RECOVERY-ORIENTED COMPUTING

# Why Recovery Should Be Free, And Often Can Be

Armando Fox, Stanford University (fox@cs.stanford.edu)

Joint work with George Candea, Andrew Huang, Ben Ling, Emre Kiciman, Stanford University and Dave Patterson, Mike Jordan, Randy Katz, et al., UC Berkeley

# History: Recovery-Oriented Computing

- Joint project between Stanford (Fox) and UC Berkeley (Patterson)
- ROC philosophy ("Peres's Law"):

  *"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"*

  Israeli foreign minister Shimon Peres

  - Failures (hardware, software, operator-induced) are a fact; recovery is how we cope with them over time
  - Availability = MTTF/MTBF= MTTF / (MTTF + MTTR) - rather than just making MTTF very large, make MTTR << MTTF
- Major research areas
  - Fast, generic failure detection and diagnosis
  - Fast recovery techniques and design-for-recovery
- If recovery were predictable and fast, it would simplify both *failure detection* and *recovery management.*

# Statistical Analysis will Save the World

- We have an unprecedented opportunity to collect and analyze data on running systems
- These systems' workloads lend themselves well to statistical analysis
- Statistical learning theory and machine learning techniques (SLT/ML) can help make sense of this data and *spot anomalies* that may indicate failures or impending failures
- Reacting to such detection can automate many aspects of online operations

***Observe - Analyze - Act***

# Outline of Talk

- ***Observe:*** Salient characteristics of today's systems
  - The promise of middleware
  - Laws of large numbers
- ***Analyze:*** Examples: How SLT can help
  - Using SLT for bug finding, performance fault detection, etc.
- ***Act:*** crash-only systems make false positives irrelevant
  - Combining *crash-only software* with SLT
- A general architecture for pervasive SLT/ML integration
  - Architectural challenges
  - Agenda

# Observe: Middleware & data collection

- Component frameworks allow for non-intrusive data collection without modifying the applications
  - Inter-EJB calls through runtime-managed level of indirection
  - Slightly coarser grain of analysis: restrictions on "legal" paths make it more likely we can spot anomalies
- Virtual machine monitors provide additional observation points
  - Already used by ASP's, for load balancing, app migration, etc.
  - Transparent to applications *and hosted OS's*

We can collect lots of data without changes to applications, especially if they are "framework-intensive"

# Observe: Workload

| Observation | Consequence |
|---|---|
| Internet service workloads consist of large numbers of independent users | Large number of independent samples gives basis for success of statistical techniques |
| Even a flaky service is doing mostly the right thing most of the time | Steady-state behavior can be extracted from normal operation |
| Heavy traffic volume means most of the service is exercised in a relatively short time | Baseline model can be learned rapidly and updated in place periodically |

- Internet service workloads are a great match for SLT/ML
- We can continuously extract baseline models from system itself, rather than building them *a priori*

# Analyze: Anomaly Detection and Bugs

- Example: distributed assertion sampling [Liblit et al, 2003]
  - Instrument source code with assertions on pairs of variables ("features")
  - Use sampling so that any given run of program exercises only a few assertions (to limit performance impact)
  - Use classification algorithm to identify which features are most predictive of faults
  - Found source code bugs in *bc*, other programs now being instrumented

SLT is a toolbox of techniques to examine large volumes of data and determine which features are most "interesting"

# Act: Recovery Management

- So what happens when we detect an anomaly?
- Think of recovery as actuating “control points”--must be:
  - Safe - doesn’t cause incorrect application behavior
  - Predictable - cost of actuating control point must be well-known
  - Non-disruptive - doesn’t significantly impact online performance (as long as we don’t do it too often)
- Various existing systems try to achieve these via combination of isolation and redundancy/failover

*These properties are especially important because of false positives.*

# False Positives

- Statistical techniques inevitably have nonzero false positive rates
  - Both "algorithmic" and "semantic" false positives
  - Some algorithms trade false positive rate for detection rate
- Our approach: make false positives irrelevant
  - Make control points so inexpensive to actuate that occasional mistakes don't matter
  - Hint: think of "rolling reboots" as a degenerate case of this
- Result: think in terms of *adaptation,* not recovery.

Challenge: how to design software whose control points are safe, predictable and non-intrusive?

# Crash-Only Software: Simplifying Recovery Management

- Transactions (analogy): provides easy-to-understand invariants that simplify programming (of data-centric apps)
- Crash-only design: provides easy-to-understand invariants that simplify failure detection and recovery management
- A crash-only component provides PWR switch: stop = crash
  - clean shutdown = loss of power = kernel panic = ...
- One way to go down = one way to come up: start = recover
- "Power switch" is external to component => uniform behavior
  - kill -9, "turning off" (process kill) a VM, pull power cord
  - Intuition: the "infrastructure" supporting the power switch is usually simpler than the applications using it, and common across all those applications

If recovery is cheap and predictable, can use "dumber" *(therefore more predictable)* recovery strategy

# Rest of talk

- Three crash-only building blocks
- Combination of SLT algorithms with crash-onlyness to obtain a degree of self-management
- Plans for generalization, research challenges, etc.

