

Embracing Failure: Availability via Repair-centric Design

Aaron Brown

Qualifying Examination
13 April 2001

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

Outline

- Grand vision: repair-centric design philosophy
- Research and implementation plan
- Evaluation plan
- Summary and timeline

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
 - but, despite marketing, progress seems slow. . .
- **Why?**

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
 - system failure modes cannot be modeled or predicted
 - » commodity components do not fail cleanly
 - » black-box system design thwarts models
 - » unanticipated failures 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!**

Repair-centric hypothesis

“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 repair is how we cope with them
- Improving repair will improve availability
 - availability =
$$\frac{\text{MTTF}}{(\text{MTTF} + \text{MTTR})}$$

Repair-centric systems

- **A repair-centric system**
 - uses repair to tolerate failures of hardware, software, and humans
 - provides **rapid** repair
 - » efficiently detects and diagnoses failures
 - provides **effective** repair
 - » proactively verifies efficacy and speed of repair procedures
 - provides **robust** repair
 - » tolerates errors during repair and maintenance

Context: repair-centric 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 repair-centric approaches**
 - restartable systems
 - » Recursive Restartability, soft-state worker frameworks
 - application-level checkpoint recovery

Outline

- Grand vision: repair-centric design philosophy
- **Research and implementation plan**
- Evaluation plan
- Summary and timeline

Research Plan

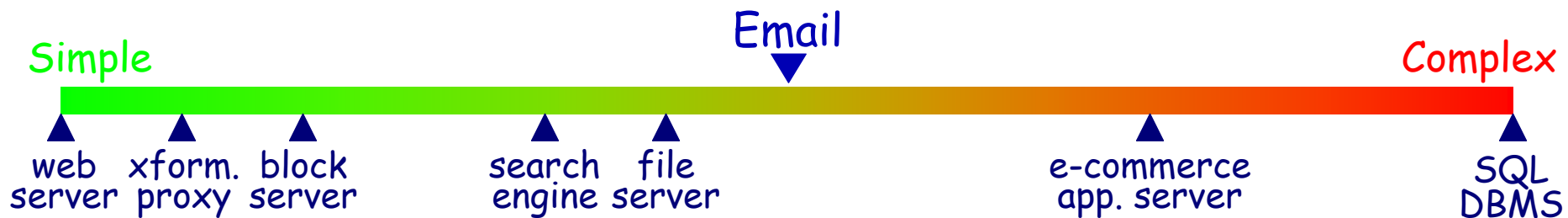
- **Evaluate the repair-centric hypothesis by**
 - identifying repair-centric design techniques
 - implementing the design techniques in a prototype
 - assessing the resulting availability improvements using availability benchmarks
- **Target application: Internet email service**
- **Staged research plan**
 - addresses practical concerns of scope, new grads
 - provides coherent fallback positions

Context: application

- **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

- realistic workloads available for email

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

- **Big picture of the stages**
 - 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**

