RAS-Models: A Building Block for Self-Healing Benchmarks

Rean Griffith, Ritika Virmani, Gail Kaiser

Programming Systems Lab (PSL)

Columbia University

PMCCS-8 Edinburgh, Scotland

September 21st 2007

Presented by Rean Griffith

rg2023@cs.columbia.edu

Overview

- Introduction
- Problem
- Hypothesis
- Experiments & Examples
- Proposed Evaluation Methodology
- Conclusion & Future Work

Introduction

- A self-healing system "…automatically detects, diagnoses and repairs localized software and hardware problems" – The Vision of Autonomic Computing 2003 IEEE Computer Society

Why a Self-Healing Benchmark?

- To quantify the impact of faults (problems)
  - Establish a baseline for discussing "improvements"
- To reason about expected benefits for systems currently lacking self-healing mechanisms
  - Includes existing/legacy systems
- To quantify the efficacy of self-healing mechanisms and reason about tradeoffs
- To compare self-healing systems

Problem

- Evaluating self-healing systems and their mechanisms is non-trivial
  - Studying the failure behavior of systems can be difficult
  - Multiple styles of healing to consider (reactive, preventative, proactive)
  - Repairs may fail
  - Partially automated repairs are possible

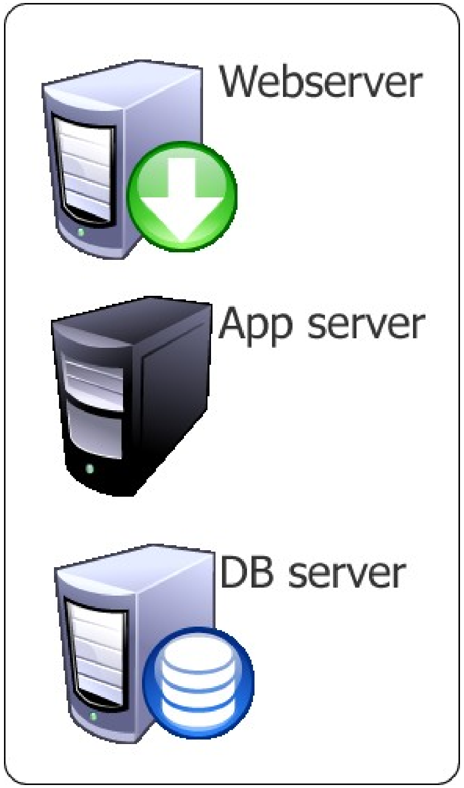
Hypotheses

- Reliability, Availability and Serviceability provide reasonable evaluation metrics
- Combining practical fault-injection tools with mathematical modeling techniques provides the foundation for a feasible and flexible methodology for evaluating and comparing the reliability, availability and serviceability (RAS) characteristics of computing systems

Objective

- To inject faults into the components of the popular n-tier web-application
  - Specifically the application server and Operating System
- Observe its responses and any recovery mechanisms available
- Model and evaluate available mechanisms
- Identify weaknesses

Experiment Setup



Target: 3-Tier Web Application

TPC-W Web-application
Resin 3.0.22 Web-server and (Java) Application Server
Sun Hotspot JVM v1.5
MySQL 5.0.27
Linux 2.4.18

Remote Browser Emulation clients to simulate user loads

Practical Fault-Injection Tools

- Kheiron/JVM
  - Uses bytecode rewriting to inject faults into Java Applications
  - Faults include: memory leaks, hangs, delays etc.
- Nooks Device-Driver Fault-Injection Tools
  - Uses the kernel module interface on Linux (2.4 and now 2.6) to inject device driver faults
  - Faults include: text faults, memory leaks, hangs etc.