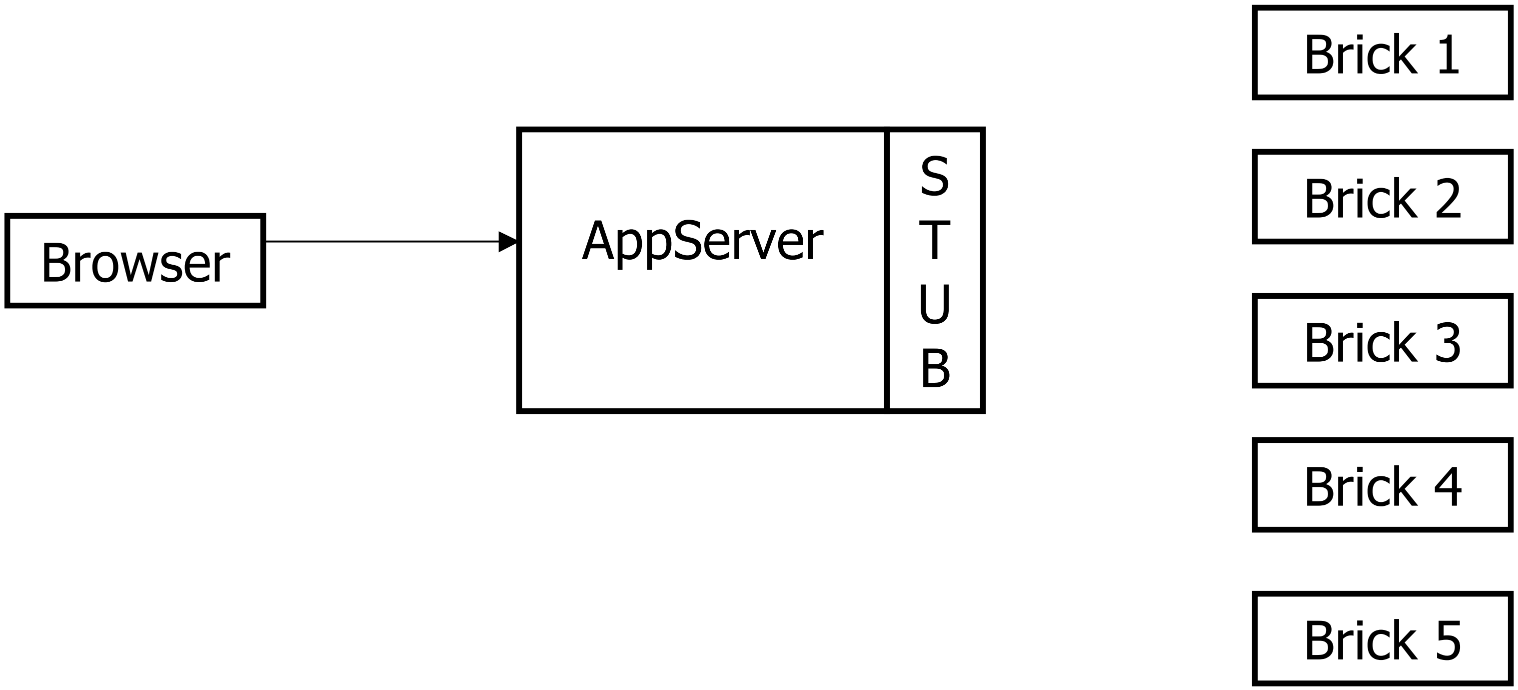
# Crash-Only Building Blocks

| Subsystem | Control point | How realized | Statistical monitoring |
|---|---|---|---|
| SSM (diskless session state store) [NSDI 04] | Whole-node reboot | Quorum-like redundancy; relaxed consistency | Time series of state and activity (Tarzan) |
| DStore (persistent hashtable) [in preparation] | Whole-node reboot | Quorum-like redundancy and predictable repair; relaxed consistency | Time series of state and activity (Tarzan) |
| JAGR (J2EE application server) [AMS 2003 & in prep.] | Microreboots of EJB's | Modify appserver to undeploy/ redeploy EJB's and stall pending reqs | Pinpoint: Anomalous code paths and component interactions (Probabilistic context-free grammar) |

- Control points are safe, predictable, non-disruptive
- Crash-only design: shutdown=crash, recover=restart
- Makes state-management subsystems as easy to manage as stateless Web servers

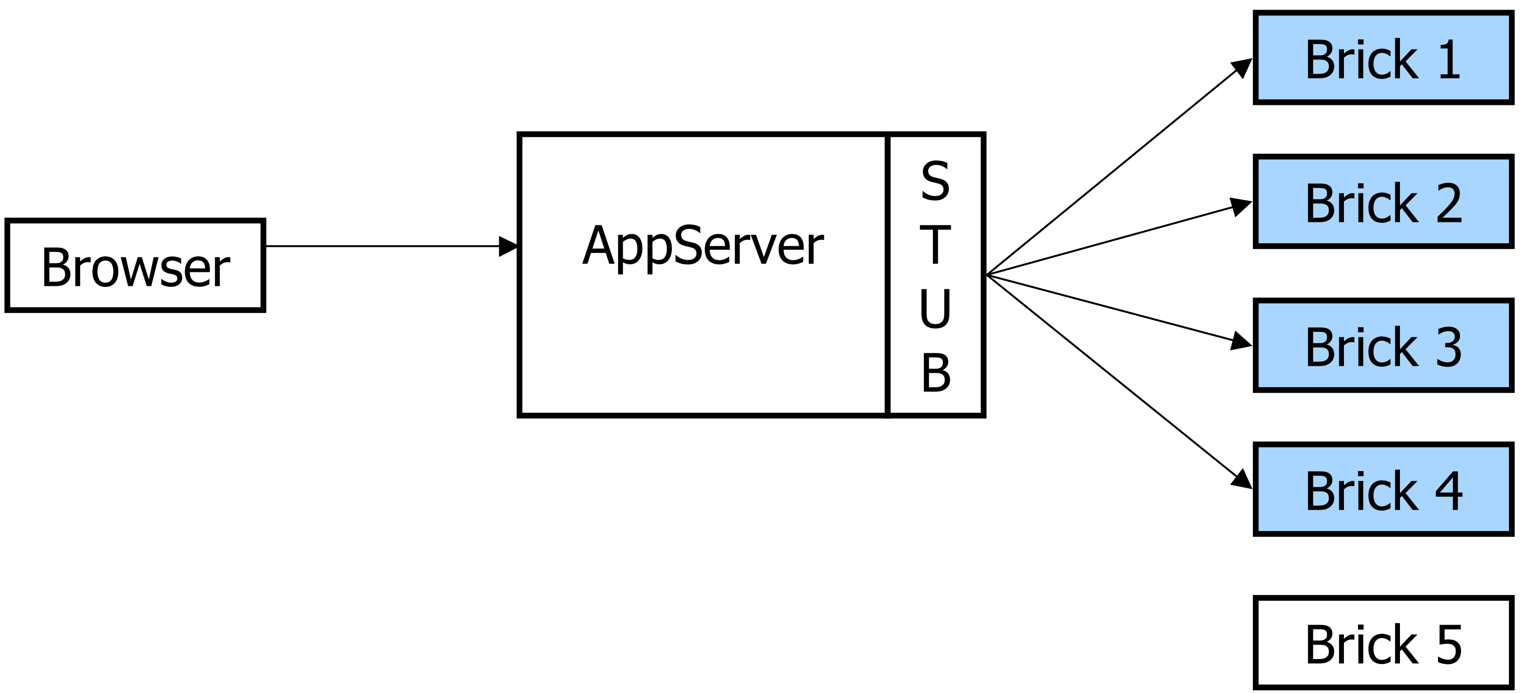
# SSM Write example: "Write to Many, Wait for Few"

Try to write to W bricks, W = 4
Must wait for WQ bricks to reply, WQ = 2



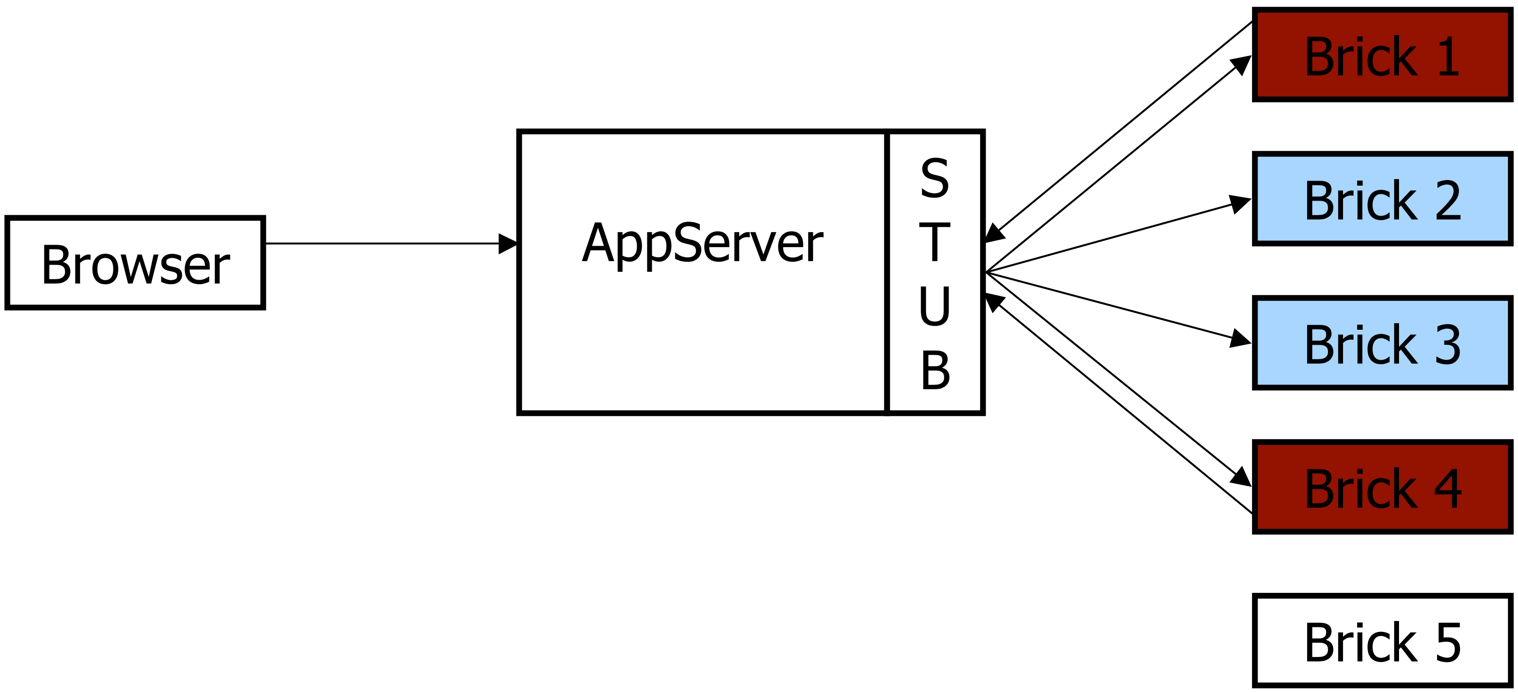
# Write example: "Write to Many, Wait for Few"

