Tools and Techniques for Designing and Evaluating Self-Healing Systems

Rean Griffith, Ritika Virmani, Gail Kaiser
Programming Systems Lab (PSL)
Columbia University

Presented by
Rean Griffith
Overview

► Introduction
► Challenges
► Problem
► Hypothesis
► Experiments
► 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
Challenges

How do we evaluate the efficacy of a self-healing system and its mechanisms?

- How do we quantify the impact of the problems these systems should resolve?
- How can we reason about expected benefits for systems currently lacking self-healing mechanisms?
- How do we quantify the efficacy of individual and combined self-healing mechanisms and reason about tradeoffs?
- How do we identify sub-optimal mechanisms?
Motivation

► Performance metrics are not a perfect proxy for “better self-healing capabilities”
  ▪ Faster != “Better at self-healing”
  ▪ Faster != “Has better self-healing facilities”

► Performance metrics provide insights into the feasibility of using a self-healing system with its self-healing mechanisms active

► Performance metrics are still important, but they are not the complete story
Problem

Evaluating self-healing systems and their mechanisms is non-trivial

- Studying the failure behavior of systems can be difficult
- Finding fault-injection tools that exercise the remediation mechanisms available is difficult
- Multiple styles of healing to consider (reactive, preventative, proactive)
- Accounting for imperfect repair scenarios
- Partially automated repairs are possible
Proposed Solutions

► Studying failure behavior
  ▪ “In-situ” observation in deployment environment via dynamic instrumentation tools

► Identifying suitable fault-injection tools
  ▪ “In-vivo” fault-injection at the appropriate granularity via runtime adaptation tools

► Analyzing multiple remediation styles and repair scenarios (perfect vs. imperfect repair, partially automated healing etc.)
  ▪ Mathematical models (Continuous Time Markov Chains, Control Theory models etc.)
Hypotheses

- Runtime adaptation is a reasonable technology for implementing efficient and flexible fault-injection tools.
- Mathematical models e.g. Continuous Time Markov Chains (CTMCs), Markov Reward Models and Control Theory models are a reasonable framework for analyzing system failures, remediation mechanisms and their impact on system operation.
- Combining runtime adaptation with mathematical models allows us to conduct fault-injection experiments that can be used to investigate the link between the details of a remediation mechanism and the mechanism’s impact on the high-level goals governing the system’s operation, supporting the comparison of individual or combined mechanisms.
Runtime Fault-Injection Tools

► Kheiron/JVM (ICAC 2006)
  - Uses byte-code rewriting to inject faults into running Java applications
  - Includes: memory leaks, hangs, delays etc.
  - Two other versions of Kheiron exist (CLR & C)
  - C-version uses Dyninst binary rewriting tool

► Nooks Device-Driver Fault-Injection Tools
  - Developed at UW for Linux 2.4.18 (Swift et. al)
  - Uses the kernel module interface to inject faults
  - Includes: text faults, stack faults, hangs etc.
  - We ported it to Linux 2.6.20 (Summer 07)
Mathematical Techniques

- Continuous Time Markov Chains (PMCCS-8)
  - Reliability & Availability Analysis
  - Remediation styles

- Markov Reward Networks (PMCCS-8)
  - Failure Impact (SLA penalties, downtime)
  - Remediation Impact (cost, time, labor, production delays)

- Control Theory Models (Preliminary Work)
  - Regulation of Availability/Reliability Objectives
  - Reasoning about Stability
Fault-Injection Experiments

Objective

- To inject faults into the components a multi-component n-tier web application – specifically the application server and Operating System components
- Observe its responses and the responses of any remediation 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
Healing Mechanisms Available

► Application Server
  ▪ Automatic restarts

► Operating System
  ▪ Nooks device driver protection framework
  ▪ Manual system reboot
Metrics

► 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.)
  ▪ Expected SLA penalties
Application Server Memory Leaks

Memory leak condition causing an automatic application server restart every 8.1593 hours (95% confidence interval)
Resin Memory-Leak Handler
Analysis

► Analyzing perfect recovery e.g. mechanisms addressing resource leaks/fatal crashes
  - $S_0$ – UP state, system working
  - $S_1$ – DOWN state, system restarting
  - $\lambda_{\text{failure}} = 1$ every 8 hours
  - $\mu_{\text{restart}} = 47$ 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

<table>
<thead>
<tr>
<th>Availability guarantee</th>
<th>Max downtime per year</th>
<th>Expected penalties</th>
</tr>
</thead>
<tbody>
<tr>
<td>99.999</td>
<td>~5 mins</td>
<td>$(866 - 5)*p$</td>
</tr>
<tr>
<td>99.9</td>
<td>~53 mins</td>
<td>$(866 - 53)*p$</td>
</tr>
<tr>
<td>99.0</td>
<td>~526 mins</td>
<td>$(866 - 526)*p$</td>
</tr>
<tr>
<td>99.0</td>
<td>~5256 mins</td>
<td>$0$</td>
</tr>
</tbody>
</table>
Linux w/Nooks Recovery Analysis

