Multi-perspective Evaluation of Self-Healing Systems using Simple Probabilistic Models

Rean Griffith, Gail Kaiser
Columbia University
{rg2023,kaiser}@cs.columbia.edu

Javier Alonso López
Universitat Politècnica de Catalunya
{alonso}@ac.upc.edu

Abstract

Quantifying the efficacy of self-healing systems is a challenging but important task, which has implications for increasing designer, operator and end-user confidence in these systems. During design system architects benefit from tools and techniques that enhance their understanding of the system, allowing them to reason about the tradeoffs of proposed or existing self-healing mechanisms and the overall effectiveness of the system as a result of different mechanism-compositions. At deployment time, system integrators and operators need to understand how the self-healing mechanisms work and how their operation impacts the system's reliability, availability and serviceability (RAS) in order to cope with any limitations of these mechanisms when the system is placed into production.

In this paper we construct an evaluation framework for self-healing systems around simple, yet powerful, probabilistic models that capture the behavior of the system's self-healing mechanisms from multiple perspectives (designer, operator, and end-user). We combine these analytical models with runtime fault-injection to study the operation of VM-Rejuv – a virtual machine based rejuvenation scheme for web-application servers. We use the results from the fault-injection experiments and model-analysis to reason about the efficacy of VM-Rejuv, its limitations and strategies for managing/mitigating these limitations in system-deployments. Whereas we use VM-Rejuv as the subject of our evaluation in this paper, our main contribution is a practical evaluation approach that can be generalized to other self-healing systems.

1. Introduction

Self-healing mechanisms are intended to improve the reliability, availability and serviceability (RAS) of a system by enabling it to automatically detect, diagnose and repair localized hardware and software problems [20]. However, the inclusion of recovery or repair mechanisms (self-healing mechanisms) is no guarantee that these mechanisms work well, are bug free, or that the failure modes and limitations of these mechanisms are well understood. The inadequate testing of recovery mechanisms and the unexpected/unintended negative side effects of recovery have resulted in a number of (in)famous failures, which have been discussed in previous work [6], [11], [28], [18]. The rigorous testing, analysis and validation of these mechanisms are important but sometimes overlooked steps in system-construction that would otherwise allow designers to better understand how these mechanisms work and identify their limitations.

The limitations of self-healing mechanisms can take many forms including, but not limited to: periods of vulnerability to successive faults (vulnerability windows), imperfect repair or recovery scenarios (e.g., instances where recovery fails, or service is only partially restored requiring additional rollback or compensation semantics to put the system back into some well understood state), and situations where the rate of failure or the sequence of failure events overwhelm the self-healing mechanisms available resulting in system instabilities.

System designers, operators and end-users have an interest in understanding the limitations of the recovery mechanisms available in the system. These limitations impact them in different ways, and as a result an evaluation framework for self-healing systems needs to accommodate their different perspectives. In addition to enhancing their understanding of the system, the insights about the limits of the recovery mechanisms obtained from an evaluation can be used to inform contingency plans for coping with these limitations in real deployments.

To assist designers and operators in system evaluations there are a number of well-studied modeling formalisms and associated analytical techniques that can be used to describe and reason about both system structure and behavior. Examples include, Markov Chains, Petri Nets, Stochastic Activity Networks (SANs), and Queuing Models ([21, 14, 31, 26]). In terms of practical tools, fault-injection has been accepted as a powerful tool for validating and evaluating recovery mechanisms in systems [39, 9] and a number of fault-injection strategies (and tools that use them) are available [16]. Note that while fault-injection is accepted as a powerful system-validation tool it is also accepted that fault-injection cannot predict actual availability or mean time between failures (MTBF) [40, 16]. However our goal in this paper is not to make absolute predictions about these measures, but rather to present a consistent framework for reasoning quantitatively about the limitations of recovery mechanisms and developing contingency plans that can address these limitations.

1.1 Contributions

The main contribution of our work is to demonstrate how an evaluation framework for self-healing systems can be constructed around simple probabilistic models that capture different evaluator-perspectives. We use the analytical tools and fault-injection tools to reason quantitatively about the limitations of a system's self-healing mechanisms and to inform strategies for mitigating these limitations. In this paper we create a simple analytical model of VM-Rejuv, a virtual machine based rejuvenation scheme for web-application servers, using Continuous Time Markov Chains (CTMCs). We identify RAS measures that can be used to quantify the efficacy of VM-Rejuv from the perspective of the designer, operator and end-user. We then use runtime fault-injection tools to inject memory leaks into the web-application servers deployed under VM-Rejuv and exercise VM-Rejuv's self-healing capabilities. The results from our fault-injection experiments are used to gather parameters for the analytical model, which in turn is used to identify and reason about potential limitations of VM-Rejuv and to propose strategies for managing/mitigating these limitations in system deployments.

The remainder of this paper is organized as follows. §2 details our case study – §2.1 presents background on software rejuvenation and an overview of the architecture and operation of VM-Rejuv. §2.2 describes our evaluation methodology and §2.3 describes our experiments and results – §3 discusses our results, §4 outlines related work and §5 summarizes our conclusions and future work.

2 Case Study: VM-Rejuv

2.1 Overview

In our case-study we model and experimentally evaluate the efficacy of VM-Rejuv – a prototype implementation of a virtual machine (VM) based software rejuvenation scheme for application servers and internet sites [32] developed at the Universitat Politècnica de Catalunya (UPC) in Barcelona.

Software rejuvenation is the concept of gracefully terminating an application and immediately restarting it in a clean internal state [17]. This technique has been implemented as a form of preventative/proactive maintenance in a number of systems, e.g., AT&T billing applications [17][1], telecommunications switching software [3], online transaction processing (OLTP) servers [8], middleware applications [5] and web/application-servers [22], as an approach to mitigate the effects of software aging – the degradation of the state of a software system, which may eventually lead to system performance degradation or crash/hang failure [1].

