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

Rean Griffith, Ritika Virmani, Gail Kaiser
Columbia University
New York, USA
Email: {rg2023, rv2171, kaiser@cs}.columbia.edu

Abstract

To evaluate the efficacy of self-healing systems a rigorous, objective, quantitative benchmarking methodology is needed. However, developing such a benchmark is a non-trivial task given the many evaluation issues to be resolved, including but not limited to: quantifying the impacts of faults, analyzing various styles of healing (reactive, preventative, proactive), accounting for partially automated healing and accounting for incomplete/imperfect healing. We posit, however, that it is possible to realize a self-healing benchmark using a collection of analytical techniques and practical tools as building blocks. This paper highlights the flexibility of one analytical tool, the Reliability, Availability and Serviceability (RAS) model, and illustrates its power and relevance to the problem of evaluating self-healing mechanisms/systems, when combined with practical tools for fault-injection.

I. Introduction

In this paper we employ analytical techniques based on Continuous Time Markov Chains (CTMCs), specifically Birth-Death processes and Non-Birth-Death processes to construct Reliability, Availability and Serviceability (RAS) models, which we use to evaluate a system's self-healing mechanisms and its overall self-healing capabilities. We study the existing self-healing mechanisms of an operating system and an application server hosting an N-tier web-application. These recovery/repair mechanisms are exercised using focused fault-injection. Based on the experimental results of our fault-injection experiments we construct a set of RAS-models, evaluate each remediation mechanism individually, and then study the combined impact of the remediation mechanisms on the system.

This paper makes two contributions. First, we show the flexibility of simple RAS-models as they easily address two of the challenges of evaluating self-healing systems – quantifying the impact of imperfect remediation scenarios and analyzing the various styles of remediations (reactive, preventative and proactive). This flexibility positions RAS-models as a practical analysis tool for aiding in the development of a comprehensive, quantitative benchmark for self-healing systems.

Second, we demonstrate that it is possible to analyze fault-injection experiments under simplifying Markovian assumptions about fault distributions and system-failures and still glean useful insights about the efficacy of a system's self-healing mechanisms. Further, we show that the metrics obtained from RAS-models – remediation success rates, limiting availability, limiting reliability and expected yearly downtime – allow us to identify sub-optimal remediation mechanisms and reason about the design of improved remediation mechanisms.

The goal of our experiments is to inject faults into specific components of the system under test and study its response. The faults we inject are intended to exercise the remediation mechanisms of the system. We use the experimental data to mathematically model the impact of the faults we inject on the system's reliability, availability and serviceability with and without the remediation mechanisms.

Whereas our fault-injection experiments may expose the system to rates of failure well above what the system may see in a given time period, these artificially high failure rates allow us to

explore the expected and unexpected system responses under stressful fault conditions, much like performance benchmarks subject the system under test to extreme workloads.

To conduct our experiments we need: a test-platform, i.e. a hardware/software stack executing a reasonable workload, a fault model, fault-injection tools, a set of remediation mechanisms and a set of system configurations.

For our test platform we use VMWare GSX virtual machines configured with: 512 MB RAM, 1 GB of swap, an Intel x86 Core Solo processor and an 8 GB harddisk running Redhat 9 on 2.4.18 kernels. We use an instance of the TPC-W web-application (based on the implementation developed at the University of Madison-Wisconsin) running on MySQL 5.0.27, the Resin 3.0.22 application server and webserver, and Sun Microsystems' Hotspot Java Virtual Machine (JVM), v1.5. We simulate a load of 20 users using the Shopping Mix [1] as their web-interaction strategy. User-interactions are simulated using the Remote Browser Emulator (RBE) software also implemented at the University of Madison-Wisconsin. Our VMs are hosted on a machine configured with 2 GB RAM, 2 GB of swap, an Intel Core Solo T3100 Processor (1.66 GHz) and a 51 GB harddisk running Windows XP SP2.