Try to write to W bricks, W = 4
Must wait for WQ bricks to reply, WQ = 2



# Write example: “Write to Many, Wait for Few”

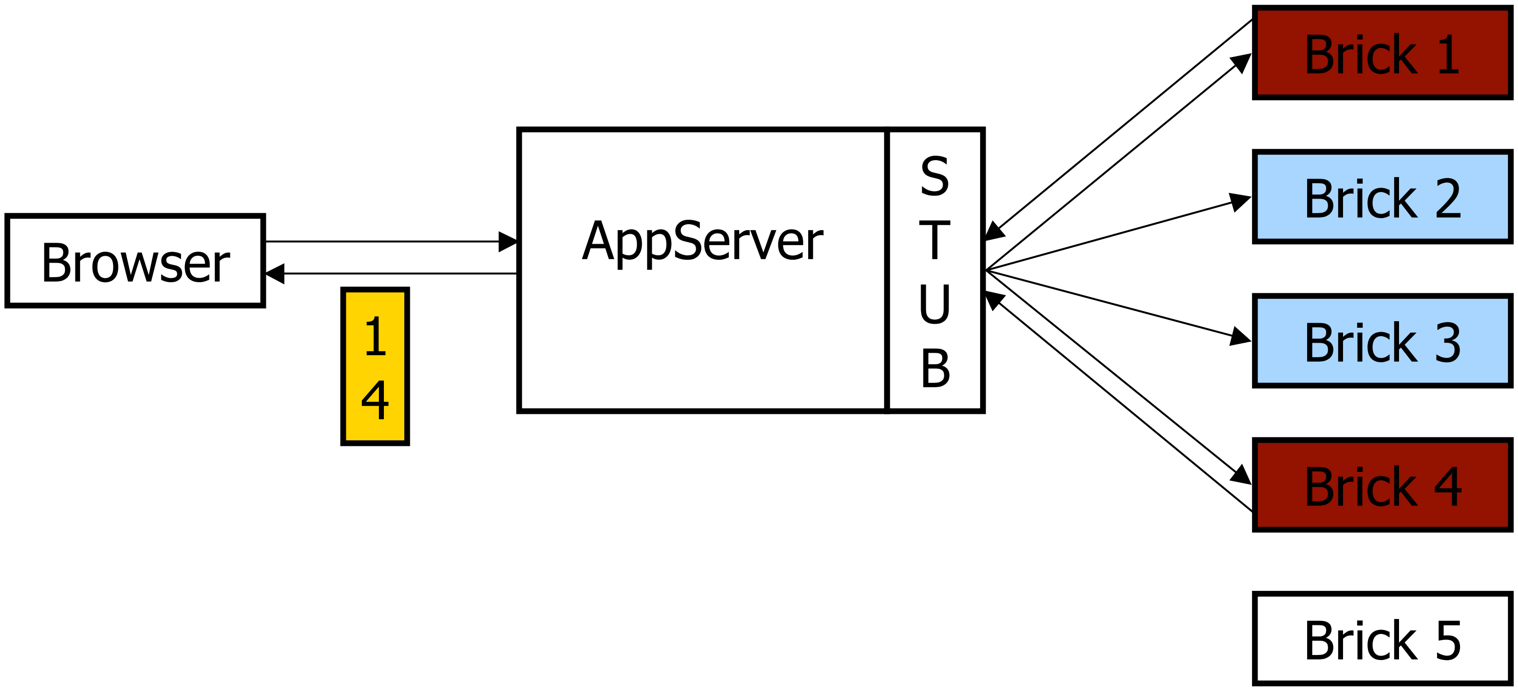
Try to write to W bricks, W = 4
Must wait for WQ bricks to reply, WQ = 2



# Write example: “Write to Many, Wait for Few”

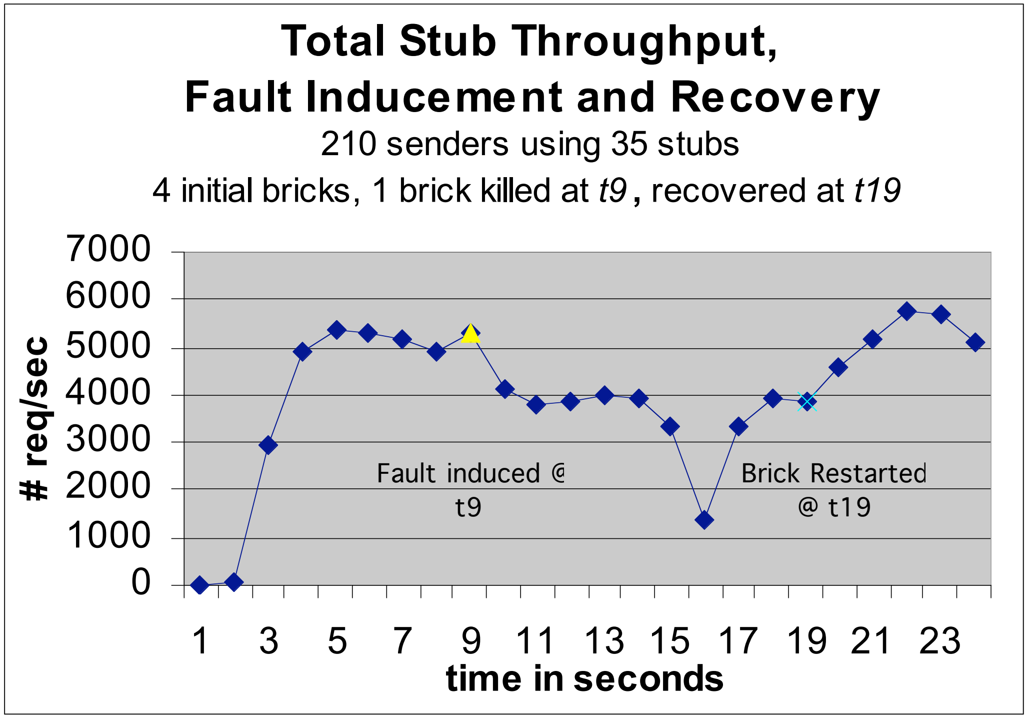
Try to write to W random bricks, W = 4
Must wait for WQ bricks to reply, WQ = 2