Strategies for rejuvenation can be divided into two classes: *time-based* rejuvenation and *prediction-based* rejuvenation [1]. With time-based rejuvenation state-restoration activities are preformed at regular deterministic intervals, whereas with prediction-based rejuvenation the time to rejuvenate is based on the collection and analysis of system data, e.g., resource metrics. State-restoration activities performed during rejuvenation may include one or more of: garbage collection, preemptive rollback, memory defragmentation, therapeutic reboots, flushing and/or reinitializing data structures [8].

VM-Rejuv employs a **prediction-based** rejuvenation strategy for mitigating the effects of software aging and transient failures on web/application-servers. Software aging and transient failures are detected through continuous monitoring of system data and performance metrics of the application-server; if some anomalous behavior is identified the system triggers an automatic rejuvenation action [32]. Rejuvenation actions in VM-Rejuv take the form of **preventative** application-server restarts.

To minimize the disruption to clients due to an application-server restart, VM-Rejuv employs redundancy and load-balancing. Web-application servers are deployed under VM-Rejuv in multiple virtual machines logically organized in a cluster. Figure 1 shows the architecture of a VM-Rejuv virtual machine cluster deployed on a single physical machine.

VM-Rejuv uses three virtual machines for a hosted web-application: one VM to run a software load-balancer (VM1), one VM to be the main/"active" application server and one VM to be a hot-standby replica of the main application server (VMs 2 and 3).

The first virtual machine, VM1, runs:

[1]The original proposal of the software rejuvenation technique by Huang et al.

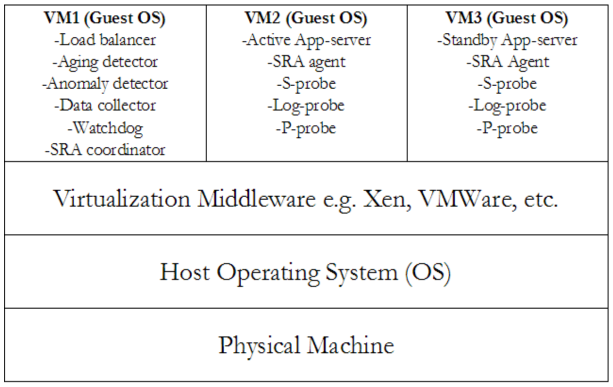

Figure 1. VM-Rejuv framework

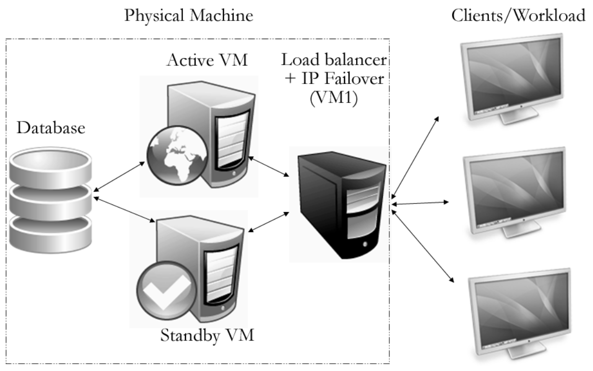

Figure 2. VM-Rejuv deployment[2]

- A load-balancer – the VM-Rejuv prototype uses Linux Virtual Server (LVS) as its load-balancer [38]. LVS is a layer-4 load-balancer, which provides IP-failover and a number of load-balancing policies (round-robin, weighted round-robin, etc.).
- An Aging detector – module for forecasting aging-related failures. In the current VM-Rejuv prototype the Aging detector uses simple threshold techniques concerned with memory utilization [32].
- An Anomaly detector – module that detects anomalies in VM2 and VM3 using threshold violations as indicators of anomalies, e.g., throughput falling below a preset threshold or response time exceeding a preset threshold (SLA violations).
- A Data collector – module that collects statistics from VMs 2 and 3 for analysis.
- A Watchdog – module that detects server outages. VM-Rejuv uses ldirectord, which is used to monitor and administer real servers in LVS clusters [29].
- Software Rejuvenation Agent (SRA) coordinator – module that directs SRAs on VMs 2 and 3 to initiate an application-server restart.

While virtual machines 2 and 3 run:

- The web-application server – the resource being load-balanced and periodically rejuvenated.
- Software rejuvenation agents – modules that initiate rejuvenation actions as directed by the SRA coordinator.
- A set of probes – modules that collect statistics from various sources including log files, (guest) operating system kernel (e.g., CPU utilization, memory usage, swap space, etc.) and application-server proxies (e.g., the P-probe module sits in front of the application-server collecting statistics on throughput and latency).

An example deployment of a web-application using VM-Rejuv is shown in Figure 2. During its operation, client requests to the web-application are routed by the LVS load-balancer on VM1 to the application server on the active VM, while the standby VM (and its application-server) remains ready but inactive as a hot replica until a rejuvenation is signaled by the SRA coordinator.

When a rejuvenation action is signaled, the active VM and standby VM switch roles. New client requests are routed to the application server on the standby VM (old standby VM marked as the "new" active VM); the application-server on the old active VM finishes processing any outstanding requests before the local SRA agent restarts the application server. The interval of time the old active VM spends processing client requests that are in-flight/outstanding when a rejuvenation is signaled is referred to as the *pre-rejuvenation delay-window*.

The use of redundancy in VM-Rejuv and coordinated switch-overs between the active VM and the standby VM support application-server restarts that minimize the loss of in-flight client-requests during rejuvenation. These elements combined with application-specific technologies like session migration/replication (e.g., as found in the Apache Tomcat web/application server [32]) allow rejuvenations to be performed without disrupting clients, which potentially improves the client-perceived availability of the web-application.