- 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
  - $\lambda_{\text{driver\_failure}} = 4$ faults every 8 hrs
  - $\mu_{\text{nooks\_recovery}} = 4,093$ mu seconds
  - $\mu_{\text{reboot}} = 82$ seconds
  - $c$ – coverage factor/success rate
Resin + Linux + Nooks Analysis

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

- $\lambda_{\text{driver\_failure}} = 4$ faults every 8 hrs
- $\mu_{\text{nooks\_recovery}} = 4,093$ mu seconds
- $\mu_{\text{reboot}} = 82$ seconds
- $c$ – coverage factor
- $\lambda_{\text{memory\_leak}} = 1$ every 8 hours
- $\mu_{\text{restart\_resin}} = 47$ seconds

Max availability = 99.835%
Min downtime = 866 minutes
Proposed Preventative Maintenance

- Non-Birth-Death process with 6 states, 6 parameters:
  - $S_0$ – UP state, first stage of lifetime
  - $S_1$ – UP state, second stage of lifetime
  - $S_2$ – DOWN state, Resin reboot
  - $S_3$ – UP state, inspecting memory use
  - $S_4$ – UP state, inspecting memory use
  - $S_5$ – DOWN state, preventative restart
  - $\lambda_{2ndstage} = 1/6 \text{ hrs}$
  - $\lambda_{failure} = 1/2 \text{ hrs}$
  - $\mu_{restart\_resin\_worst} = 47 \text{ seconds}$
  - $\lambda_{inspect} = \text{Memory use inspection rate}$
  - $\mu_{inspect} = 21,627 \text{ microseconds}$
  - $\mu_{restart\_resin\_pm} = 3 \text{ seconds}$
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 (availability, production targets, production-delay reduction 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
    - May under or overstate the effects/impacts
    - 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 could 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)

Proposed Data-Driven Evaluation (7U)

1. Gather failure data and specify fault-model
2. Establish fault-remediation relationship
3. Select fault-injection tools to mimic faults in 1
4. Identify Macro-measurements
   - Identify environmental constraints governing system-operation (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

1. Gather data on faults, failures and their associated impacts (costs, time labor, production delays etc.) in the environment of interest

2. Gather data on existing remediations (manual and/or automated)

3. Select or develop fault-injection tools capable of inducing/replicating the faults of interest from Step 1

4. Decide on macro-measurements (constraints governing the operation of the system in a given environment e.g. reliability, availability, serviceability, goodput, SLA satisfaction or SLA penalty avoidance

5. Construct pre-experiment and post-experiment models for analysis and comparison

6. Run fault-injection experiments to exercise remediations and study the failure-behavior of the system

7. Decide on micro-measurements (quantitative measures related to the specifics of self-healing - detection, diagnosis and repair e.g. success rates or coverage and recovery times)
Preliminary Work – Control Models

Objective

- Can we reason about the stability of the system when the system has multiple repair choices for individual faults using Control Theory?
- Can we regulate availability/reliability objectives?
- What are the pros & cons of trying to use Control Theory in this context?
Expected Downtime = \( f(\text{Reference/Desired Success Rate}) \)

Measured Downtime = \( f(\text{Actual Success Rate}) \)

Smoothed Downtime Estimate = \( f(\text{Actual Success Rate}) \)
Preliminary Work – Control Parameters

- **D_E(z)** – represents the occurrence of faults
  - Signal magnitude equals worst case repair time/desired repair time for a fault
- **Expected downtime** = \( f(\text{Reference Success Rate}) \)
- **Smoothed downtime estimate** = \( f(\text{Actual Success Rate}) \)
- **Downtime error** – difference between desired downtime and actual downtime incurred
- **Measured Downtime** – repair time impact on downtime.
  - 0 for transparent repairs or \( 0 < r \leq D_E(k) \) if not
- **Smoothed Downtime Estimate** – the result of applying a filter to Measured Downtime
Preliminary Simulations

- Reason about stability of repair selection controller/subsystem, $R(z)$, using the poles of transfer function $R(z)/[1+R(z)H_R(z)]$
- Show stability properties as expected/reference success rate and actual repair success rate vary
- How long does it take for the system to become unstable/stable
Preliminary Work – Desired Goal

Can we extend the basic model to reason about repair choice/preferences?
Conclusions

► Dynamic instrumentation and fault-injection lets us transparently collect data “in-situ” and replicate problems “in-vivo”