Healing Mechanisms Available

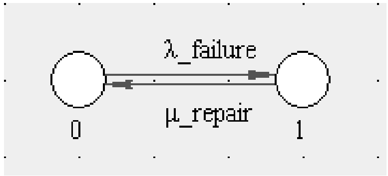
- Application Server
  - Automatic restarts
- Operating System
  - Nooks device driver protection framework
  - Manual system reboot

Mathematical Modeling Techniques

- Continuous Time Markov Chains (CTMCs)
  - Limiting/steady-state availability
  - Yearly downtime
  - Repair success rates (fault-coverage)
  - Repair times
- Markov Reward Networks
  - Downtime costs (time, money, #service visits etc.)
  - SLA penalty-avoidance

Example 1: Resin App Server

- Analyzing perfect recovery e.g. mechanisms addressing resource leaks/fatal crashes
  - S_0 – UP state, system working
  - S_1 – DOWN state, system restarting
  - $\lambda_{failure}$ = 1 every 8 hours
  - $\mu_{restart}$ = 47 seconds
- Attaching a value to each state allows us to evaluate the cost/time impact associated with these failures.

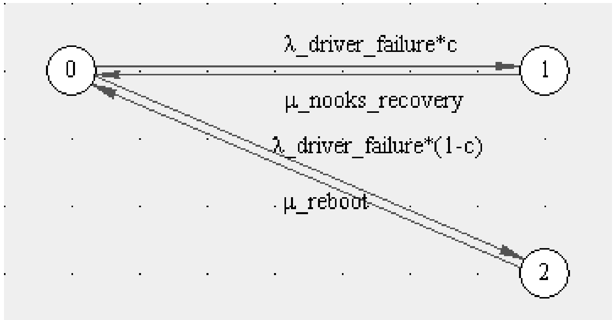


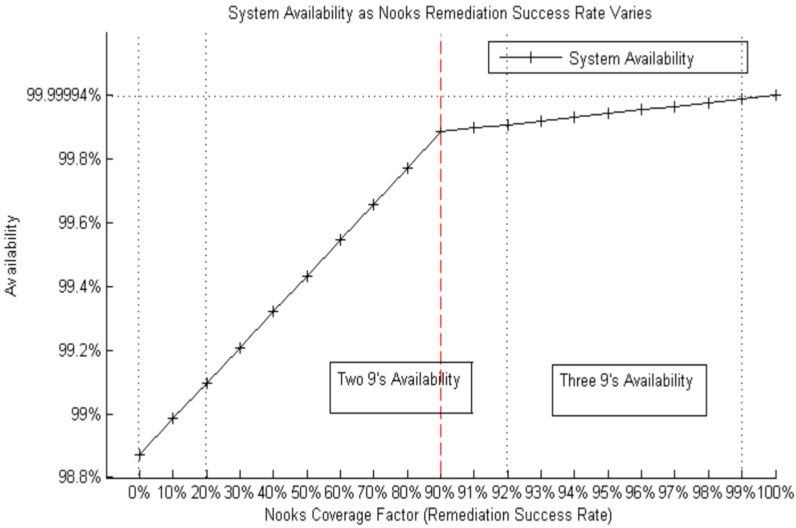
Results:
Steady state availability: 99.838%
Downtime per year: 866 minutes

Availability guarantee	Max downtime per year	Expected penalties
99.999	~5 mins	(866 - 5)*$p
99.99	~53 mins	(866 - 53)*$p
99.9	~526 mins	(866 - 526)*$p
99	~5256 mins	$0

Example 2: Linux w/Nooks

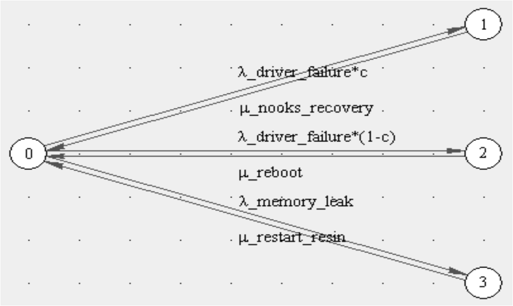
- Analyzing imperfect recovery e.g. device driver recovery using Nooks
  - S_0 – UP state, system working
  - S_1 – UP state, recovering failed driver
  - S_2 – DOWN state, system reboot
  - $\lambda_{driver\_failure}$ = 4 faults every 8 hrs
  - $\mu_{nooks\_recovery}$ = 4,093 mu seconds
  - μ_{reboot} = 82 seconds
  - c – coverage factor/success rate

