The Role of Reliability, Availability and Serviceability (RAS) Models in the Design and Evaluation of Self-Healing Systems

Rean Griffith
Columbia University
rg2023@cs.columbia.edu

Ritika Virmani
Columbia University
rv2171@columbia.edu

Gail Kaiser
Columbia University
kaiser@cs.columbia.edu

Abstract

In an idealized scenario, self-healing systems predict, prevent or diagnose problems and take the appropriate actions to mitigate their impact with minimal human intervention. To determine how close we are to reaching this goal we require analytical techniques and practical approaches that allow us to quantify the effectiveness of a system's remediations mechanisms. In this paper we apply analytical techniques based on Reliability, Availability and Serviceability (RAS) models to evaluate individual remediation mechanisms of select system components and their combined effects on the system. We demonstrate the applicability of RAS-models to the evaluation of self-healing systems by using them to analyze various styles of remediations (reactive, preventative etc.), quantify the impact of imperfect remediations, identify sub-optimal (less effective) remediations and quantify the combined effects of all the activated remediations on the system as a whole.

1 Introduction

Self-healing systems are expected to respond to problems that arise in their environments with minimal human intervention. To achieve this, these systems may employ a variety of remediation strategies including preventative, proactive and reactive strategies. In the ideal case, such systems will use a combination of these strategies to predict, prevent or diagnose and react to problems. To determine how close we are to realizing this goal we require rigorous analytical tools that quantify the efficacy of the remediation mechanisms, and by extension, allow for quantitative evaluations of entire self-healing systems.

The most well-understood evaluation tools for general computing systems however, are performance-centric. Performance benchmarks, such as those produced by the National Institute of Science and Technology (NIST) [19], the Standard Performance Evaluation Corporation (SPEC®) [23] and the Transaction Processing Performance Council (TPC™) [26], are routinely used to demonstrate that an experimental system prototype is better than the state-of-the-art because it delivers acceptable or superior performance. They are also used as reasonable workloads to exercise a system during experiments.

Whereas performance benchmarks can also be a source of reasonable workloads for self-healing systems, performance numbers, however, do not allow us to draw conclusions about one system being "better" than another with respect to its ability to heal itself. Measures concerned with overall system performance do not adequately capture the details that distinguish one remediation mechanism from another e.g. *remediation accuracy/success rates, the impact of remediation failures, the consequences of remediation style, remediation timings, the impact of remediations on system manageability and accounting for partially automated remediations*. This deficiency of performance benchmarks limits our ability to use them as the primary means of comparing or ranking self-healing systems.

We posit that analytical tools that can be adapted to study the many facets of remediations, including but not limited to those mentioned, will provide greater insights into the design, development and evaluation of self-healing systems than pure performance-evaluations.

In addition to analyzing the details of remediation mechanisms, there are a number of high-level measures that also can be used to differentiate self-healing systems. According to [11], a self-healing system "...automatically detects, diagnoses and repairs localized software and hardware problems" [11]. As a result, it is reasonable to expect that these systems exhibit fewer severe-outages, better *reliability* and *availability* characteristics than vanilla systems (i.e. systems lacking self-healing capabilities). The system's diagnostic capabilities are expected to enhance its manageability by improving the selection and execution of (fully or partially automated) repair mechanisms. Where 100% automated repair mechanisms are lacking, effective, automated diagnosis can guide human operators, reducing the total time and effort needed for performing repairs. We also expect these systems to exhibit lower *serviceability* "costs" e.g. lower yearly downtimes and a lower frequency of unexpected servicing/maintenance events.

In this paper we employ analytical techniques based on Continuous Time Markov Chains (CTMCs) [12], 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 remainder of this paper is organized as follows; §2 defines some key terms. §3 describes the setup of the fault-injection experiments conducted on our N-tier web application's components. §4 analyzes and discusses the results. §5 covers related work and §6 presents our conclusions and describes future work.

2 Terminology

In this section we formalize some of the terms used throughout this paper.