Deploying a web-application under a prediction-based rejuvenation scheme like VM-Rejuv has a number of implications for its reliability, availability and serviceability.

Rejuvenation activities can be used as preventative maintenance to avoid certain kinds of failures, e.g., memory-leaks as shown in [32]. The use of redundancy and IP failover allow clients to be shielded from the failure of the active VM and minimizes disruptions due to preventative restarts.

[2]Server icons by Fast Icon Studio (http://www.fasticon.com) designed by Dirceu Veiga. Client/workstation icons by Layered System Icons designed by BogdanGC (http://bogdangc.deviantart.com/). Database icon by DryIcons (http://dryicons.com).

These aspects of VM-Rejuv's operation potentially improve the web-application reliability, availability and serviceability. However, the efficacy of problem detection/prediction mechanisms, the frequency of rejuvenation actions, the success rate of rejuvenation actions, and the size of the pre-rejuvenation delay-window are all elements that can negatively affect the RAS properties of an application deployed under VM-Rejuv.

Problem detection/prediction mechanisms influence the rate at which rejuvenation actions are initiated. Imperfect detection/predictions can result in too many or too few rejuvenation actions.

Whereas too many rejuvenations may not disrupt clients (due to the redundancy and fail-over) time spent waiting to rejuvenate (the pre-rejuvenation delay-window) represents a period of vulnerability during which a failure of the active VM can affect clients. Further, frequent rejuvenations may put the system in a state where the active and standby VMs are constantly switching roles, indicating that the thresholds used to trigger rejuvenations may be inappropriate or may make the system unstable. Finally, rejuvenation actions may also fail, e.g., application servers can fail to restart or node-failover may be unsuccessful, in which case some other mechanism would need to be in place to rectify the situation. On the other hand, too few rejuvenations may result in failures/unplanned downtime, which could have been avoided and may indicate inadequate fault/failure coverage for the system.

In our evaluation of VM-Rejuv we wish to quantify the effects of; the rejuvenation frequency, the success rate of rejuvenation actions (node-failover and application-server-restart), and the size of the pre-rejuvenation delay-window on its reliability, availability and serviceability.

2.2 Evaluation Methodology

This section describes the six key elements of our evaluation; the system under test, the fault-model of interest, the relationship between faults and the remediations provided by VM-Rejuv, the micro-measurements (metrics collected from the remediation mechanisms), the macro-measurements (high-level measures of interest, e.g., facets of reliability, availability and serviceability, used to score the overall system) and the data collection infrastructure (metric collectors) used.

System under test. For the system under test we use the TPC-W web-application hosted on two Apache Tomcat web/application servers [36] under VM-Rejuv. Tomcat is a Java-based web/application server developed by The Apache Foundation. Apache Tomcat is used as the web-application server in the VM-Rejuv experiments since a P-probe designed specifically for communicating performance statistics from Tomcat to the SRA coordinator is included in the VM-Rejuv prototype [3].

Fault model. VM-Rejuv's main detection mechanisms use the violation of response time and/or throughput thresholds to indicate that a rejuvenation action is required. We identify faults that can be used to trigger these detection mechanisms. Severe memory leaks affect both throughput and response time, degrading these performance metrics [32] in application servers. We use Kheiron/JVM [13][4] to inject memory leaks into the web-application servers deployed under VM-Rejuv.

Fault-remediation relationship. VM-Rejuv initiates a node-failover and signals a rejuvenation (application-server restart) action in response to throughput or response time violations or application server crashes.

Micro-measurements. For micro-measurements we collect metrics on: the time for node-failover, the frequency of rejuvenation actions, the success of a rejuvenation, the size of the pre-rejuvenation delay-window, and application-server restart, server-side estimates of request throughput, and response time client-side goodput via instrumenting parts of VM-Rejuv (specifically the SRA agent coordinator and the SRA agents), and parsing application-server logs and parsing TPC-W client logs (client-side goodput is reported as the number of web-interactions performed by TPC-W clients).

Macro-measurements. For macro-measurements we use the seven node, six parameter scoring model shown in Figure 3, with parameter descriptions in Table 2, to quantify facets of reliability, availability and serviceability for VM-Rejuv deployments.

The structure of our model is designed to focus on the following high-level activities in VM-Rejuv; the rejuvenation cycle, active VM loss during normal operation, and active VM loss during rejuvenation (Table 1).

Rejuvenation cycle	$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_0$
Active VM loss during normal operation	$S_0 \rightarrow S_3 \rightarrow S_4 \rightarrow S_2 \rightarrow S_0$
Active VM loss during rejuvenation	$S_0 \rightarrow S_1 \rightarrow S_5 \rightarrow S_6 \rightarrow S_2 \rightarrow S_0$ $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_5 \rightarrow S_6 \rightarrow S_2 \rightarrow S_0$

Table 1. VM-Rejuv RAS model structure

In constructing this model we make the following simplifying assumptions. All state transitions in VM-Rejuv are ex-

[3]The Tomcat P-probe is a Java class that is installed as a filter [34] in the pipeline that processes requests received by the application-server.

[4]Kheiron/JVM uses bytecode rewriting and the Java Virtual Machine Tool Interface (JVMTI)[35] to interact with running Java programs.

ponentially distributed, and the rate of failure during normal operation ($S_0 \rightarrow S_3$) and the rate of failure during rejuvenation ($S_1 \rightarrow S_5$ and $S_2 \rightarrow S_5$) is kept the same[5].

