

• Inject realistic faults to simulate failures
  - targeted faults simulate most likely failure modes
  - random faults capture tail of the failure distribution

• Allow automatic recovery attempt
  - if recovery fails or is not available, log fault and use in human exercises
    » approach is self-tuning for level of automatic recovery

• Perform human training/testing
  - using fault set that failed automatic recovery

• Do testing on isolated subset of system
  - to avoid damage to production system
Issues in exercising repair

• **Fault injection**
  - need realistic fault set and injection harness
  - also needed for evaluation -> discussed later

• **Verification**
  - straightforward for targeted faults
    » effects are known
  - a challenge for random faults
    » use stage 2 testing and verification infrastructure

• **Protection**
  - use partition-isolation mechanisms from stage 2
Stage 4: Diagnosis aids

- **Goal**: assist human diagnosis, not subsume it
  - reduce space of possible root causes of failure
  - provide detailed “health status” of all components

- **Technique #1: dependency analysis**
  - model dependencies of requests on system resources
    - use model to identify potential resource failures when a request fails
    - correlate dependencies across symptomatic requests to reduce failure set
  - generate model dynamically
    - stamp requests with ID of each resource/queue they touch
  - issues
    - tracking dependencies across decoupling points
    - accounting for failures in background non-request processing
Diagnosis aids

- Technique #2: propagating fault information
  - explicitly propagate component failure and recovery information upward
    » provide “health status” of all components
    » can attempt to mask symptoms, but still inform upper layers
    » rely on online verification infrastructure for detection
- issues
  » devising a general representation for health information
  » using health information to let application participate in repair
## Details: application spectrum

<table>
<thead>
<tr>
<th>Application</th>
<th>Hard state</th>
<th>Consistency requirement</th>
<th>Interface complexity</th>
<th>Internal knowledge of data semantics</th>
<th>Query complexity</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL database</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>E-commerce app. server</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>12</td>
</tr>
<tr>
<td>Email</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>File server</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Search engine</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>Block server</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Transforming proxy</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Web server</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
**Context: undo**

- **Undo is common for application recovery**
  - database transaction rollback
  - checkpoint/restore of long-running scientific codes
  - app. checkpointing may help tolerate Heisenbugs

- **But is rare at the system level**
  - only common example is snapshotting file systems
    » Network Appliance, new BSD FFS, Elephant, etc.
  - system-level undo needed to handle maintenance errors

- **Implementing undo requires implementing standard recovery techniques at system level**
  - checkpointing, logging, snapshots, . . .

*Sources: Hitz95, Lowell98, Lowell00, Mohan92*
Context: exercising repair

- Similar to traditional “fire-drill” testing
  - but automated, so it really gets done
  - unique to perform testing in context of live system using fault-injection
- Training aspect is similar to offline training
  - Tandem’s “uptime champion” uses pilot-system-trained operators to increase availability
  - aircraft industry has long-standing tradition of simulator-based training to reduce human error
  - our approach provides same, but on live system
- Built-in fault injection similar to mainframes
  - IBM 3090, ES/9000 used built-in fault injection, but only during test-floor burn-in

Sources: Bartlett01, Merenda92, Nawrocki81
Context: online verification

• Most existing approaches are in hardware
  - lockstep hardware in mainframe and FT systems
  - ECC and other hardware verification schemes
  - hardware Built-In-Self-Test (BIST), online & offline

• Online software techniques are usually ad-hoc
  - assertion checking
  - heartbeats
  - checksums

• We systematically extend hardware techniques to software and system level

Sources: Gray86, Spainhower92, Spainhower98, Steininger99
Context: diagnosis

- **One-off system-specific diagnosis aids**
  - NetApp network diagnoser: cross-layer correlation and expert-system approaches

- **General diagnostic methods**
  - expert systems and fault-tree approaches
    - all require good understanding/model of failure modes, and thus conflict with real-world observations
  - dependency-based root-cause analysis
    - requires system model, but only at level of resource dependencies
    - our request-tracing approach dynamically discovers resource dependency model

Sources: Banga00, Brown01, Kar00, Orge92
What we’re NOT trying to do

• Invent new recovery mechanisms for NinjaMail
  - orthogonal
• Remove the human operator from the loop
  - unrealistic. But we can maybe simplify their job.
• Eliminate human errors completely
  - impossible
• Guarantee fault detection, fail-stop behavior
  - orthogonal: byzantine fault-tolerance
• Precisely auto-diagnose failure root causes
• Build the world’s fastest email service
  - willing to sacrifice performance for effective repair
Ninja details

- **Framework for cluster-based Internet services**
  - SPMC programming model
  - built-in mechanisms
    » clone groups (virtual nodes)
    » partitions
    » FE connection manager
    » asynchronous comm. layer
  - built-in services
    » distributed hash table
    » streaming, txnal file system
  - size: ~20,000 lines of code
    » NinjaMail: ~3,000
    » file system: ~5,000
    » hash table: ~12,000
Context: repair-centric design

- The philosophy of repair-centric design is rarely seen
  - mostly found in “restartable systems”
    » Recursive Restartability repairs Heisenbugs via reboot
    » soft-state designs (TACC, Ninja, some production services) tolerate coding errors by restarting errant workers
  - our approach is much broader and adds human focus
    » almost no work in systems and fault-tolerance community on tolerating human error
    » UI work minimizes human errors, but cannot prevent entirely

- Some repair-centric mechanisms more common
  - but not in service to repair-centric philosophy
  - unique: maintenance undo, proactive verification via online fault-injection

Sources: Candea01, Fox97, Ninja01
• **At minimum, committed to:**
  - stage 1 (undo) and stage 3 (exercising repair)
  - a partial implementation of stage 2 (online verification)
  - failure data collection
  - availability benchmarking using human trials
Research Plan

• Evaluate the repair-centric hypothesis by
  - identifying repair-centric design techniques
  - implementing the design techniques in a prototype
  - assessing the resulting availability improvements using availability benchmarks

• Target application: Internet email service

• Staged research plan
  - addresses practical concerns of scope, new grads
  - provides coherent fallback positions
Context: implementation platform

- **Base implementation: NinjaMail**
  - research implementation from UCB Ninja group
  - already implements non-repair-centric HA techniques
    » clustered, replicated, load-balanced, modular, restartable
  - written in Java in the Ninja environment
    » low-level Ninja mechanisms useful for repair-centric design
  - using existing system increases relevance, saves work
Staged research plan

- Techniques for ROC
  - 1) fault isolation
    1) **undo**: the ultimate repair mechanism
       » tolerate human error and repair unanticipated failures
    2) **online verification**: fully-integrated online testing
       » detect failures quickly to expedite repair
    3) **exercising repair**: online fault-injection
       » provide trust in repair mechanisms and train operators
    4) **diagnosis**: dependency and fault tracking
       » assist operator in pinpointing failures to expedite repair

- Evaluation can be done after any stage