► The CTMC-models are flexible enough to quantitatively analyze various styles and “impacts” of repairs

► We can use them at design-time or post-deployment time

► 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
Acknowledgements

► Prof. Gail Kaiser (Advisor/Co-author), Ritika Virmani (Co-author)

► Prof. Angelos Keromytis (Secondary Advisor), Carolyn Turbyfill Ph.D. (Stacksafe Inc.), Prof. Michael Swift (formerly of the Nooks project at UW now a faculty member at University of Wisconsin), Prof. Kishor Trivedi (Duke/SHARPE), Joseph L. Hellerstein Ph.D., Dan Phung (Columbia University), Gavin Maltby, Dong Tang, Cynthia McGuire and Michael W. Shapiro (all of Sun Microsystems).

► Our Host: Matti Hiltunen (AT&T Research)
Questions, Comments, Queries?

Thank you for your time and attention

For more information contact:

Rean Griffith
rg2023@cs.columbia.edu
Extra Slides
How Kheiron Works

► Key observation
   ▪ All software runs in an execution environment (EE), so use it to facilitate performing adaptations (fault-injection operations) in the applications it hosts.

► Two kinds of EEs
   ▪ Unmanaged (Processor + OS e.g. x86 + Linux)
   ▪ Managed (CLR, JVM)

► For this to work the EE needs to provide 4 facilities...
## EE-Support

<table>
<thead>
<tr>
<th>EE Facilities</th>
<th>Unmanaged Execution Environment</th>
<th>Managed Execution Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELF Binaries</td>
<td>JVM 5.x</td>
<td>CLR 1.1</td>
</tr>
<tr>
<td>Program tracing</td>
<td>ptrace, /proc</td>
<td>JVMTI callbacks + API</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ICorProfilerInfo</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ICorProfilerCallback</td>
</tr>
<tr>
<td>Program control</td>
<td>Trampolines + Dyninst</td>
<td>Bytecode rewriting</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MSIL rewriting</td>
</tr>
<tr>
<td>Execution unit metadata</td>
<td>.symtab, .debug sections</td>
<td>Classfile constant pool + bytecode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Assembly, type &amp; method metadata + MSIL</td>
</tr>
<tr>
<td>Metadata augmentation</td>
<td>N/A for compiled C-programs</td>
<td>Custom classfile parsing &amp; editing APIs + JVMTI RedefineClasses</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IMetaDataImport, IMetaDataEmit APIs</td>
</tr>
</tbody>
</table>
SampleMethod( args ) [throws NullPointerException]
<room for prolog>
push args
call _SampleMethod( args ) [throws NullPointerException]
{ try{…} catch (IOException ioe){…} } // Source view of _SampleMethod’s body
<room for epilog>
return value/void
public void someMethod()
{
    call StatsCop.methodEnter( "someMethod" ) // profile method enter
    call FaultManager.injectFault( "someMethod" ) // lookup fault to inject
    call _someMethod(); // call original implementation of someMethod
    call StatsCop.methodExit( "someMethod" ) // profile method exit
}
## Kheiron/C Operation

<table>
<thead>
<tr>
<th>Mutator</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kheiron/C</td>
<td>void foo(int x, int y) { int z = 0; }</td>
</tr>
<tr>
<td>Dyninst API</td>
<td>Snippets</td>
</tr>
<tr>
<td>Dyninst Code</td>
<td>C/C++ Runtime Library</td>
</tr>
<tr>
<td>ptrace/procfs</td>
<td></td>
</tr>
</tbody>
</table>

```c
void foo(int x, int y) {
    int z = 0;
}
```
Kheiron/C – Prologue Example
Kheiron/CLR & Kheiron/JVM Feasibility

Kheiron/CLR Overheads when no adaptations active

Kheiron/JVM Overheads when no adaptations active
Kheiron/C Feasibility

Performance comparison SciMark - normalized to w/o Dyninst - simple jump into adaptation library

Kheiron/C Overheads
when no adaptations active
Kheiron Summary

- Kheiron supports contemporary managed and unmanaged execution environments.
- Low-overhead (<5% performance hit).
- Transparent to both the application and the execution environment.
- Access to application internals
  - Class instances (objects) & Data structures
  - Components, Sub-systems & Methods
- Capable of sophisticated adaptations.
- Fault-injection tools built with Kheiron leverage all its capabilities.
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)

<table>
<thead>
<tr>
<th>Availability Guarantee</th>
<th>Max Downtime Per Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>99.999</td>
<td>~5 mins</td>
</tr>
<tr>
<td>99.99</td>
<td>~53 mins</td>
</tr>
<tr>
<td>99.9</td>
<td>~526 mins</td>
</tr>
<tr>
<td>99</td>
<td>~5256 mins</td>
</tr>
</tbody>
</table>

► 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