The facets of reliability, availability and serviceability that we quantify using our RAS model include:

- Reliability – frequency of active VM failures during rejuvenation (transitions from S_1 to S_5 and from S_2 to S_5).
- Availability – basic steady state availability (proportion of time spent in S_0, which excludes time spent rejuvenating) and tolerance availability [15] (proportion of time spent in S_0, S_1 and S_2, which includes time spent rejuvenating). We use these two facets of availability to distinguish between the administrator's perspective and the client's perspective on system availability respectively[6]. This differentiation is discussed at the end of §2.3 when we present our results.
- Serviceability – mean time to system restoration. We also quantify this from the perspectives of both the administrator and the client.

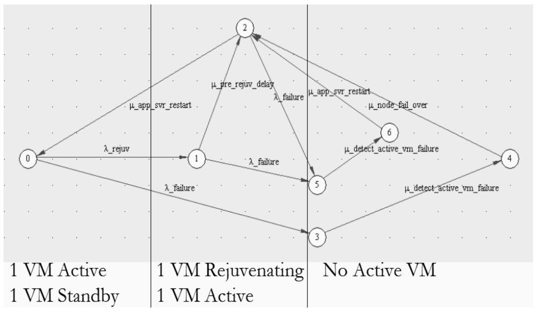

Figure 3. VM-Rejuv RAS model

Workload and metric collectors. Scripts that parse TPC-W client logs, Tomcat logs, SRA coordinator logs and SRA agent logs are used to gather micro-measurement data.

2.3 VM-Rejuv Evaluation

We create a test deployment of VM-Rejuv consisting of three virtual machines co-located on a single physical machine. VM1 is configured with 640 MB RAM, 1GB swap, 2 virtual CPUs and an 8GB harddisk. VM2 and VM3 are each configured with 384 MB RAM, 512 MB swap, 2 virtual CPUs and 8GB harddisks. All three VMs run Centos 5.0 with a Linux 2.6.18-8.el5 SMP kernel.

[5]The memoryless property of the CTMC does not account for the fact that in practice the rate of failure in S_2 may be lower after a failure occurs during S_1 than it would be if no failure occurred in S_1.

[6]Arguably system designers are interested in both of these metrics.

S_0	state where active VM services requests and standby remains VM ready
S_1	state where VM-Rejuv prepares to rejuvenate the active VM and the standby VM becomes the new active VM servicing new client requests
S_2	state where old active VM is ready to rejuvenate
S_3	state where the active VM has failed during normal operation
S_4	state where the failure of the active VM has been detected
S_5	state where the new active VM (the old standby VM) has failed while the old active VM is rejuvenating
S_6	state where the failure of the active VM during rejuvenation has been detected
λ_{rejuv}	rate of rejuvenation
$\lambda_{failure}$	forced/induced rate of failure of the active VM
$\mu_{pre\_rejuv\_delay}$	size of pre-rejuvenation delay-window
$\mu_{app\_svr\_restart}$	mean time to restart/rejuvenate the application server on the active VM
$\mu_{detect\_active\_vm\_failure}$	mean time to detect that the active VM has failed/crashed
$\mu_{node\_fail\_over}$	mean time to failover to the standby VM

Table 2. VM-Rejuv RAS model

To enable LVS load-balancing, the network interface on VM1 is configured with two IP addresses, one public IP address and one private IP address (192.168.1.xxx). Our LVS configuration is based on LVS-NAT [37]. VM2 and VM3 are configured with private IP addresses only (192.168.1.xxx). VM2 and VM3 can route to VM1 only, whereas VM1 can route to VMs 2 and 3 and the internet.

The physical machine hosting the VMs is configured with 2 GB RAM, 2 GB swap, an Intel Core Duo E6750 Processor (2.67 GHz) and a 228 GB harddisk running Windows XP Media Center Edition SP2.

The VM-Rejuv configuration used in our experiments is identical to that shown in Figure 2 except that the database is installed on VM1 (the VM with the load-balancer). We install Apache Tomcat v5.5.20 and Sun Microsystems' Hotspot Java Virtual Machine v1.5 on VMs 2 and 3 as well as instances of the TPC-W web-application. We use the MySQL 5.0.27 database server to store the TPC-W web-application data, and this is installed on VM1. The TPC-W web-application instances on VMs 2 and 3 are configured to access the database server on VM1. The LVS tools (IPVS v1.2.1 and ipvsadm v1.24) are installed on VM1 [37]. The following VM-Rejuv components are installed on the three VMs: the SRA coordinator, ldirectord watchdog, response time and throughput monitors are installed on VM1 while the SRA agents are installed on VM2 and VM3.

The VM-Rejuv prototype works with the Apache Tomcat web/application server [32]. Whereas the components of VM-Rejuv are written in Java, operations such as rejuvenating application servers and updating LVS tables for failover are facilitated by shell scripts called from Java using the java.lang.Runtime::exec() API. To restart/rejuvenate Tomcat, VM-Rejuv's SRA agents invoke the shutdown.sh and startup.sh scripts in the bin directory under the Tomcat installation directory, while updates to the LVS table to designate the new active VM are performed via calls to the Linux Virtual Server Administration utility, ipvsadm.

We simulate a client load of 50 TPC-W clients using the Shopping Mix[24] as their web-interaction strategy.

During 15 failure-free runs, each lasting 22 minutes, the average number of client-side interactions recorded is 7745.2 ± 748.9 (Figure 4). Figures 5 and 6 show a 10 minute sample of the throughput and response time data reported by VM probes during one of our failure-free runs. From our failure-free runs the average throughput is ∼13 requests per second and the average response time is ∼11 ms. We use the server-side throughput and response time numbers reported to set the SLA violation thresholds for VM-Rejuv and inject faults that result in the violation of these thresholds, triggering rejuvenation actions so we can estimate the parameters for our scoring model.