Our fault model consists of device driver faults targeting the Operating System and memory leaks targeting the application server. We chose device driver faults because device drivers account for $\sim$70% of the Linux kernel code and have error rates seven times higher than the rest of the kernel [2] – faulty device drivers easily compromise the integrity and reliability of the kernel. While memory leaks and general state corruption (dangling pointers and damaged heaps) are highlighted as common bugs leading to system crashes in large-scale web deployments [3].

We identified the operating system and the application server as candidate targets for fault-injection. Given the operating system's role as resource manager [4] and part of the native execution environment for applications [5] its reliability is critical to the overall stability of the applications it hosts. Similarly, application servers act as containers for web-applications responsible for providing a number of services, including but not limited to, isolation, transaction management, instance management, resource management and synchronization. These responsibilities make application-servers another critical link in a web-application's reliability and another prime target for fault-injection. Database servers are also reasonable targets for fault-injection; however database fault-injection experiments are outside the scope of this work but will be the focus of future work.

We use a version of the SWIFI device driver fault-injection tools [6], [7] (University of Washington) and a tool based on our own Kheiron/JVM [5] implementation for application-server fault-injection.

There are three remediation mechanisms we consider: (manual) system reboots, (automatic) application server restarts and Nooks device driver protection and recovery [6] – Nooks isolates the kernel from device drivers using lightweight protection domains, as a result driver crashes are less likely to cause a kernel crash. Further, Nooks supports the transparent recovery of a failed device driver.

Finally, we use the following system-configurations: **Configuration A** – Fault-free system operation, **Configuration B** – System operation in the presence of memory leaks, **Configuration C** – System operation in the presence of device-driver failures (Nooks disabled), **Configuration D** – System operation in the presence of device-driver failures (Nooks enabled), and **Configuration E** – System operation in the presence of memory leaks and driver failures (Nooks enabled).

II. Results and Analysis

In our experiments we measure both client-side and server-side activity. On the client-side we use the number of web interactions and client-perceived rate of failure to determine client-side availability.

A typical fault-free run of the TPC-W (**Configuration A**), takes $\sim$24 minutes to complete and records 3973 successful client-side interactions.

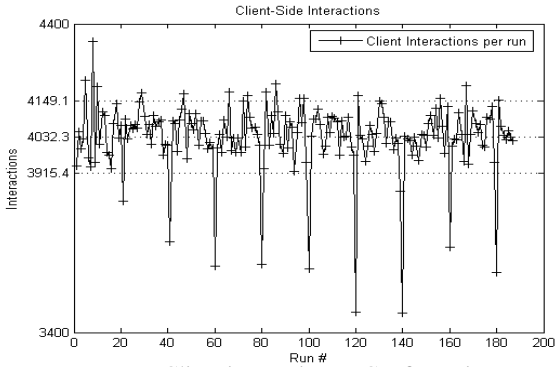

Fig. 1. *Client interactions – Configuration B*

Figure 1 shows the client-side goodput over ∼76 hours of continuous execution (187 runs) in the presence of an accumulating memory leak – **Configuration B**. The average number of client-side interactions over this series of experiments is 4032.3 ± 116.8473. In this figure there are nine runs where the number of client interactions is 2 or more standard deviations below the mean. Client-activity logs indicate a number of successive failed HTTP requests over an interval of ∼1 minute during these runs. Resin's logs indicate that the server encounters a low-memory condition, forces a number of JVM garbage collections before restarting the application server. During the restart, requests sent by RBE-clients fail to complete. A poisson fit of the time-intervals between these nine runs at the 95% confidence interval yields a hazard rate of 1 memory-leak related failure (Resin restart) every 8.1593 hours.

Figure 2 shows a trace sampling the number of client interactions completed every 60 seconds for a typical run, (Run #2), compared to data from some runs where low memory conditions cause Resin to restart. Data obtained from Resin's logs record startup times of 3,092 msecs (initial startup) and restart times of approximately 47,582 msecs.