Can tolerate **WQ-1** failures before data loss



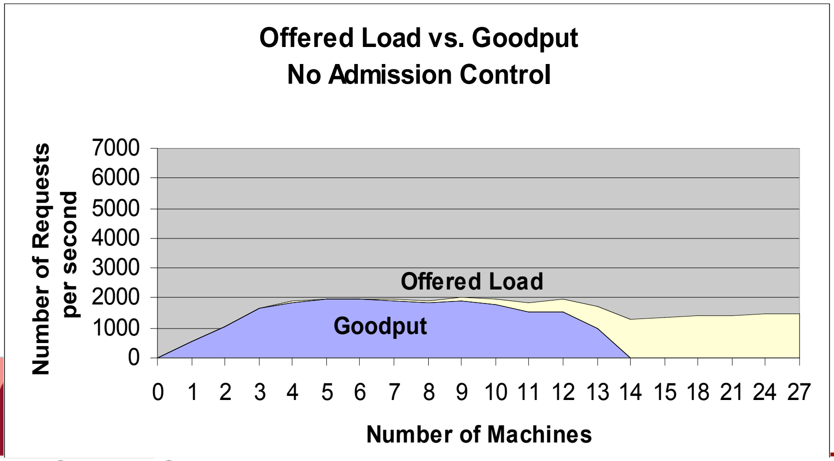
# Fault-injection and recovery with 4 bricks

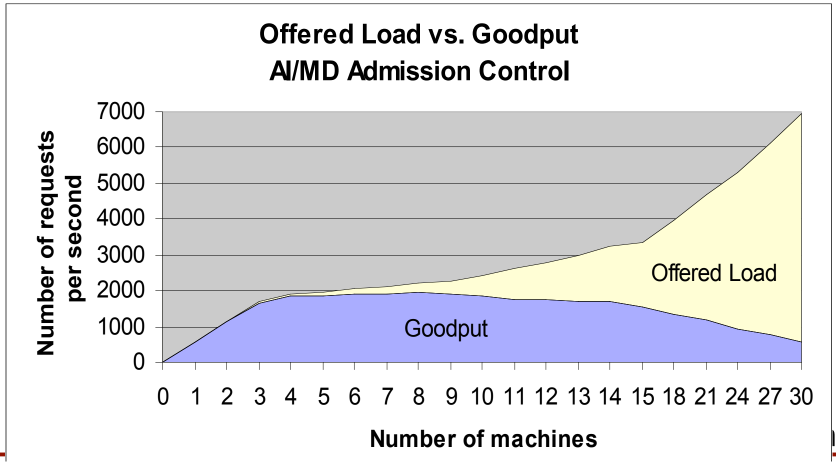
- After fault, steady-state throughput drops *(but doesn't fall off saturation cliff)*, and recovers smoothly when new brick is added
  - Dip at t16 is caused by JVM heap size increase



# Predictability Through Backpressure

- Self-tuning and backpressure to shed excess load
  - TCP-like "windowing" mechanism at stubs maintained per-brick lets system discover its maximum per-brick capacity
  - backpressure (early reject) to shed excess load and avoid saturation cliff
  - Gives operator a margin of error to add more resources while maintaining *highly predictable* response time
- New bricks automatically absorbed and load is eventually redistributed
- Experiments with N=3, W=3, WQ=2
  - Note - "# of hyperactive users" != "# reqs per unit time"



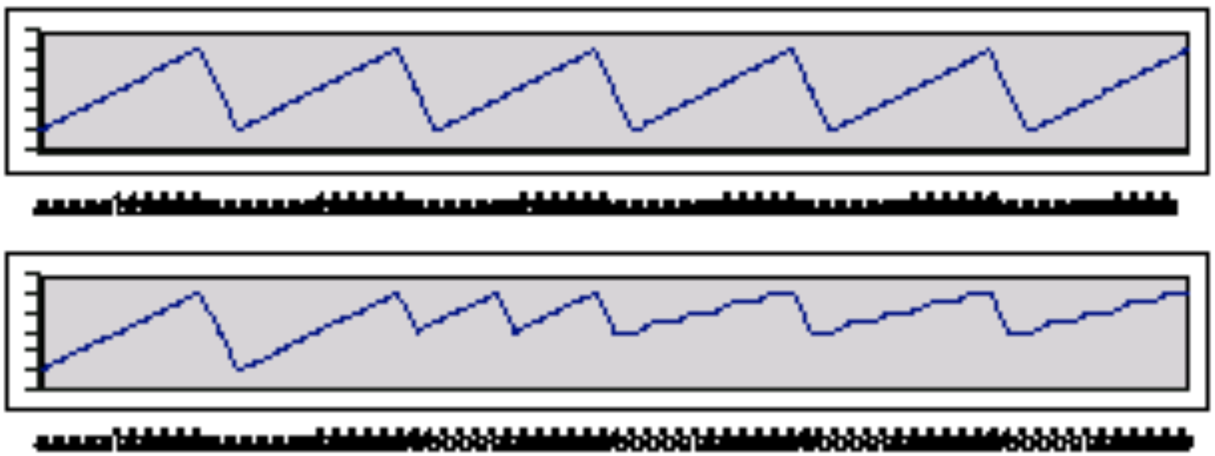
# Detecting “Anomalous” Conditions

- 9 metrics collected per brick every second
  - NumDropped, NumWriteProcessed, NumReadProcessed, TimeInterval, FreeMemory, NumElements, InboxSize, NumRequestsHandled, MemoryUsed
  - “Activity” statistics capture a notion of “forward progress”
  - “State” statistics capture resource utilization under “normal” circumstances
- Metrics compared against those of “peer” bricks
  - Basic idea: Changes in workload tend to affect all bricks equally
  - Underlying (weak) assumption: “Most bricks are doing mostly the right thing most of the time”
  - Anomaly in 6 or more (out of 9) metrics => reboot brick

# Detecting anomalies, cont.

- "Activity" statistics compared against other bricks (absolute median deviation)
- "State" statistics use simple time-series analysis (Tarzan)
  - keep N-length time series, discretize each data point
  - count relative frequencies of all substrings of length *k* or shorter
  - Works even when period is irregular or not known *a priori*

- Note! We are not SLT/ML researchers!
- Goal is to *enable* those techniques to be brought to bear



# What faults does this handle?

- Substantially all non-Byzantine faults we injected:
  - Memory bitflips in code, data, and checksums (=> crash)
  - hang/timeout/freeze
  - Network loss (drop up to 70% of packets randomly)
  - Periodic slowdown (eg from garbage collection)
  - Persistent slowdown (one node lags the others)
- Intuition: the metrics capture some notion of *forward progress and satisfactory progress* (relative to peers)
- All anomalies are "coerced" to crash faults
  - If that turned out to be the wrong thing, it didn't cost you much to try it
  - Human notified after threshold number of restarts

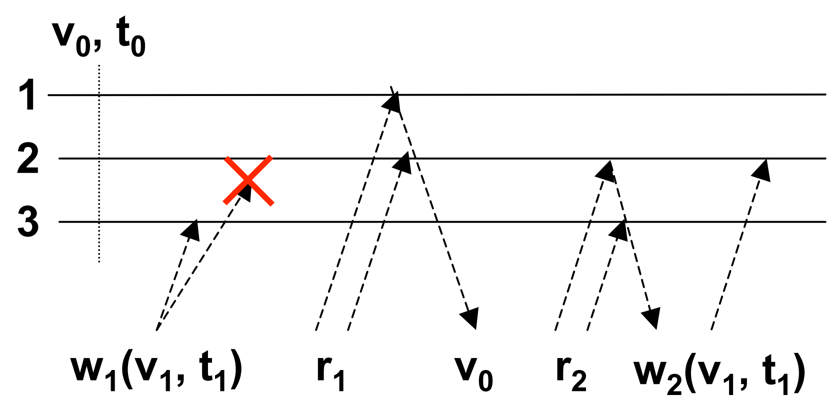
This system is "always recovering" -- by adapting