- **Error** – the deviation of system external state from correct service state [14].
- **Fault** – the adjudged or hypothesized cause of an error [14].
- **Fault Hypothesis/Fault Model** – the set of faults a self-healing system is expected to be able to heal [13].
- **Remediation** – the process of correcting a fault. In this paper remediation spans the activities of detection, diagnosis and repair since the first step in responding to a fault is detection [13].
- **Failure** – an event that occurs when the delivered service violates an environmental/contextual constraint e.g. a policy or SLA. This definition emphasizes the client-side (end-user's) perspective over the server-side perspective [1].
- **Reliability** – the number (or frequency) of client-side interruptions.
- **Availability** – a function of the rate of failure/maintenance events and the speed of recovery [10].
- **Serviceability** – a function of the number of service-visits and their duration and costs.

3 Experiments

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 [16] 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 ~70% of the Linux kernel code and have error rates seven times higher than the rest of the kernel [4] – 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 [25] and part of the native execution environment for applications [8] 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 [17, 18] (University of Washington) and a tool based on our own Kheiron/JVM [8] 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 [17] – 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).

4 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 ~24 minutes to complete and records 3973 successful client-side interactions.

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 encounteres 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.

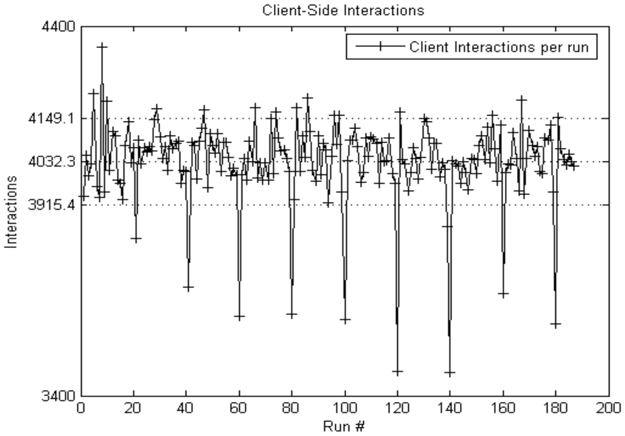

Figure 1. *Client interactions – Configuration B*

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.

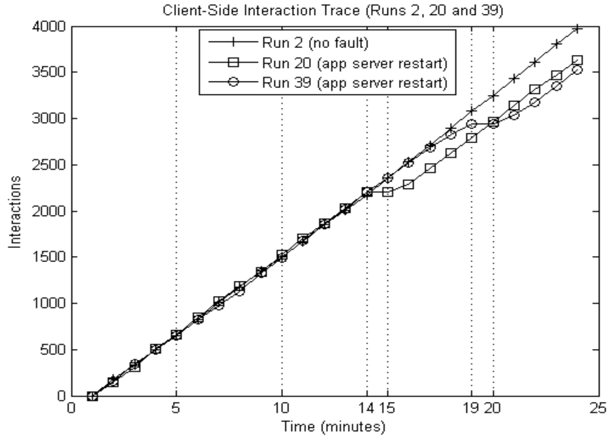

Figure 2. *Client-side Interaction Trace - Configuration B*

To evaluate the RAS-characteristics of the system in the presence of the memory leak, we use the SHARPE RAS-modeling and analysis tool [20] to create the basic 2-node, 2-parameter RAS-model shown in Figure 3. Table 1 lists the model's parameters.

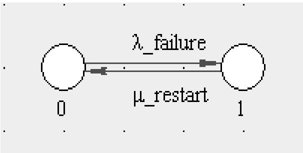

Figure 3. *Simple RAS Model*

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

Table 1. *RAS-Model Parameters – Configuration B*

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 [12]: $A = \frac{\frac{1}{\lambda}}{\frac{1}{\lambda}+\frac{1}{\mu}}$ 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, the system is capable of delivering two 9's of availability. Table 2 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 2. *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 3 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 3. *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 4, its parameters are listed in Table 4.