Stage 1: 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

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

Outline

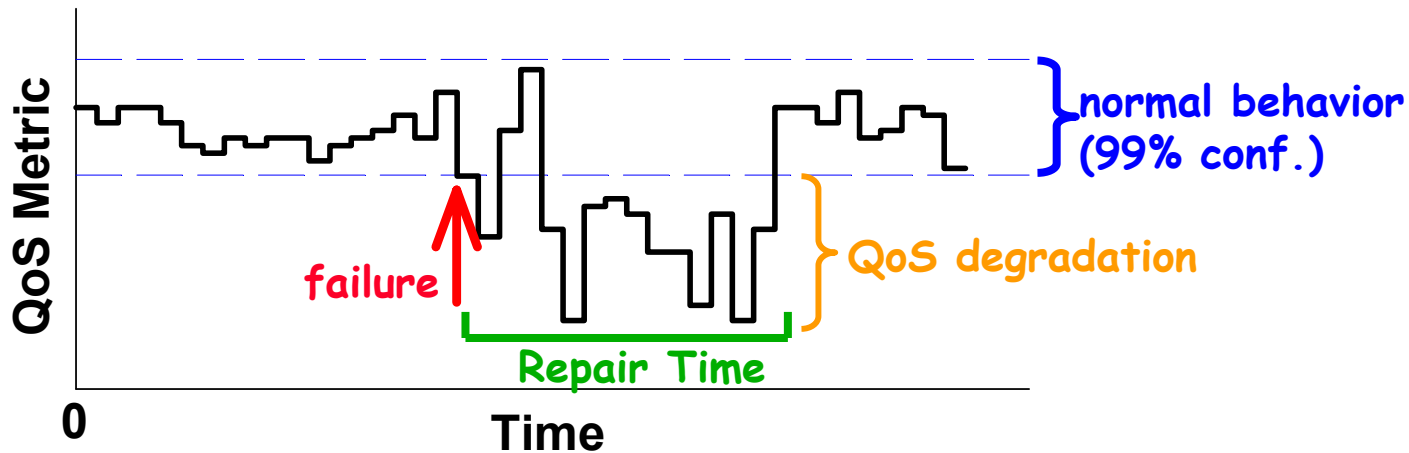
- Grand vision: repair-centric design philosophy
- Research and implementation plan
- **Evaluation plan**
- Summary and timeline

Evaluation plan

- **Goal: evaluate overall availability gains from repair-centric design**
 - compare modified, repair-centric NinjaMail to stock implementation
- **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

Availability benchmarks for email

- **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 for stage 3 (exercising repair)