# DStore: Crash-only Single Key Persistent Store

- For single-key/single-user data (e.g. profiles), make persistence layer as easy to manage as stateless.
  - SSM relies on frequent refresh; doesn't work for persistent state
  - DStore relies on quorums and uses single-phase operations
  - API: hash table with put(), get(), delete(); no partial updates
  - Used for Yahoo! user database, Amazon merchandise catalog, many others
- Write to majority, read from majority
  - On read, if timestamps differ, writeback later timestamp to a majority
  - "Delayed-commit" semantics possible if node failure happens, but linearizable schedule is guaranteed

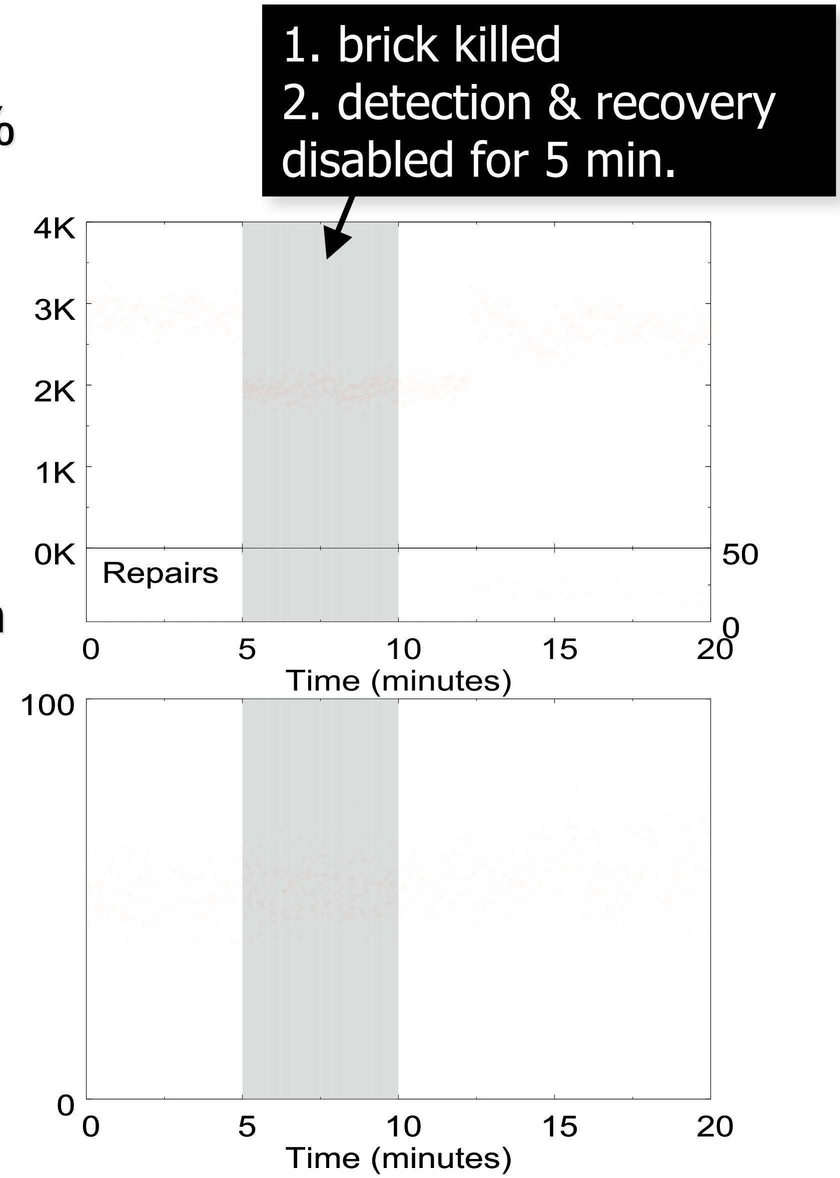
# DStore quorum algorithm

- Basic quorum algorithm
  - Write: broadcast to all, wait for a majority to respond
  - Read: read value from one, read timestamp from majority-1
- Partial writes: coordinator failure (no 2-phase commit)
- Repair: $r_2$ returns $v_1$
  - Reads issued prior to $r_2$ return v no newer than $v_1$
  - Reads issued after $r_2$ return v no older than $v_1$
  - Linearizability for fail-stop



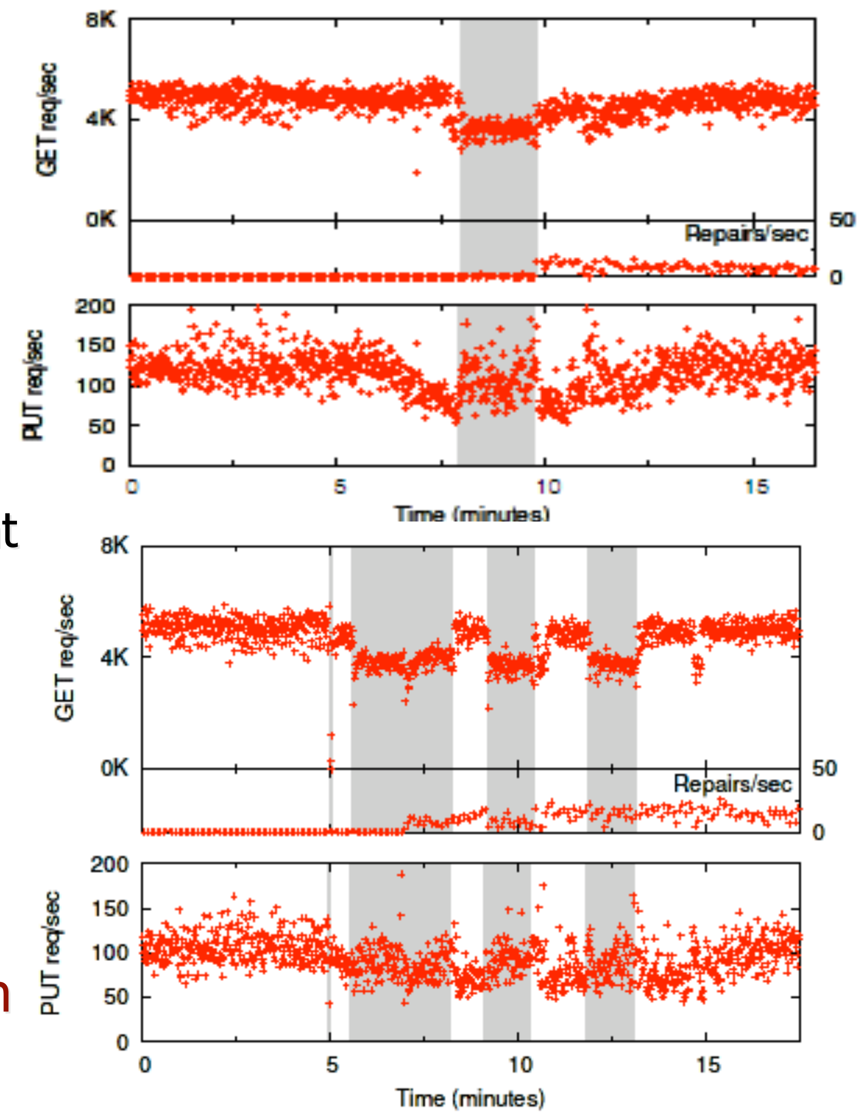
# Results: Fast, non-intrusive recovery

- 3 bricks, 90/10 read/write mix, 85% timestamp cache hit rate
  - Common-case performance comparable to ROWA schemes
- Rebooting a node is...
  - Safe - due to replication
  - Predictable - throughput restored in <1 min. after reboot
  - Non-disruptive - Data available for both GETs and PUTs throughout