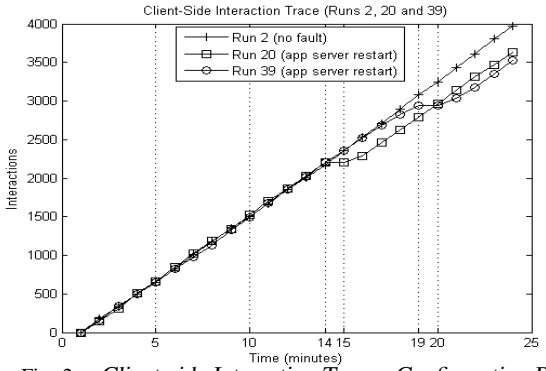

Fig. 2. *Client-side Interaction Trace - Configuration B*

To evaluate the RAS-characteristics of the system in the presence of the memory leak, we use the SHARPE RAS-modeling and analysis tool [8] to create the basic 2-node, 2-parameter RAS-model shown in Figure 3. Table 4 lists the model's parameters.

Whereas the model shown in Figure 3 implicitly assumes that the detection of the low memory condition is perfect and the restart of the application server resolves the problem 100% of the time, in this instance these assumptions are validated by the experiments.

Using the steady-state/limiting availability formula [9]: $A = \frac{\frac{1}{\lambda}}{\frac{1}{\lambda}+\frac{1}{\mu}}$ the steady state availability of the system is 99.838%. Further, the system has an expected downtime of 866 minutes per year – given by the formula $(1 - Availability) * T$ where $T = 525,600$ minutes in a year. At best,

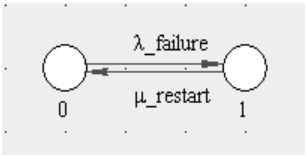

Fig. 3. *Simple RAS Model*

S_0	an UP state where the system services requests
S_1	a DOWN state, no client requests are serviced while the application server is being restarted
$\lambda_{failure}$	observed rate of failure, 1 failure every 8 hours
$\mu_{restart}$	time to restart the application server, ∼47 seconds

Fig. 4. *RAS-Model Parameters – Configuration B*

the system is capable of delivering two 9's of availability. Table I shows the expected penalties per year for each minute of downtime over the allowed limit. As an additional consideration, downtime may also incur costs in terms of time and money spent on service visits, parts and/or labor, which add to any assessed penalties.

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

TABLE I
Expected SLA Penalties for Configuration B

In **Configuration C** we inject faults into the pcnet32 device driver with Nooks driver protection disabled. Each injected fault leads to a kernel panic requiring a reboot to make the system operational again. For this set experiments we arbitrarily choose a fault rate of 4 device failures every 8 hours and use the SWIFI tools to achieve this rate of failures in our system under test. The fact that that the remediation mechanism (the reboot) always restores the system to an operational state allows us to reuse the basic 2-parameter RAS-model shown in Figure 3 to evaluate the RAS-characteristics of the system in the presence of device driver faults. Table II shows the parameters of the model.

S_0	an UP state where the system services requests
S_1	a DOWN state, no client requests are serviced while the application server is being restarted
$\lambda_{failure}$	achieved rate of failure, 4 failures every 8 hours
$\mu_{restart}$	time to reboot the system, 1 minute 22 seconds

TABLE II
RAS-Model Parameters – Configuration C

Using SHARPE, we calculate the steady state availability of the system as 98.87%, with an expected downtime of 5924 minutes per year i.e. under this fault-load the system cannot deliver two nines of availability.

Next we consider the case of the system under test enhanced with Nooks device driver protection enabled – **Configuration D**. Whereas we reuse the same fault-load and fault-rate, 4 device driver failures every 8 hours, we need to revise the RAS-model used in our analysis to account for the possibility of imperfect repair i.e. to handle cases where Nooks is unable to recover the failed device driver and restore the system to an operational state. To achieve this we use the RAS-model shown in Figure 5, its parameters are listed in Figure 6.