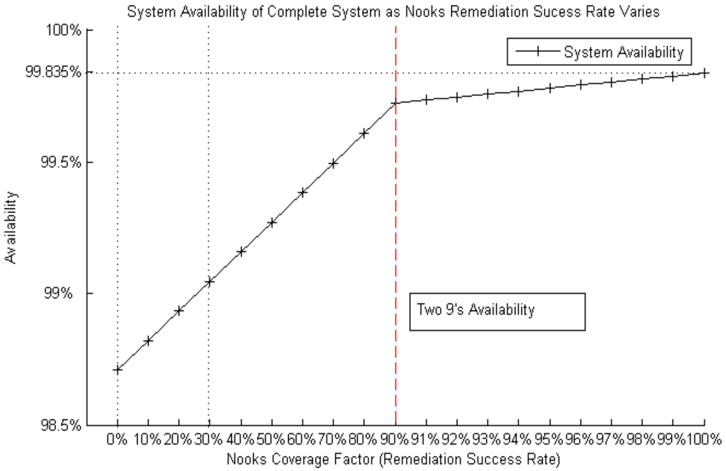




Example 3: Resin + Linux + Nooks

- Composing Markov chains
  - S_0 – UP state, system working
  - S_1 – UP state, recovering failed driver
  - S_2 – DOWN state, system reboot
  - S_3 – DOWN state, Resin reboot
  - $\lambda_{driver\_failure}$ = 4 faults every 8 hrs
  - $\mu_{nooks\_recovery}$ = 4,093 mu seconds
  - μ_{reboot} = 82 seconds
  - c – coverage factor
  - $\lambda_{memory\_leak\_}$ = 1 every 8 hours
  - $\mu_{restart\_resin}$ = 47 seconds





Max availability = 99.835%
Min downtime = 866 minutes

Benefits of CTMCs + Fault Injection

- Able to model and analyze different styles of self-healing mechanisms
- Quantifies the impact of mechanism details (success rates, recovery times etc.) on the system's operational constraints (SLA penalties, availability etc.)
  - Engineering view AND Business view
- Able to identify under-performing mechanisms
- Useful at design time as well as post-production
- Able to control the fault-rates

Caveats of CTMCs + Fault-Injection

- CTMCs may not always be the “right” tool
  - Constant hazard-rate assumption
    - True distribution of faults may be different
  - Fault-independence assumptions
    - Limited to analyzing near-coincident faults
    - Not suitable for analyzing cascading faults (can we model the precipitating event as an approximation?)
- Some failures are harder to replicate/induce than others
  - Better data on faults will improve fault-injection tools
- Getting detailed breakdown of types/rates of failures
  - More data should improve the fault-injection experiments and relevance of the results

Real-World Downtime Data*

- Mean incidents of unplanned downtime in a year: 14.85 (n-tier web applications)
- Mean cost of unplanned downtime (Lost productivity #IT Hours):
  - 2115 hrs (52.88 40-hour work-weeks)
- Mean cost of unplanned downtime (Lost productivity #Non-IT Hours):
  - 515.7 hrs** (12.89 40-hour work-weeks)

* "IT Ops Research Report: Downtime and Other Top Concerns," **StackSafe**. July 2007. (Web survey of 400 IT professional panelists, US Only)
** "Revive Systems Buyer Behavior Research," Research Edge, Inc. June 2007

Quick Analysis – End User View

- Unplanned Downtime (Lost productivity Non-IT hrs) per year: 515.7 hrs (30,942 minutes).
- Is this good? (94.11% Availability)