# Automatic Detection & Recovery

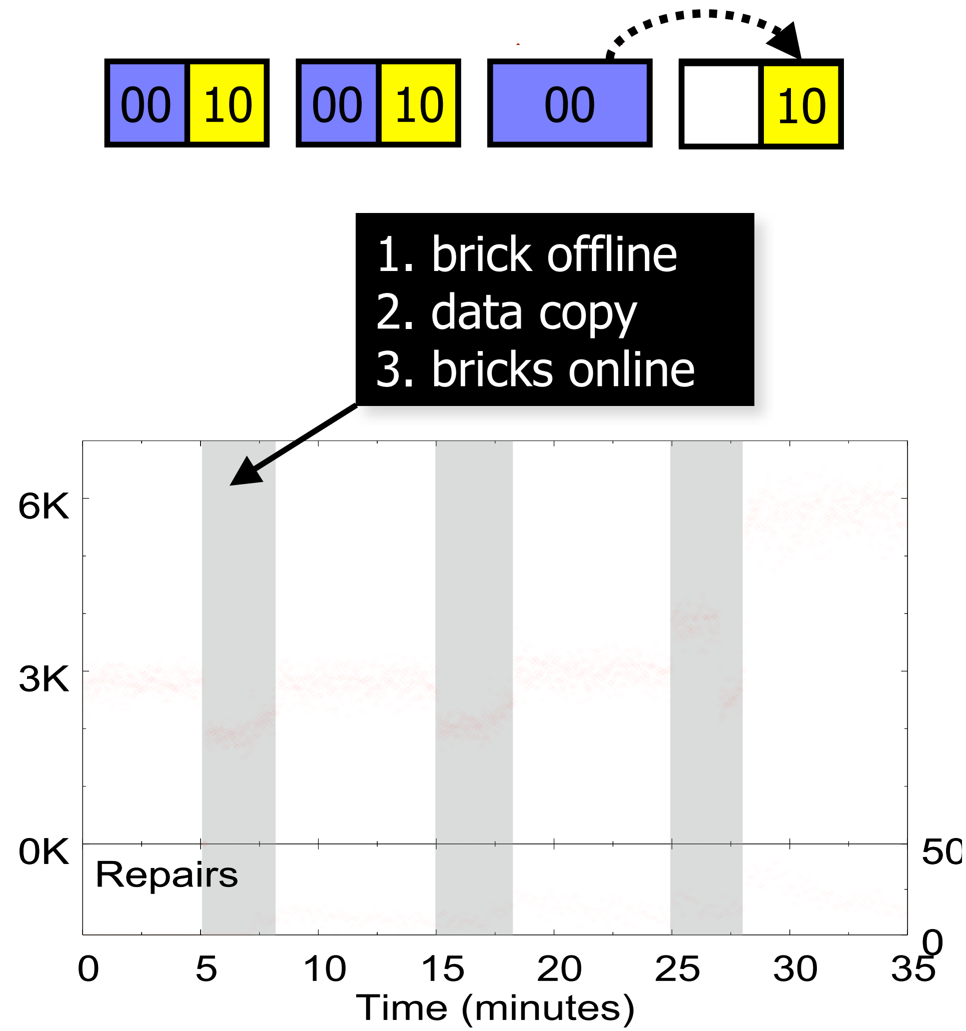
- Metrics and algorithm comparable to those used in SSM
- We inject "fail-stutter" behavior by increasing request latency
  - Top: threshold=8, anomaly caught later
  - Bottom: threshold=5, anomaly caught earlier
  - Earlier detection also results in 2 "unnecessary" reboots
  - But they don't matter much




Illustrates trade-off of fast detection vs. false positives

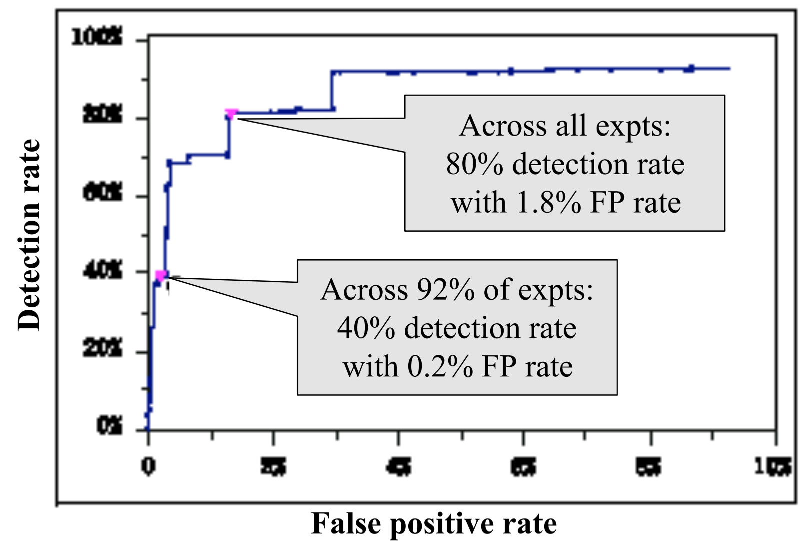
# Casting repartitioning as recovery

- Split replica group ID (rgid), but announce both
- Existing repair mechanisms used for "recovery"
- Automatic detection of which rgid to split
- Example: growing from 3 to 6 bricks