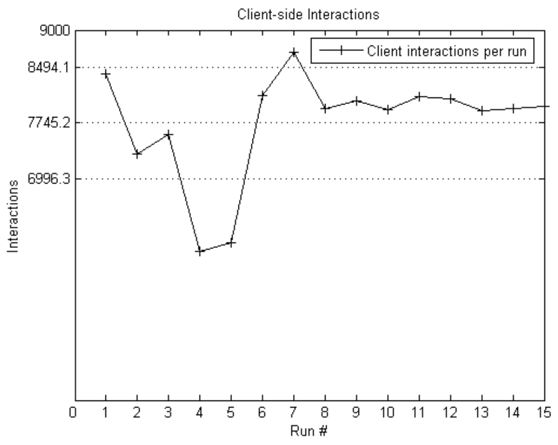

Figure 4. VM-Rejuv baseline performance (client-side interactions)

To estimate the size of the rejuvenation window, we set VM-Rejuv's response time violation threshold at mean response time (11 ms) and re-run the workload of 50 clients. VM-Rejuv triggers rejuvenations after four consecutive SLA violations. During three 22 minute runs we observe an average of 4 rejuvenation actions per run. During rejuvenation actions, the mean failover time is 25.62 msecs ± 3.46 msecs (see Figure 7) with a mean pre-rejuvenation delay window size of 14,769 msecs ± 5,420 msecs (see Figure 8).

In our fault-injection experiments we subject both Tomcat application servers deployed under VM-Rejuv to memory leaks that result in resource exhaustion within 5.53 minutes[7] (332.017 seconds) of running the 50 client TPC-W workload. We set VM-Rejuv's response time violation threshold to the mean response time of the failure-free runs (11 ms) and measure the frequency of rejuvenations, and the size of the pre-rejuvenation delay window. Introducing memory leaks in the Tomcat application servers increases the response time and delays the rejuvenation of the old active

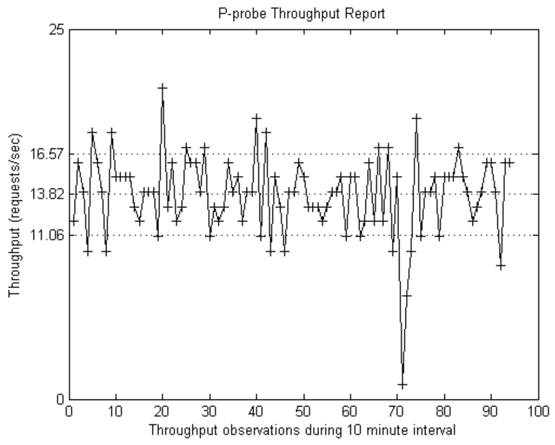

Figure 5. VM-Rejuv baseline throughput

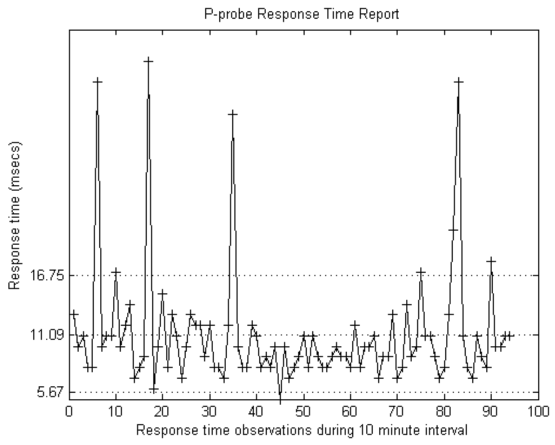

Figure 6. VM-Rejuv baseline response time

[6]Server icons by Fast Icon Studio (http://www.fasticon.com) designed by Dirceu Veiga. Client/workstation icons by Layered System Icons designed by BogdanGC (http://bogdangc.deviantart.com/).

[7]Whereas we acknowledge that a system that runs out of memory every 5.53 minutes would be quickly redesigned or abandoned by its user base (see §3 for more discussion on this), our goal is to evaluate VM-Rejuv under an aggressive memory-leak scenario.

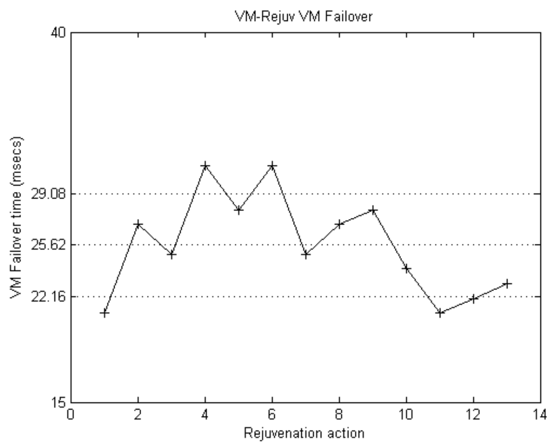

Figure 7. VM-Rejuv VM failover time

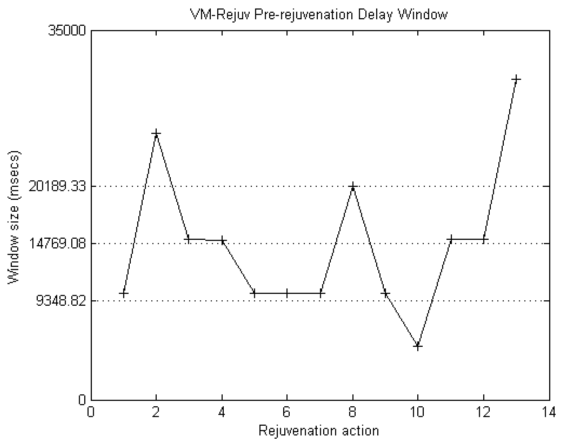

Figure 8. VM-Rejuv rejuvenation window size (50 clients)

VM after the standby server is brought online, since the old active VM must service outstanding requests before it rejuvenates. Table 3 summarizes the results from five 22 minute memory-leak experiments.