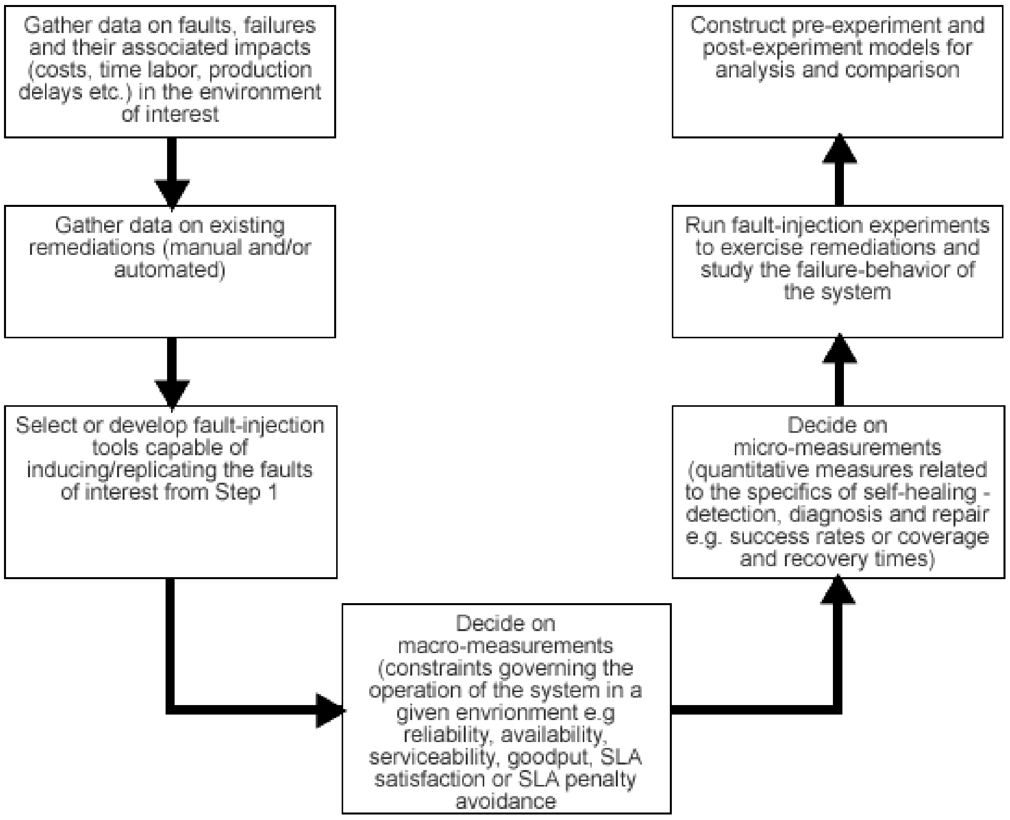
Availability Guarantee	Max Downtime Per Year
99.999	~5 mins
99.99	~53 mins
99.9	~526 mins
99	~5256 mins

- Less than two 9's of availability
  - Decreasing the down time by an order of magnitude could improve system availability by two orders of magnitude

Proposed Data-Driven Evaluation (7U)

- 1. Gather failure data and specify fault-model
- 2. Establish fault-remediation relationship
- 3. Select/create fault-injection tools to mimic faults in 1
- 4. Identify Macro-measurements
  - Identify environmental constraints governing system-operation (SLAs, availability, production targets etc.)
- 5. Identify Micro-measurements
  - Identify metrics related to specifics of self-healing mechanisms (success rates, recovery time, fault-coverage)
- 6. Run fault-injection experiments and record observed behavior
- 7. Construct pre-experiment and post-experiment models

The 7U-Evaluation Method



Conclusions

- Dynamic instrumentation and fault-injection lets us transparently collect data and replicate problems
- The CTMC-models are flexible enough to quantitatively analyze various styles of repairs
- The math is the “easy” part compared to getting customer data on failures, outages, and their impacts.
  - These details are critical to defining the notions of “better” and “good” for these systems

Future Work

- More experiments on an expanded set of operating systems using more server-applications
  - Linux 2.6
  - OpenSolaris 10
  - Windows XP SP2/Windows 2003 Server
- Modeling and analyzing other self-healing mechanisms
  - Error Virtualization (From STEM to SEAD, Locasto et. al Usenix 2007)
  - Self-Healing in OpenSolaris 10
- Feedback control for policy-driven repair-mechanism selection

Questions, Comments, Queries?

Thank you for your time and attention

For more information contact:

Rean Griffith

rg2023@cs.columbia.edu