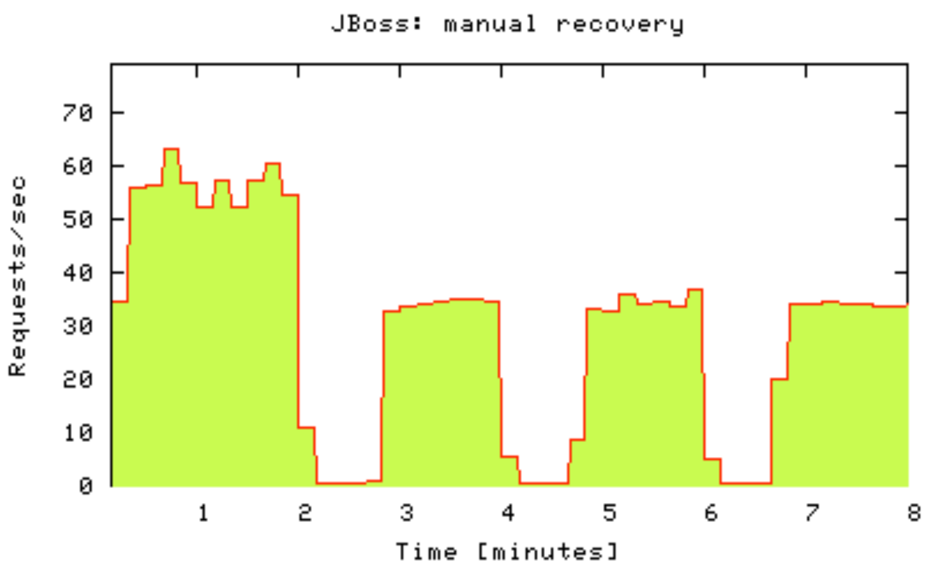


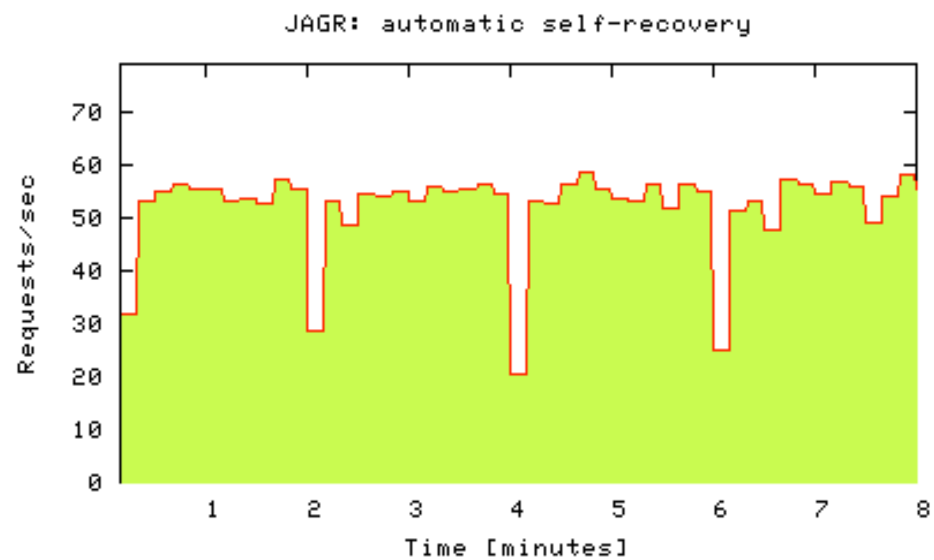
# Pinpoint: Anomalous Path Detection

- Capture paths through EJB's as dynamic call trees (intra-method calls hidden)
- Build probabilistic context-free grammar from these
- Detect trees that correspond to very low probability parses
  - Component interaction analysis currently finds 55-75% of failures.
  - Path shape analysis detects >90% of failures; but correctly diagnoses fewer.
  - Shared-data analysis pending



# JAGR: JBoss with Micro-reboots

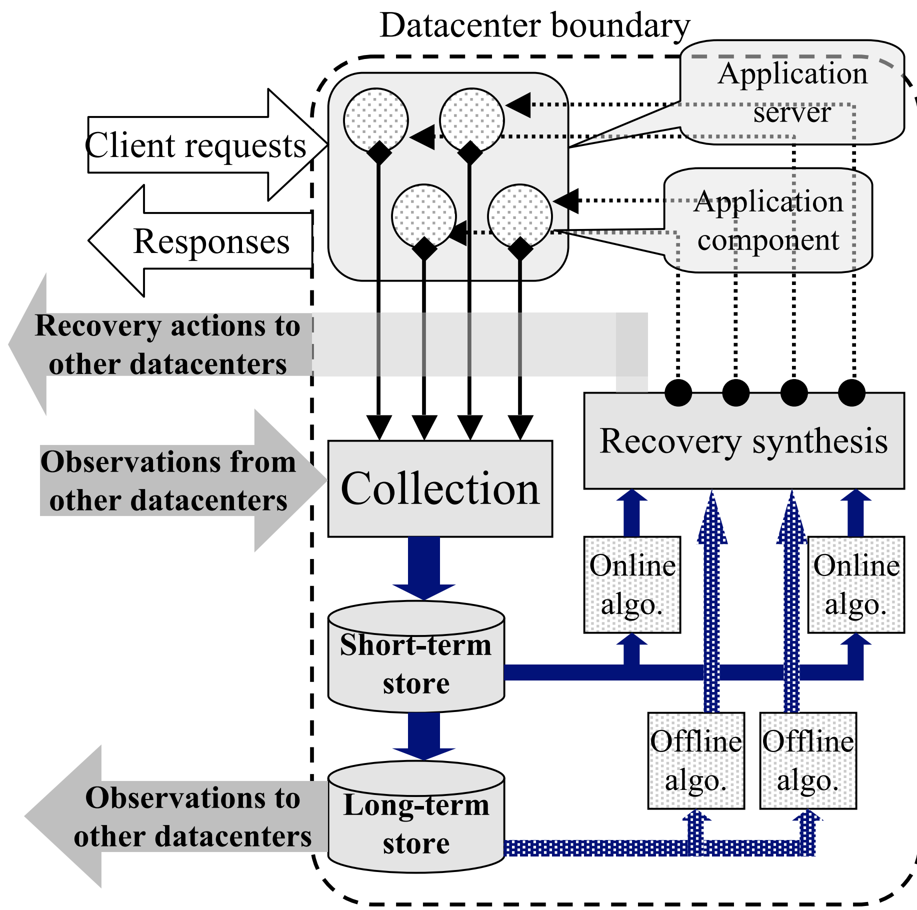



- performability of RUBiS (goodput/sec vs. time)
  - vanilla JBoss w/manual restarting of app-server, vs. JAGR w/automatic recovery and micro-rebooting
  - JAGR/RUBiS does 78% better than JBoss/RUBiS
  - Maintains 20 req/sec, even in the face of faults
  - Lower steady-state after recovery in first graph: class reloading, recompiling, etc., which is not necessary with micro-reboots

# A General Architecture for SLT/ML

- Challenges:
  - SLT algorithms must be *integrated* and *online*
  - Data collection without perturbing system
  - Data storage and management for models
  - Wily attackers who can game the algorithms
  - Multi-level learning and multi-timescale learning
  - Much more



# Ongoing Work

- Complete Pinpoint and JAGR, and integrate these
  - Pinpoint being deployed now at Amazon.com
- Benchmark JAGR+SSM running unmodified J2EE apps
  - To be submitted to OSDI'04
- Broader research program: RADS (Reliable Adaptive Distributed Systems), jointly with UC Berkeley
  - Aggressive application of SLT/ML
  - Includes lower layers: programmable network elements at edge networks, wide-area resilient routing protocols, generic software architecture for data collection and SLT/ML application
  - NSF proposal being submitted next week

# Summary

- Statistical analysis is a toolbox of powerful techniques for anomaly/novelty detection, classification, etc.
  - Time is ripe to bring these to bear on dependable computing
- Crash-only design can make cost of false-positives sufficiently low that we can simply tolerate them
- Crash-only design makes recovery predictable by controlling it using dead-simple mechanisms
- Many technologies and trends already in place to generalize this approach
  - Middleware-intensive apps, Virtual Machines, ...

# Backup Slides

# Crash-Only Design Lessons from SSM

- Eliminate coupling
  - No dependence on any specific brick, just on a subset of minimum size -- even at the granularity of individual requests
  - Not even across phases of an operation: single-phase nonblocking ops only => predictable amount of work/request
  - Use randomness to avoid deterministic worst cases and hotspots
  - We initially violated this guideline by using an off-the-shelf JMS implementation that was centralized
- Make parts interchangeable
  - Any replica in a write-set is as good as any other
  - Unlike erasure coding, only need 1 replica to survive
  - Cost is higher storage overhead, but we're willing to pay that to get the self-* properties

# Design Lessons, cont.

- It's OK to say no: use backpressure and AIMD to limit load, and don't make promises you can't keep
  - Initially violated this too: blocking implementation of NetworkWrite() would cause lock starvation when SAN failure was injected
- It's OK to make mistakes
  - Enables future use of aggressive statistical monitoring techniques
  - Potentially allows a large body of statistical process control and machine learning to be brought to bear on this problem