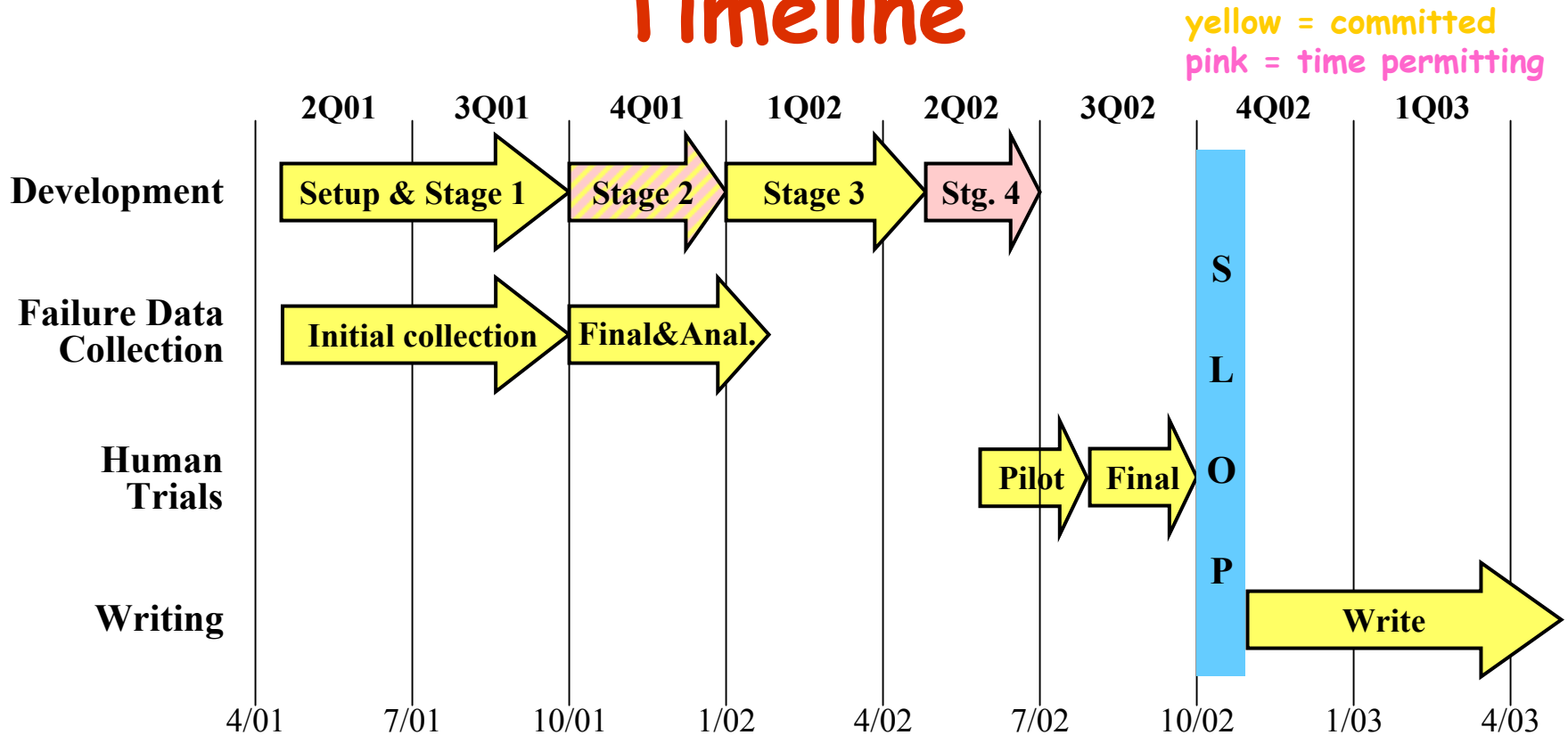
Evaluation: human aspects

- **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

- **Repair-centric design hypothesis**
 - a new approach to increasing availability by focusing on repair
 - based on realities of today's Internet service env't
 - tackles the universally-ignored problem of human error
- **Prototyping plan in NinjaMail email service**
- **Evaluation plan using availability benchmarks**
- **If successful, a significant contribution to state-of-the-art in high-availability design**

