Combining Statistical Monitoring and Predictable Recovery for Self-Management

Armando Fox, Emre Kiciman, Stanford University

Dave Patterson, Mike Jordan, Randy Katz, UC Berkeley

WOSS 2004 Workshop, Newport Beach, CA
Disclaimer & background

- I don’t do research in software engineering
- I am not a “software architect”
- But I have had both titles

Motivation...

- Systems background, focusing on Internet servers
- Known & stable environment & specs => Not
  - Heisenbugs, race conditions, environment-dependent and hard-to-reproduce bugs still account for majority of SW bugs in live systems
  - but often difficult to detect w/o specialized checks
Transient failures in middleware-intensive apps

- Pain: up to 80% of bugs found in production are those for which a fix is not yet available*
  - most are “Heisenbugs”, up to 60% are reboot-curable

- Good news for middleware-intensive apps
  - Modular app structure allows *localized recovery*
  - We know where the state is (in J2EE anyway): app writer API’s make application-level “checkpoint” explicit, shared state awkward to express
  - Highly regular app behavior: updates to semi-persistent state framed by short sequences of EJB calls; occasional update to persistent state

- Can we exploit these observations to do transient-failure *repair* without the understanding needed for *diagnosis*?

Approach: machine learning + microrecovery

1. Exploit modular app structure, workload, & rich middleware platform to build statistical models of app behavior
   - eg, anomalies in component-level code paths may suggest an application-level failure

2. Use low-cost, usually-helpful, guaranteed-to-do-no-harm “microrecovery” mechanisms to react to anomalies
   - *Makes inevitable false positives tolerable*

- A new way to think about managing a running system
  - Invariant: always safe to try microrecovery first (even if more expensive recovery eventually required)
  - “Always adapting, always recovering”
Example SLT method: Path shape analysis

- Model paths as parse trees in probabilistic CFG
  - Build grammar under “believed normal” conditions, then mark very unlikely paths as anomalous
  - after classification, build decision tree to localize anomalies in path

- Correlation != diagnosis != localization, but often, localization ⇒ recovery

- Detection: 89-96% of injected failures, vs. 20-79% for existing application-generic methods

- Localization: tradeoff of recall vs. precision (1 - false positive rate)
  - From [R]=.68, [P]=.14 to [R]=.34, [P]=.93
  - *Cheap recovery suggests we trade in favor of better recall (detection)*
Example microrecovery: “microrebooting” EJB’s

- dramatically improves user-perceived availability over full reboot [OSDI 04]

- 89% reduction in failed requests, despite 6 false positives due to crude localization
  - Up to 97% false positives still gives better availability than full reboot

- User’s session state ("checkpoint") copied to a crash-only state store [NSDI 04] to survive microreboot
Testbed: JBoss+uRB’s+SSM+fault injection

Fault injection: null refs, deadlocks/infinite loop, corruption of volatile EJB metadata, resource leaks, Java runtime errors/exc

RUBiS: online auction app
132K items, 1.5M bids, 100K users
150 users (35-45 req/sec) / node
Workload mix based on a commercial auction site
Observations on why to try this

- Both instrumentation and microrecovery can be done in the middleware, supporting existing apps
- Large complex systems tend to exercise a lot of their functionality in a fairly short amount of time
- Even if we don’t know what to measure, statistical and data mining techniques can help figure it out
- Most systems work well most of the time, so anomaly detection is reasonable
- Non-experts (in SLT) can achieve encouraging results even with “simple” algorithms
Non-goals/complementary work

- Byzantine fault tolerance
- In-place repair of persistent data structures
- Hard-real-time response guarantees
- Adding checkpointing to legacy non-componentized applications
- Source code bug finding
- Configuration troubleshooting (but see Wang et al. 2002-2004, Microsoft Research)
- SLT for performance optimization (*but...some performance problems indicate failure masked at lower level*)
- Advancing the state of the art in SLT (analysis algorithms)
Thoughts for discussion

- All problems inherit from the problem of state management (in the sense of “application state”)
  - Most programmers don’t do a good job of unwinding state properly when handling complex exceptions [Weimer 2004]
  - Techniques for recovering/scaling/etc “stateless server farms” rely on invariant: “always safe to reboot”
  - This has caused us to think more clearly about separating out different kinds of state (execution, semipersistent, various flavors of persistent)
  - What other applications could be (re)built this way? We have performance to spare, in most cases

- **Future:** applying linear control theory - “knowing you have it right”
  - Welsh et al. 2001: early example of “reconfiguration” using CT