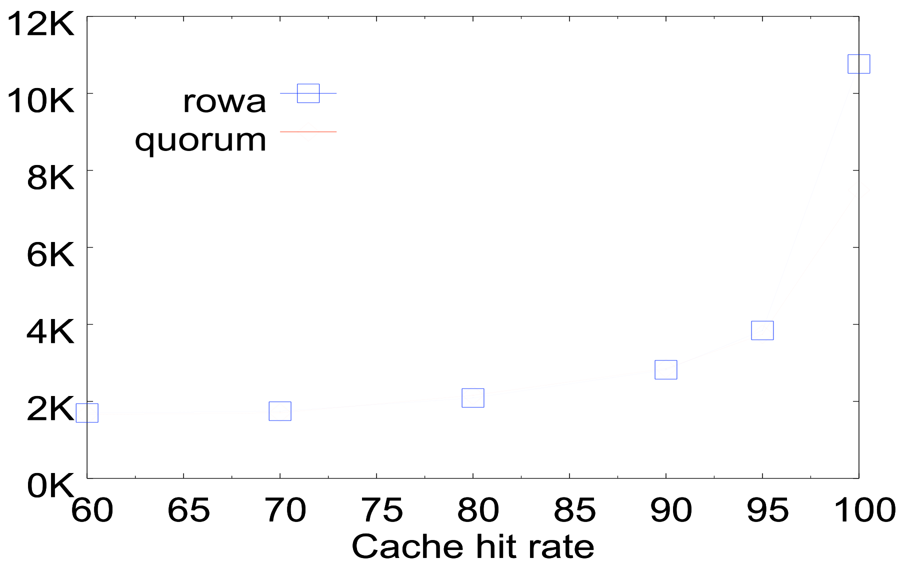
# Design lessons, cont.

- For storage nodes… "Be independent"
  - A storage node shouldn't be dependent on other storage nodes to service a request
  - Anti-examples: primary-secondary replication, multi-database-node join
  - In practice: expose a simple hash table API to reduce data dependencies
  - Avoid single operations that lead to torrents of new work: use lazy repair to fix inconsistencies as they are found
- For clients… "Don't be picky"
  - A client shouldn't rely on any <u>specific</u> node to be up
  - Anti-examples: ROWA, 2-phase commit
  - In practice: use quorums to tolerate internal inconsistency among replicas

# DStore: Read timestamp overhead

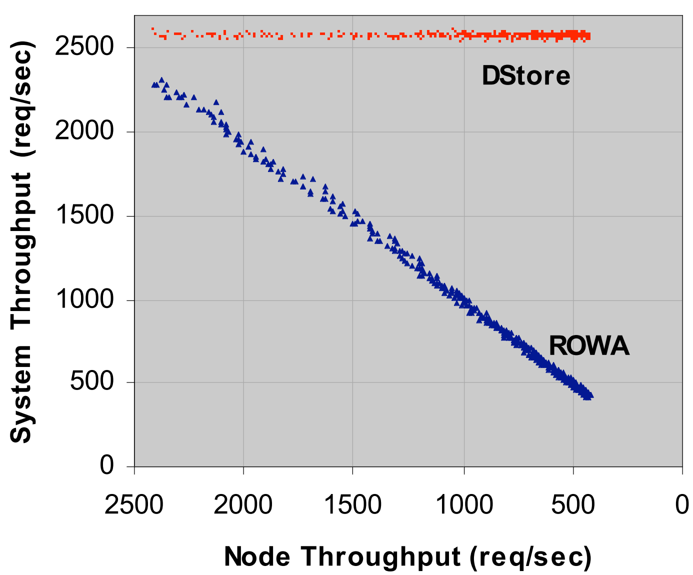
- Benchmark details:
  - 3 bricks, 3 GET clients
  - read_ts optimization: read value from 1 brick, timestamp from 1 brick
- Summary:
  - Disk is bottleneck, so reading a timestamp (pinned in memory) adds little overhead

→ Common-case performance comparable to ROWA



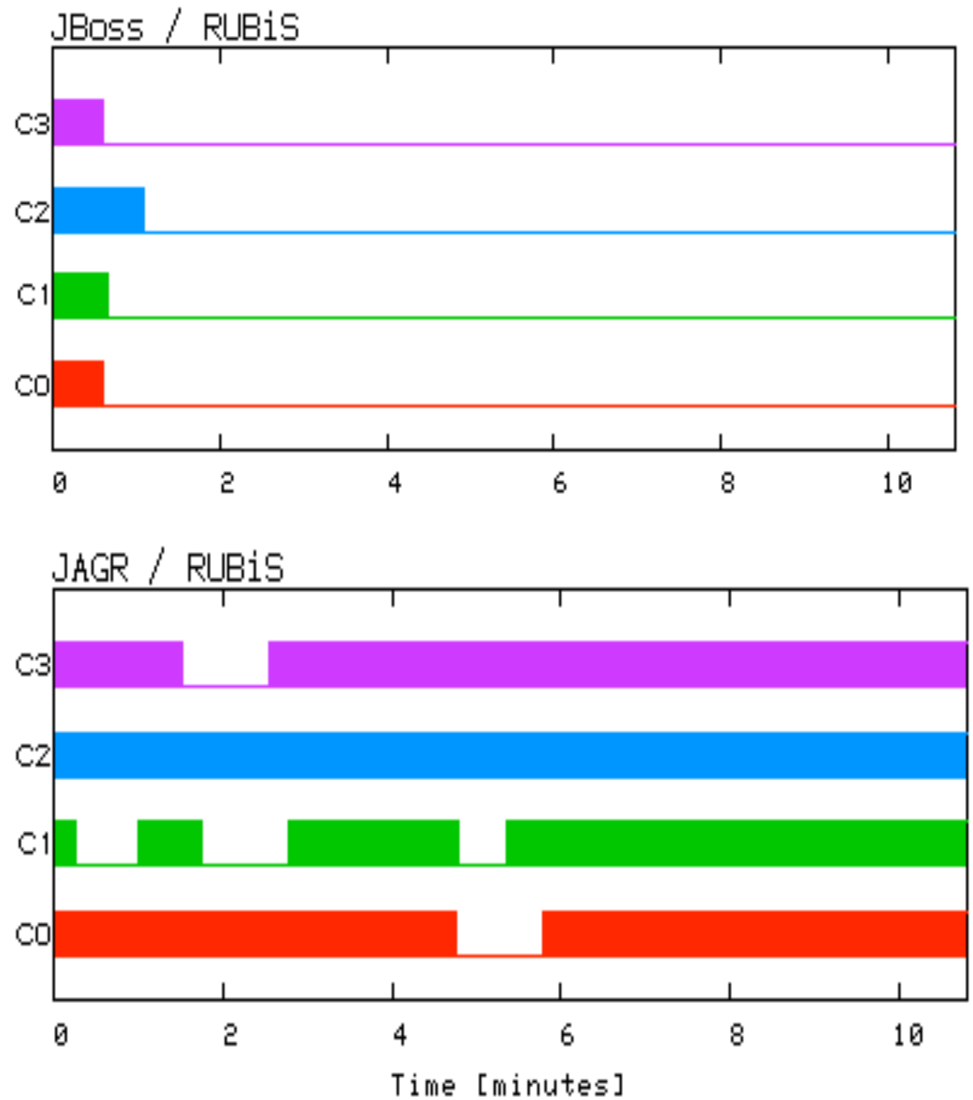
# DStore R/W mix microbenchmarks

- After failure, thruput restored in seconds
- Throttling one brick doesn't bottleneck the system
- Online repartitioning: "fail"  a brick, copy it, reintegrate both
- In all cases, data available for *both reads & writes* throughout



# JAGR: Recovery microbenchmarks

- RUBiS
  - E-Bay-like app
  - Has many naturally occurring faults
- Running on vanilla JBoss gives poor availability
  - 4 concurrent clients causes deadlock
- JAGR automatically recovers every time



# JAGR: Modifing JBoss