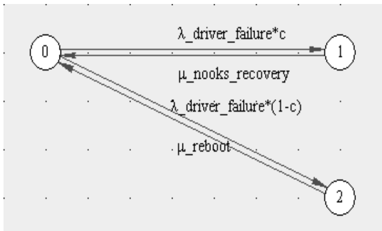

Figure 4. *RAS-Model of a system with imperfect repair*

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 attempt 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 minute 22 seconds

Table 4. *RAS-Model Parameters – Configuration D*

Figure 5 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 promote the system into another availability bracket while a 92% success rate reduces the expected

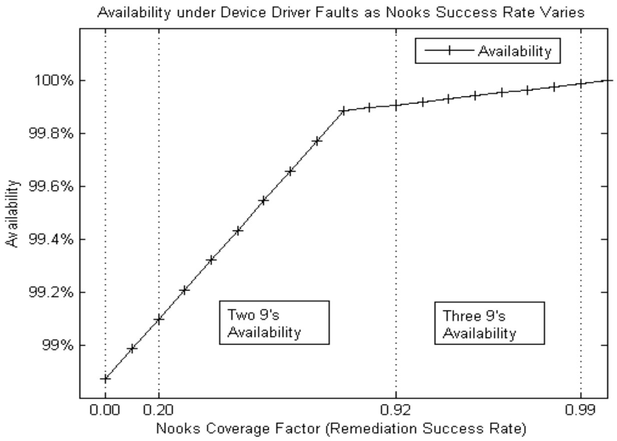

Figure 5. *Availability – Configuration D*

downtime and SLA penalties by two orders of magnitude (see Figure 5) [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 6.

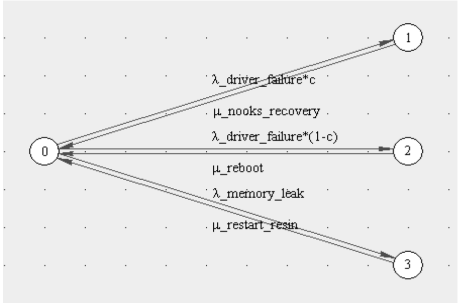

Figure 6. *Complete RAS-model – Configuration E*

Figure 7 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 preventative maintenance. For this to be an option we assume that the system's failure distribution is hypoexponential. We divide the system's lifetime into two stages, where the time spent in each state is exponentially distributed.

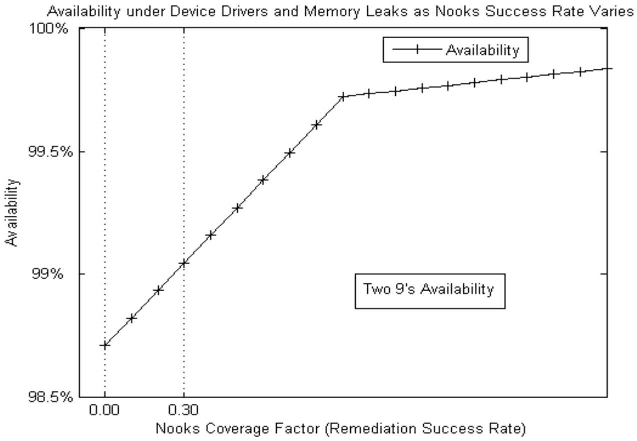

Figure 7. *Availability – Configuration E*

We use the RAS-model shown in Figure 8 in our analysis. Its parameters are listed in Table 5.

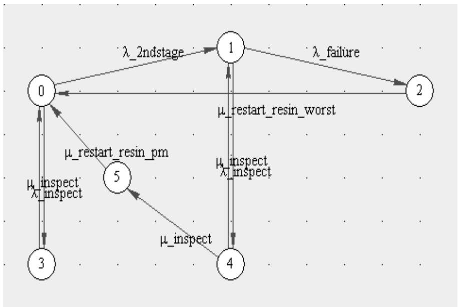

Figure 8. *Preventative Maintenance RAS-model*

Using these parameters we plot the graph shown in Figure 9, which shows the expected availability of the system as $\lambda_{inspect}$ varies. We see that performing a check 6 times every hour and performing preventative maintenance is expected to improve the system's availability; however, actually implementing this scheme and running more experiments is the only way to validate this model.