Using a mean rejuvenation interval of 154.06 seconds, mean rejuvenation window size of 27,401.52 msecs and mean failover time of 28.94 msecs, we score the VM-Rejuv deployment using the RAS model in Figure 3. The mean time to restart Tomcat during the memory leak experiments is 3 seconds and the mean time to detect a server outage (via the ldirectord watchdog) is 5 seconds.

The steady-state probabilities of the VM-Rejuv model are shown in Table 4 and model analysis results are shown in Table 5.

Using the scoring model we can estimate the number of active VM failures (F_{avf}) expected during rejuvenation actions per day, i.e., the frequency of transitions from S_1 to S_5 ($F_{S_1 \rightarrow S_5}$) plus the frequency of transitions from S_2 to S_5 ($F_{S_2 \rightarrow S_5}$). This we estimate at 41 per day under the failure conditions used in our experiments (1 memory-leak failure every 5.53 minutes).

Run #	Rejuvenation actions	Rejuvenation interval (secs)	Failover time (msecs)	Pre-rejuvenation delay window (msecs)
1	8	155.47	33.88	34,657.63
2	9	142.13	31.63	18,321.38
3	6	155.76	27.20	16,175.60
4	7	149.08	24.71	37,538.57
5	8	167.86	27.29	30,314.43
Avg	**7.6**	**154.06**	**28.94**	**27,401.52**

Table 3. VM-Rejuv subjected to memory leaks

π_0	0.824673
π_1	0.135495
π_2	0.023510
π_3	0.012419
π_4	0.000072
π_5	0.002395
π_6	0.001437

Table 4. VM-Rejuv steady state probabilities – memleak scenario

From the steady-state probabilities of the model we estimate that the deployment spends ~82% of the time in its normal operating mode/configuration, π_0, and ~16% of its time rejuvenating ($\pi_1 + \pi_2$). While rejuvenations are taking place client-requests are serviced by the standby VM; as a result the system would be considered UP from the client's perspective in states $\{S_0, S_1, S_2\}$ – $UP_{client} = 1416.5$ minutes per day (98.37%) and DOWN 23.5 minutes per day (1.63%). Administrators on the other hand may consider the system to be UP if it is in state S_0 since states S_1 and S_2 represent a window of vulnerability. From the administrator's perspective the system is $UP_{admin} = 1187.5$ minutes per day (82.47%) and DOWN 252.5 minutes per day (17.53%), of which 229 minutes are spent performing rejuvenation actions.

Similarly, the mean time to system restoration can be quantified from the perspective of the client and the administrator. For the client, this is the mean time to restore the system to a state in $\{S_0, S_1, S_2\}$, $MTTSR_{client} = 5,509$ msecs, whereas for the administrator this is the mean time to restore the system to S_0, $MTTSR_{admin} = 22,373$ msecs.

In state S_1 clients still connected to the old active VM may experience some performance degradation and even lose requests if the degree of resource depletion on the old active VM is so severe that it cannot clear its backlog before the other VM needs rejuvenating. Further, increasing the size of the pre-rejuvenation delay window (either through missing rejuvenation opportunities or imperfect prediction) increases the time spent in S_1 (waiting for the backlog to clear

Measure	Metrics	Results
Reliability	Frequency of active VM failures during rejuvenation per day $F_{avf} = F_{S_1 \to S_5} + F_{S_2 \to S_5}$	41.377455
Availability	Basic steady-state availability ($UP_{admin} = \{S_0\}$)	0.824673
	Tolerance availability ($UP_{client} = \{S_0, S_1, S_2\}$)	0.983678
Serviceability	Mean-time to system restoration ($UP_{admin} = \{S_0\}$)	22,373 msecs
	Mean-time to system restoration ($UP_{client} = \{S_0, S_1, S_2\}$)	5,509 msecs

Table 5. Summary of VM-Rejuv RAS model analysis

at the resource-depleted old active VM), which places the system in a state where it is vulnerable to a failure of the current active VM.

3 Discussion

3.1 Model Sensitivity

In our evaluation of VM-Rejuv we build an RAS model of a web-application deployed under VM-Rejuv and we use an aggressive failure scenario (memory leak) to exercise its remediation mechanisms and estimate parameters for the RAS model. One question to consider, is the sensitivity of our analytical results to changes in the parameter estimates. Table 6 shows how the RAS measures calculated for VM-Rejuv are expected to change as the rate of resource exhaustion ($\lambda_{failure}$) is varied. In our sensitivity analysis we focus on varying the rate of failure for the following reason. The rate of resource exhaustion chosen for our experiments is overly aggressive; whereas this allows us to stress the self-healing mechanisms of VM-Rejuv, a system that actually runs out of resources within 5 minutes would likely be discarded and re-implemented. As a result, we use longer time horizons for resource exhaustion related failures in our sensitivity analysis (Table 6).

$\lambda_{failure}$	1 hr	1 day	1 week
F_{avf} per day	3.94087	0.16479	0.02354
UP_{client}	0.99847	0.99994	0.99999
UP_{admin}	0.83427	0.83515	0.83518
$MTTSR_{client}$	5,517 ms	5,518 ms	5,518 ms
$MTTSR_{admin}$	29,349 ms	30,355 ms	30,394 ms

Table 6. Sensitivity of VM-Rejuv RAS model to variations in resource exhaustion rates ($\lambda_{failure}$)

A second question to consider is the impact of the rate of rejuvenation on VM-Rejuv deployments. To reason about this, we look at the steady state probability of being in state S_3, this state captures resource exhaustion failures that occur during normal operation. Intuitively we expect that early/frequent rejuvenation actions reduce the total time spent in S_3 (lower steady-state probability) whereas waiting longer to rejuvenate increases the total time spent in S_3 (higher steady-state probability) as shown in Figure 9.