Figure 7 shows the expected impact of Nooks recovery on the system's RAS-characteristics as its success rate varies.

Whereas Configuration C of the system under test is unable to deliver two 9's of availability in the presence of device driver faults, a modest 20% success rate from Nooks is expected to

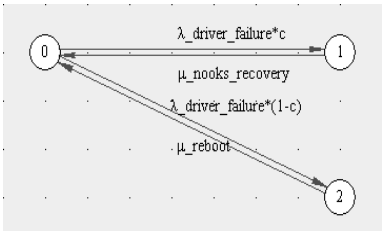

Fig. 5. *RAS-Model of a system with imperfect repair*

S_0	an UP state where the system services requests
S_1	an UP state, where Nooks is recovering a failed driver
S_2	a DOWN state, where Nooks' recovery fails and the system needs to be rebooted
$\lambda_{driver\_failure}$	achieved rate of failure, 4 failures every 8 hours
$\mu_{nooks\_recovery}$	time for Nooks to successfully recover a failed device driver 4093 microseconds worst case
c	the *coverage factor*, represents the success rate of Nooks, varying this parameter lets us study the impact of imperfect recovery
μ_{reboot}	time to reboot the system, 1 min 22 secs

Fig. 6. *RAS-Model Parameters – Configuration D*

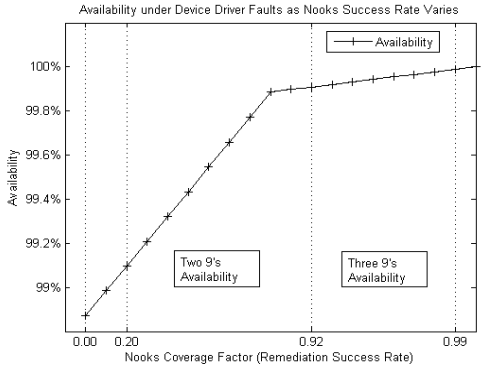

Fig. 7. *Availability – Configuration D*

promote the system into another availability bracket while a 92% success rate reduces the expected downtime and SLA penalties by two orders of magnitude (see Figure 7) [1].

Thus far we have analyzed the system under test and each fault in isolation i.e. each RAS-model we have developed so far considers one fault and its remediations. We now develop a RAS-model that considers all the faults in our fault-model and the remediations available, **Configuration E** – see Figure 8.

Figure 9 shows the expected availability of the complete system. The system's availability is limited to two 9's of availability even though the system could deliver better availability and downtime numbers – the minimum system downtime is calculated as 866 minutes per year, the same as for Configuration B, the memory leak scenario. Thus, even with perfect Nooks recovery, the system's availability is limited by the reactive remediation for the memory leak. To improve the system's overall availability we need to improve the handling of the memory leak. One option for improvement is to consider whether preventative maintenance could be beneficial. [10] presents a model of a preventative maintenance mechanism to address this memory leak scenario.

[1]In our experiments we were unable to encounter a scenario where Nooks was unable to successfully recover a failed device driver; however the point of our exercise is to demonstrate how that eventually could be accounted for in an evaluation of a remediation mechanism.