In summary, our analysis of existing remediations and a yet-to-be-added preventative maintenance scheme produce artifacts (measurements and models) that can validate or justify system (re)design decisions.

5 Related Work

The work most similar to ours is [7]. In this paper the authors build a RAS model to explore the expected impact of Memory Page Retirement (MPR) on hardware faults associated with failing memory modules on systems running Solaris 10. MPR removes a physical page of memory from use

[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.

S_0	an UP state, 1st stage of system lifetime
S_1	an UP state, 2nd stage of system lifetime
S_2	a DOWN state, application server is restarted
S_3	an UP state, free-memory inspection occurs during the 1st stage of the system's lifetime
S_4	an UP state, free-memory inspection occurs during the 2nd stage of the system's lifetime. A preventative restart is carried out returning the system to the first stage of its lifetime
S_5	a DOWN state, preventative restart occurs
$\lambda_{2ndstage}$	rate of transition into 2nd stage of its lifetime, once every six hours
$\lambda_{failure}$	rate of transition into low-memory condition state, once in either the 7th or 8th hour
$\mu_{restart\_resin\_worst}$	time to restart Resin under low-memory conditions, $\sim$47 seconds
$\lambda_{inspect}$	rate of free-memory trend-checks
$\mu_{inspect}$	time to conduct free-memory check, 21,627 microseconds
$\mu_{restart\_resin\_pm}$	best-case time to restart application, server 3,092 milliseconds

Table 5. *Preventative maintenance model parameters*

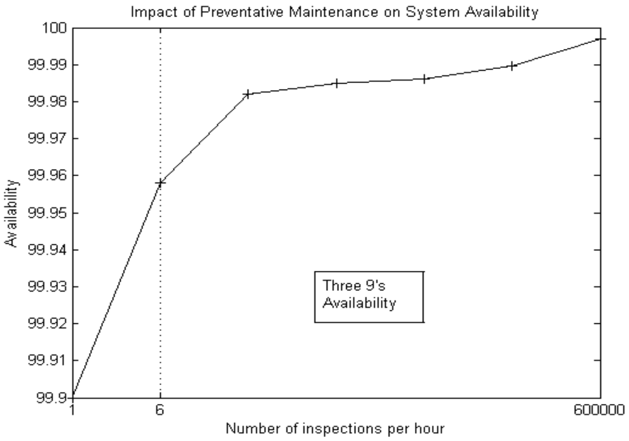

Figure 9. *Expected impact of preventative maintenance*

by the system in response to error correction code (ECC) errors associated with that page. Using their models the authors investigate the expected impact of MPR on yearly downtime, the number of service interruptions and the number of servicing visits due to hardware permanent faults. Unlike our experiments, which focus on software and rely on fault injection experiments to collect data, the authors focus on hardware failures and use field data from deployed low-end and mid-range server systems to build models.

In [6] the authors study the availability of the Sun Java System Application Server, Enterprise Edition 7. The authors use hierarchical Markov reward models to model and obtain average system availability estimates. In a distributed load-balanced deployment, including two application server instances, 2 pairs of Highly Available Databases (HADBs) – used as http session state stores – an Oracle database and a Sun Java System Directory Server, the authors induce faults concerned with whole-node removal to investigate the system's (session) fail-over and recovery mechanisms. Our experiments differ in the granularity of our fault-injection; rather than remove entire nodes, we focus on injecting faults in the individual components of a single node. Further, whereas we do not focus on evaluating remediation mechanisms that rely on whole-node redundancy or failover, RAS-modeling techniques can be adapted for this [12].

[9] describes the DBench-OLTP dependability benchmark. We differ from this work in our choice of metrics. The measures prescribed in the DBench-OLTP specification include but are not limited to: transactions per minute (tpmC), price per transaction ($/tpmC), availability from the system under test and remote terminal emulator points of view. We focus less on performance-related measures and present ways to analyze the impact of the system's remediation mechanisms on the system's reliability, availability and serviceability.