Timeline

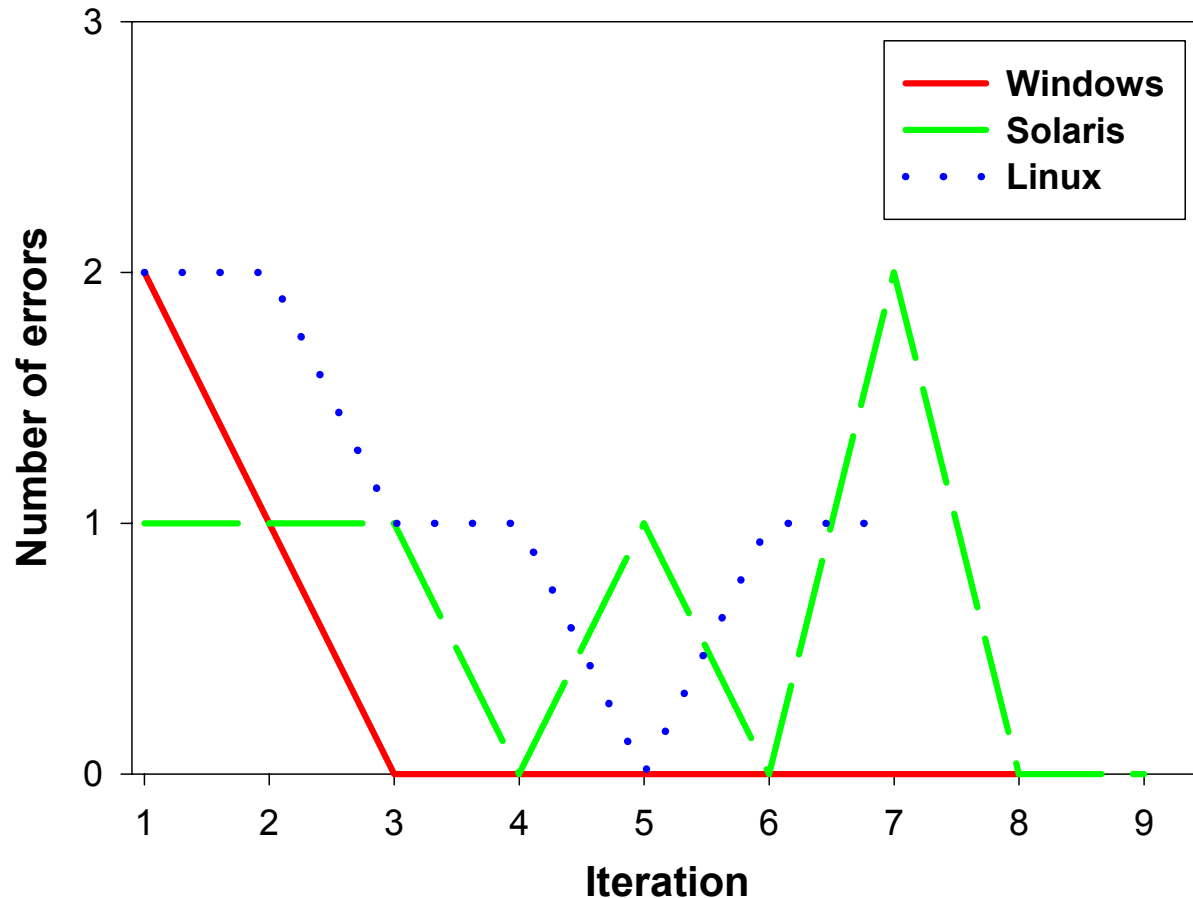


- At minimum, committed to:
 - stage 1 (undo) and stage 3 (exercising repair)
 - a partial implementation of stage 2 (online verification)
 - failure data collection
 - availability benchmarking using human trials