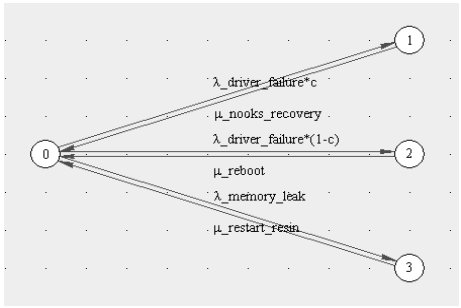

Fig. 8. *Complete RAS-model – Configuration E*

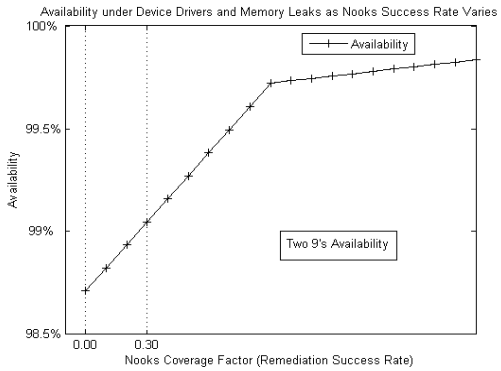

Fig. 9. *Availability – Configuration E*

III. Potential Implications for Self-Healing Benchmarks

A number of requirements for a self-healing benchmark have previously been proposed. These requirements include, but are not limited to: providing quantitative measures [11], handling partially-autonomic systems, accounting for incomplete healing, accounting for healing-specific resources and being able to quantify healing effectiveness [12] – which depends on establishing the appropriate fault-model/hypothesis driving the evaluation [13].

In our experiments we used RAS-models to provide quantitative metrics (steady-state availability, yearly downtime and possible SLA penalties) for comparing individual remediation mechanisms (Table I). Models of imperfect repair (Figure 5) allow us to study incomplete-healing scenarios and partially automated systems via the use of branches to multiple states representing choices of (manual/automated) remediations for a single failure and coverage factors representing the probability of having to select a particular remediation. Converting Figure 5 into a Markov Reward Network, by assigning a numeric value such as labor costs or time to each state in the model, would allow us to introduce additional criteria for evaluating and comparing remediation choices. Finally, the ability to compose RAS-models (and reward networks), as shown in Figure 8, allows us to consider the details of the system (e.g. remediation success rates, costs or time overheads etc.) in the context of the big picture – the impact of the fault-model under consideration on the target system and the overall efficacy of combining remediation mechanisms.

We posit that RAS-models and extensions such as Markov Reward Networks are reasonable mathematical tools for meeting these self-healing benchmarking requirements. Further, the relevance of these mathematical tools and the results they produce is enhanced by the availability of failure impact data and fault-injection tools capable of replicating the faults or inducing the failures described in that dataset.

Other examples of using RAS-models and Markov Reward Networks to evaluate self-healing mechanisms can be found in [14] and [15] respectively. While, [16], [17] and [18] highlight some of the issues involved in non-performance-oriented benchmarking.

Looking ahead, we envision employing RAS-models as one analytical tool in a data-driven methodology for evaluating self-healing systems. One example of such a data-driven methodology is our 7U-evaluation methodology, outlined in Table III, which informed the design of the fault-injection experiments in Section II.

Acknowledgments

The Programming Systems Laboratory is funded in part by NSF grants CNS-0627473, CNS-0426623 and EIA-0202063, NIH grant 1U54CA121852-01A1, and Consolidated Edison Company of New York. We would like to thank Dan Phung,

Step #	Summary
1 (Fault Hypothesis)	Construct a fault model to evaluate the system against using data (logs, trouble tickets etc.) on faults, failures and their associated impacts (frequency of occurrence, costs, labor, downtime, production delays etc.)
2 (Fault-remediation relationship)	Identify existing remediations (manual or automated) and associate them with the faults they can address to identify ideal/acceptable system responses where possible
3 (Practical fault-injection tools)	Select or develop tools capable of inducing/replicating the faults of interest from (1)
4 (Environmental constraints)	Identify macro-measurements (constraints governing the operation of the system in a given environment e.g. reliability, availability, serviceability, goodput or SLA targets or SLA penalty-avoidance goals)
5 (Detailed remediation metrics)	Identify micro-measurements (metrics related to the specifics of self-healing e.g. remediation success rates or coverage and recovery times that potentially impact the macro-measurements)
6 (Observed system responses)	Run fault-injection experiments to exercise remediations and study the failure-behavior of the system
7 (Analysis and Reports)	Construct pre-experiment and post-experiment models for analysis and comparison