FAUMachine [22] (formerly UMLinux) is a virtualization platform supporting fault-injection. The faults that can be injected include, but are not limited to: bit flips in memory and CPU registers, block device failures and network failures. For our experiments we required more fine-grained control over the faults injected. Further, the faults that could be injected using FAUMachine would not appropriately exercise the remediation mechanisms of our target system.

Our work is complementary to the work done on robustness benchmarking [5] and fault-tolerant benchmarking [27]. However, we focus less on the robustness of individual component interfaces for our fault-injection and more on system recovery in the presence of component-level faults i.e. resource leaks or delays.

[1] conducts a study of availability and maintainability benchmarks using software RAID systems. In addition to studying availability from the end-user perspective as these authors do, we also include the use of mathematical models to assist in the analysis of existing and potential remediation-mechanisms.

[15] describes the System Recovery Benchmark. The authors propose measuring system recovery on a non-clustered standalone system. The focus of the work is on detailed measurements of system startup, restart and recovery events. Our work is complementary to this, relying on measuring startup, restart and recovery times at varying granularity. We consider these measurements at node-granularity as well as application/component granularity.

[2] describes work towards a self-healing benchmark. In our work we analyze the individual mechanisms that impact the quality of service metrics of interest. Our focus on how the system accomplishes healing and its relation to the high-level system goals, dictated by SLAs and policies.

6 Conclusions & Future Work

In this paper we use reliability, availability and serviceability (RAS) metrics and models, coupled with fault-injection experiments, to analyze the impact of self-healing mechanisms on these high-level (RAS) metrics of interest. We also highlight the versatility of these models by employing them to briefly study and design various styles of remediations, analyze the impact of failed remediations and identify sub-optimal remediations. Based on our experiments and the metrics obtained we conclude that RAS-models are reasonable, rigorous, analytical tools for evaluating self-healing systems and their mechanisms.

For future work, we are interested in conducting additional fault-injection experiments and analytical studies on different operating system platforms, including Solaris 10, which has been designed with a number of self-healing mechanisms [24, 21]. We will also continue our work developing practical fault-injection tools.

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

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

[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. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An empirical study of operating system errors. In *Symposium on Operating Systems Principles*, pages 73–88, 2001.

[5] J. DeVale. Measuring operating system robustness.

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

[7] Dong 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.

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

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

[10] Ji Zhu et al. R-Cubed: Rate, Robustness and Recovery An Availability Benchmark Framework. Technical Report SMLI TR-2002-109, Sun Microsystems, 2002.

[11] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer magazine*, pages 41–50, January 2003.

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

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

[14] J.-C. Laprie and B. Randell. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004.

[15] J. Mauro, J. Zhu, and I. Pramanick. The system recovery benchmark. In *10th IEEE Pacific Rim International Symposium on Dependable Computing*, 2004.

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

[17] Michael M. Swift et al. Improving the Reliability of Commodity Operating Systems. In *International Conference Symposium on Operating Systems Principles*, 2003.

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

[19] NIST. National institute of standards and technology (nist) website. http://www.nist.gov/.

[20] A. R. Sahner and S. K. Trivedi. Reliability modeling using sharpe. Technical report, Durham, NC, USA, 1986.

[21] M. Shapiro. Self-healing in modern operating systems. http://www.acmqueue.com/modules.php?name=Content&pa=showpage&pid=242.

[22] V. Sieh and K. Buchacker. Umlinux – a versatile swifi tool. In *Proceedings of the Fourth European Dependable Computing Conference*, 2002.

[23] SPEC. Standard performance evaluation corporation (spec) website. http://www.spec.org/.

[24] Sun Microsystems. Predictive self-healing in the solaris 10 operating system. http://www.sun.com/software/whitepapers/solaris10/self_healing.pdf.

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

[26] TPC. Transaction processing performance council (tpc) website. http://www.tpc.org/.

[27] T. K. Tsai, R. K. Iyer, and D. Jewitt. An approach towards benchmarking of fault-tolerant commercial systems. In *Symposium on Fault-Tolerant Computing*, pages 314–323, 1996.