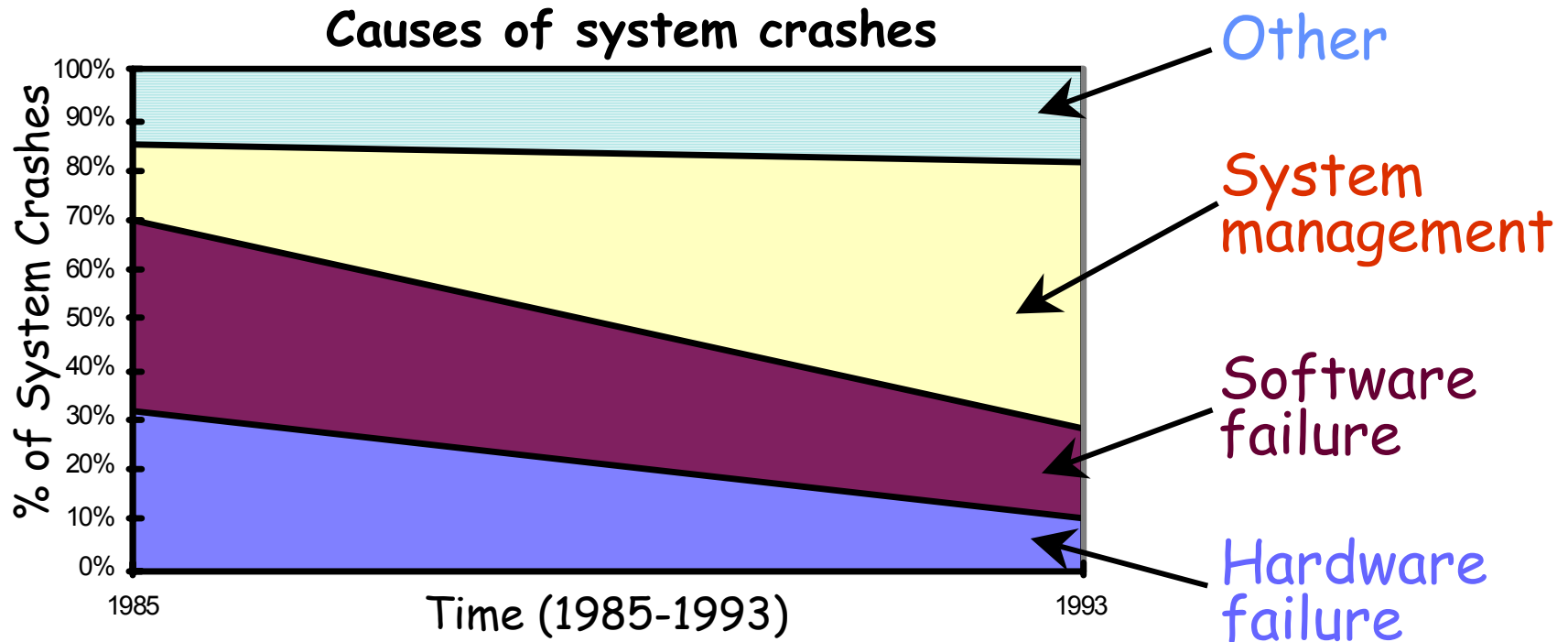
End

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

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

Details: application spectrum

Application	Hard state	Consistency requirement	Interface complexity	Internal knowledge of data semantics	Query complexity	Total
SQL database	3	3	3	3	3	15
E-commerce app. server	0	3	3	3	3	12
Email	3	1	1	2	2	9
File server	3	2	1	1	1	8
Search engine	1	1	0	3	2	7
Block server	3	2	0	0	0	5
Transforming proxy	0	0	0	3	1	4
Web server	1	1	0	1	0	3

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, . . .

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

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**

Context: diagnosis

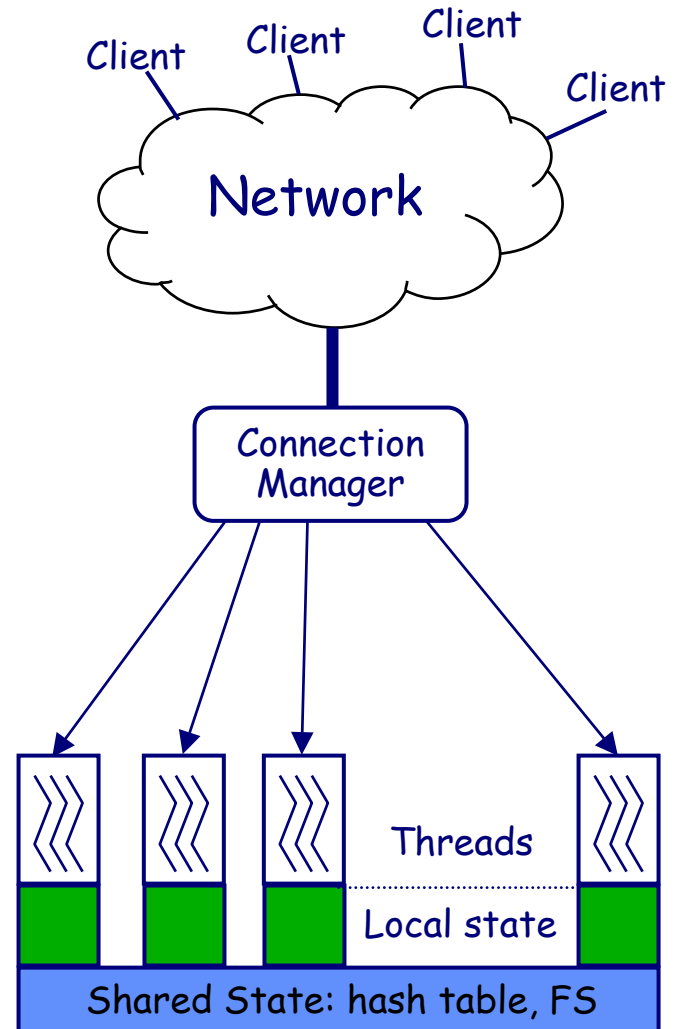
- **One-off system-specific diagnosis aids**
 - NetApp network diagnoser: cross-layer correlation and expert-system approaches
- **General diagnostic methods**
 - expert systems and fault-tree approaches
 - » all require good understanding/model of failure modes, and thus conflict with real-world observations
 - dependency-based root-cause analysis
 - » requires system model, but only at level of resource dependencies
 - » our request-tracing approach dynamically discovers resource dependency model

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