TABLE III
7U-Evaluation Method

Prof. Jason Nieh, Prof. Angelos Keromytis (all of Columbia University), Gavin Maltby, Dong Tang, Cynthia McGuire and Michael W. Shapiro (all of Sun Microsystems) for their insightful comments and feedback. We would also like to thank Prof. Michael Swift (formerly a member of the Nooks project, now a faculty member at the University of Wisconsin-Madison) for his assistance configuring and running Nooks. Finally, we wish to thank Dr. Kishor Trivedi (of Duke University) for granting us permission to use SHARPE.

REFERENCES

[1] D. Menasce, "TPC-W A Benchmark for E-Commerce," http://ieeexplore.ieee.org/iel5/4236/21649/01003136.pdf, 2002.

[2] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler, "An empirical study of operating system errors," in *Symposium on Operating Systems Principles*, 2001, pp. 73–88. [Online]. Available: citeseer.ist.psu.edu/chou01empirical.html

[3] G. Candea, J. Cutler, and A. Fox, "Improving Availability with Recursive Micro-Reboots: A Soft-State Case Study," in *Dependable systems and networks - performance and dependability symposium*, 2002.

[4] A. S. Tanenbaum, *Modern Operating Systems*. Prentice Hall, 1992, tAN a 92:1 2.Ex.

[5] R. Griffith and G. Kaiser, "A Runtime Adaptation Framework for Native C and Bytecode Applications," in *3rd International Conference on Autonomic Computing*, 2006.

[6] Michael Swift et al., "Improving the Reliability of Commodity Operating Systems," in *International Conference Symposium on Operating Systems Principles*, 2003.

[7] Michael M. Swift et al., "Recovering Device Drivers," in *6th Symposium on Operating System Design and Implementation*, 2004.

[8] A. R. Sahner and S. K. Trivedi, "Reliability modeling using sharpe," Durham, NC, USA, Tech. Rep., 1986.

[9] Kishor S. Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications 2nd Edition*. Wiley, 2002.

[10] R. Griffith, R. Virmani, and G. Kaiser, "The Role of Reliability, Availability and Serviceability (RAS) Models in the Design and Evaluation of Self-Healing Systems," Columbia University, Tech. Rep. CUCS-021-07, 2007.

[11] A. Brown et al., "Benchmarking autonomic capabilities: Promises and pitfalls," in *Proceedings of the 1st International Conference on Autonomic Computing. IEEE NSF May 2004*, 2004.

[12] A. Brown and C. Redlin, "Measuring the Effectiveness of Self-Healing Autonomic Systems," in *2nd International Conference on Autonomic Computing*, 2005.

[13] P. Koopman, "Elements of the self-healing problem space," in *Proceedings of the ICSE Workshop on Architecting Dependable Systems*, 2003.

[14] D. Tang et al., "Assessment of the Effect of Memory Page Retirement on System RAS Against Hardware Faults," in *International Conference on Dependable Systems and Networks*, 2006.

[15] Dong Tang et al., "Availability Measurement and Modeling for an Application Server," in *International Conference on Dependable Systems and Networks*, 2004.

[16] I. S. T. (IST), "Dependability benchmarking project final report," http://www.laas.fr/DBench/Final/DBench-complete-report.pdf.

[17] A. Brown, "Towards availability and maintainability benchmarks: A case study of software raid systems," Masters thesis, University of California, Berkeley, 2001, uCB//CSD011132.

[18] T. K. Tsai, R. K. Iyer, and D. Jewitt, "An approach towards benchmarking of fault-tolerant commercial systems," in *Symposium on Fault-Tolerant Computing*, 1996, pp. 314–323. [Online]. Available: citeseer.ist.psu.edu/tsai96approach.html