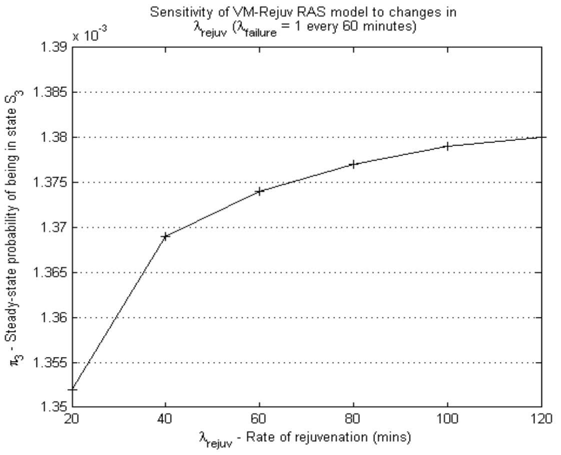

Figure 9. Sensitivity of VM-Rejuv RAS model to variations in the rate of rejuvenation λ_{rejuv}

3.2 Possible Limitations of VM-Rejuv

In VM-Rejuv the combination of redundancy, node-failover and timely rejuvenations greatly improves system availability. Whereas rejuvenation is expected to take place in a manner that is transparent to the end-user, we identify two possible limitations of a VM-Rejuv deployment along with possible mitigations for each.

Performance degradation during rejuvenation. Whereas redundancy and fail-over allows users to be switched from one server to another without interruption to their workflow, there may be performance penalties for doing so. For example, switching between servers (i.e., changing server roles) may lead to some performance perturbations as the new server warms up to the load (machine affinity effects). Reasoning about any such degradation can be done by converting our RAS model into a Markov Reward Network and assigning different throughput values (for example) to states $S_{normal} = \{S_0\}$ and $S_{rejuvenating} = \{S_1, S_2\}$. Further, in a real deployment, close monitoring of the variation in performance during rejuvenation will allow operators to detect whether rejuvenation actions adversely affect users. Any performance perturbations during rejuvenation will be exacerbated by excessive rejuvenation actions.

System degradation/vulnerability due to too few rejuvenations or failed rejuvenations. Too few rejuvenations can adversely affect the duration of rejuvenation cycles (more specifically they can increase the size of the pre-rejuvenation delay window). Short pre-rejuvenation delays imply that the server to be rejuvenated, quickly clears its backlog of outstanding requests. To the extent that this occurs, users are less likely to be affected by any interruptions. However, inaccurate predictions of when to rejuvenate can result in resource depletion so severe that the rejuvenating server cannot process its outstanding workload. Tracking the "drain-rate" and request-queue length of the rejuvenating server and employing a threshold on waiting for outstanding requests to complete can be used to reduce the amount of time the system/deployment is without a reserve node to failover to.

3.3 Improving the evaluation

Using multiple models. In this paper we present a simple model of VM-Rejuv based on CTMCs, however, we neither claim that ours is the only possible model of the system nor that CTMCs are the best modeling formalism to use. Rather, we expect there to be multiple models, some of which may be based on more sophisticated formalisms, e.g., Petri Nets, SANs, etc. Further, these models may focus on the entire system or on specific sub-systems or behavioral characteristics of the system depending on the measures of interest. Multiple models can then be composed and included as part of an in-depth system-evaluation. Finally, whereas we created a model of VM-Rejuv, other models of VM-Rejuv (or some other system of interest to evaluators) could originate from system vendors and/or the research community as has occurred with TCP [27].

4 Related Work

The analytical tools – Continuous Time Markov Chains (CTMCs) – and techniques for their analysis have been well studied and used by others to study many aspects of computing system behavior. For example, Markov chains have been used in the study and analysis of dependable and fault-tolerant systems and the techniques used to realize them. Examples include analyses of RAID (Redundant Arrays of Inexpensive Disks) [23], telecommunication systems [21] and Memory Page Retirement (MPR) in Solaris 10 [10]. They have also been used in the study of software aging [4] and in evaluating the efficacy of preventative maintenance (software rejuvenation).

[14], [21] and [26] provide a rigorous discussion of the mathematical principles (probability theory and queuing theory) underlying Markov chains and Markov reward networks as well as techniques for their analysis and solution.

[19] and [2] discuss techniques for the computationally tractable analysis and solution of Markov models. These techniques are available in the SHARPE [30] modeling tool, which we use in the construction and analysis of the RAS model for VM-Rejuv.

Performability [25] provides unified measures for considering the performance and reliability of systems together. Markov reward networks [14] have been used as a formalism for establishing this link between the performance of a system and its reliability. Other formalisms used in performability analysis include Stochastic Petri Nets (SPNs) [21] and Stochastic Activity Networks (SANs) [31], which are both built on top of Markov chains. SPNs and SANs allow for more detailed and sophisticated modeling of a system's operation, e.g., modeling concurrent activities in a system.

[7] describes work towards a self-healing benchmark. The authors identify a number of challenges to benchmarking self-healing capabilities including: quantifying healing effectiveness (identifying different metrics to quantify the impact of disturbances), accounting for incomplete healing and accounting for healing specific resources (spare disks, hot standbys, etc.). In our evaluation of VM-Rejuv we use RAS models based on Markov Chains to link low-level metrics from the mechanism (rejuvenation rates, pre-rejuvenation delay window sizes, etc.) to different facets of reliability, availability and serviceability.

5 Conclusions and Future Work

In this paper we construct an evaluation framework for VM-Rejuv using simple probabilistic models (CTMCs) and run-time fault-injection tools. We use our model and experimental results to reason about the efficacy of VM-Rejuv from the perspective of the designer, operator and end-user, identify its limitations and discuss possible mitigation strategies for addressing these limitations.

For future work we are packaging and integrating these models into system monitoring/management/analysis tools. To this end, we are encouraged by early successes [12] replicating our VM-Rejuv evaluation and analysis in the StackSafe Test Center [33], a pre-production staging, testing and analysis platform targeted at IT Operations teams[8].

6 Acknowledgments

The Programming Systems Laboratory is funded in part by NSF CNS-0717544, CNS-0627473 and CNS-0426623, and NIH 1 U54 CA121852-01A1. This work has been supported by the Spanish Ministry of Education and Science (projects TIN2007-60625).

[8]The TestCenter establishes a virtualized sandbox where IT Operations staff can test and analyze systems in a representative production environment.

References

[1] A comprehensive model for software rejuvenation. *IEEE Trans. Dependable Secur. Comput.*, 2(2):124–137, 2005. Kalyanaraman Vaidyanathan and Kishor S. Trivedi.

[2] A. Goyal and S.S. Lavenberg and K.S. Trivedi. Probabilistic modeling of computer system availability. In *Annals of Operation Research*, pages 285–306, 1987.

[3] A. Avritzer and E. J. Weyuker. Monitoring smoothly degrading systems for increased dependability. *Empirical Softw. Engg.*, 2(1):59–77, 1997.

[4] Y. Bao, X. Sun, and K. S. Trivedi. A workload-based analysis of software aging, and rejuvenation. *IEEE Transactions on Reliability*, 54(3):541–548, 2005.

[5] T. Boyd and P. Dasgupta. Preemptive module replacement using the virtualizaing operating system realizing multi-dimensional software adaptation. citeseer.ist.psu.edu/boyd02preemptive.html, 2002.

[6] P. Broadwell. Fig: A prototype tool for online verification of recovery mechanisms. In *In Workshop on Self-Healing, Adaptive and Self-Managed Systems*, 2002.

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

[8] K. Cassidy, K. Gross, and A. Malekpour. Advanced pattern recognition for detection of complex software aging phenomena in online transaction processing servers. *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 478–482, 2002.

[9] J. Clark and D. Pradhan. Fault injection: a method for validating computer-system dependability. *Computer*, 28(6):47–56, Jun 1995.

[10] 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.

[11] J. R. Garman. The "bug" heard 'round the world: discussion of the software problem which delayed the first shuttle orbital flight. *SIGSOFT Softw. Eng. Notes*, 6(5):3–10, 1981.

[12] R. Griffith. *The 7U Evaluation Method: Evaluating Software Systems via Runtime Fault-Injection and Reliability, Availability and Serviceability (RAS) Metrics and Models*. PhD thesis, Columbia University, 2008.

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

[14] Gunter Bolch and Stefan Greiner and Herman de Meer and Kishor S. Trivedi. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications 2nd Edition*. Wiley, 2006.

[15] D. I. Heimann, N. Mittal, and K. S. Trivedi. Availability and reliability modeling for computer systems. pages 175–233, 1990.

[16] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[17] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuventation: Analysis, module and applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing Pasadena CA June 1995*, pages 381 – 390, 1995.

[18] E. C. Jr., Z. Ge, V. Misra, and D. Towsley. Network resilience: Exploring cascading failures within bgp. In *Allerton Conference on Communication, Control and Computing*, October 2002.

[19] H. Kantz and K. S. Trivedi. Reliability modeling of the mars system: A case study in the use of different tools and techniques. In *PNPM*, pages 268–277, 1991.

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

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

[22] L. Li, K. Vaidyanathan, and K. Trivedi. An approach for estimation of software aging in a web server. *Empirical Software Engineering, 2002. Proceedings. 2002 International Symposium n*, pages 91–100, 2002.

[23] M. Malhotra and K. S. Trivedi. Reliability analysis of redundant arrays of inexpensive disks. *J. Parallel Distrib. Comput.*, 17(1-2):146–151, 1993.

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

[25] J. Meyer. On evaluating the performability of degradable computing systems. *IEEE Transactions on Computers*, 29(8):720–731, 1980.

[26] M.H.A. Davis. *Markov Models and Optimization*. Chapman & Hall, 1993.

[27] J. Mo, R. J. La, V. Anantharam, and J. Walr. Analysis and comparison of tcp reno and vegas. In *In Proceedings of IEEE Infocom*, pages 1556–1563, 1999.

[28] C. Perrow. *Normal Accidents: Living with High-Risk Technologies*. Princeton University Press, 1984.

[29] J. Rief and S. Horman. ldirectord. http://www.vergenet.net/linux/ldirectord/, 1999.

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

[31] W. H. Sanders and J. F. Meyer. Stochastic activity networks: formal definitions and concepts. pages 315–343, 2002.

[32] L. Silva, J. Alonso, P. Silva, J. Torres, and A. Andrzejak. Using virtualization to improve software rejuvenation. 2007.

[33] StackSafe Inc. Improve Business Uptime and Resiliency through a New Model for Software Infrastructure Testing by IT Operations. http://www.stacksafe.com/uploads/PDFs/StackSafe_White_Paper.pdf, 2007.

[34] Sun Microsystems. The essentials of filters. http://java.sun.com/products/servlet/Filters.html, 2001.

[35] Sun Microsystems. The JVM Tool Interface Version 1.0. http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html, 2004.

[36] The Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org/.

[37] The Linux Virtual Server Project. The Linux Virtual Server Project. http://www.austintek.com/LVS/LVS-HOWTO/mini-HOWTO/LVS-mini-HOWTO.html, 2001.

[38] The Linux Virtual Server Project. The Linux Virtual Server Project. http://www.linuxvirtualserver.org/, 2002.

[39] 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.

[40] D. Wilson, B. Murphy, and L. Spainhower. Progress on defining standardized classes for comparing the dependability of computer systems. citeseer.ist.psu.edu/wilson02progress.html, 2002.