Using Virtualization to Improve Software Rejuvenation

Luis Moura Silva, Javier Alonso, and Jordi Torres

Abstract—In this paper, we present an approach for software rejuvenation based on automated self-healing techniques that can be easily applied to off-the-shelf Application Servers. Software aging and transient failures are detected through continuous monitoring of system data and performability metrics of the application server. If some anomalous behavior is identified, the system triggers an automatic rejuvenation action. This self-healing scheme is meant to disrupt the running service for a minimal amount of time, achieving zero downtime in most cases. In our scheme, we exploit the usage of virtualization to optimize the self-recovery actions. The techniques described in this paper have been tested with a set of open-source Linux tools and the XEN virtualization middleware. We conducted an experimental study with two application benchmarks (Tomcat/Axis and TPC-W). Our results demonstrate that virtualization can be extremely helpful for fail-over and software rejuvenation in the occurrence of transient failures and software aging.

Index Terms—Software rejuvenation, software aging, virtualization, self-healing.

1 Introduction

Availability of business-critical application servers is an issue of paramount importance that has received special attention from the industry and academia in the last decades. According to [1], the cost of downtime per hour can go from 100,000 for online stores up to 6 million dollars for online brokerage services. The industry has adopted several clustering techniques [2] and today most business-critical servers apply some sort of server redundancy, load balancers, and fail-over techniques. Those techniques work quite well to recover from application crashes. The latest trend goes toward the development of self-healing techniques [3] that would automate the recovery procedures and prevent the occurrence of unplanned failures, whenever possible.

The idea behind this paper is to develop further the concept of software rejuvenation [4]. This technique has been widely used to avoid the occurrence of unplanned failures, mainly due to the phenomenon of software aging. The term software aging describes the progressive degradation of the running software that may lead to system crashes or undesired hang-ups [4]. It is likely to be found in any type of software that has some complexity, but it is particularly troublesome in long-running applications. It is not only a problem for desktop operating systems: it has been observed in telecommunication systems [5], Web servers [6], [7], enterprise clusters [8], OLTP systems [9], and spacecraft systems [10]. This problem has even been reported in military systems [11] with severe consequences such as loss of lives.

There are several commercial tools that help to identify some sources of memory leaks during the development phase [12], [13]. However, not all the faults can be easily spotted during the final testing phase. And those tools cannot work in third-party software packages when there is no access to the source code. This means that existing production systems have to deal with the problem of software aging while in production stage. So, in a wide number of cases the only choice is to apply some sort of software rejuvenation [4].

Two basic approaches for rejuvenation have been proposed in the literature: time-based and proactive rejuvenation. Time-based rejuvenation is widely used today in some real production systems, such as Web servers [14], [15]. Proactive rejuvenation has been studied in [8], [9], [14], [15], [16], [17], [18], [19], [20], [21]. It is widely understood that this second technique provides better results than the previous one, resulting in higher availability and lower costs. Some recent experimental studies have proved that rejuvenation can be a very effective technique to avoid failures even when it is known that the underlying middleware [21] or the protocol stack [22] suffer from clear memory leaks.

Several studies published in the literature tried to define some modeling techniques to find the optimal time for rejuvenation [14], [15], [16], [23]. The ROC project from Stanford [24] presented the concept of microrebooting to reduce the rejuvenation overhead: instead of applying restarts at the application- or system-level, they just reboot a very small set of components. Their main goal was to decrease the Mean-Time-To-Repair (MTTR) of the applications. The published results were quite promising [25]: They were able to decrease the time-to-repair by two orders of magnitude. In

- *L.M. Silva is with the Departamento de Engenharia Informática, University of Coimbra, Polo II, 3030 Coimbra, Portugal. E-mail: luis@dei.uc.pt.*
- *J. Alonso and J. Torres are with the Universitat Politècnica de Catalunya, UPC Campus Nord, C6-207, Jordi Girona 1-3, 08034 Barcelona. E-mail: {alonso, torres}@ac.upc.edu.*

Manuscript received 1 June 2008; revised 22 Dec. 2008; accepted 23 Feb. 2009; published online 24 July 2009.
Recommended for acceptance by D.R. Avresky, H. Prokop, and D.C. Verma. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2008-06-0262.
Digital Object Identifier no. 10.1109/TC.2009.119.

[26], it was well explained that it is more valuable to approach the goal of 100 percent availability by reducing the MTTR instead of paying efforts to increase the Mean-Time-Between-Failures (MTBF). If we cut down the MTTR, we can mitigate the impact of an outage to the end user, and this fact is of utmost importance for the Internet applications.

Driven by the results of the ROC project and by the goal of decreasing as much as possible the MTTR of recovery techniques, we decided to set up a project to devise a rejuvenation mechanism that could be applied in off-the-shelf Application Servers without reengineering the applications or the middleware. These rejuvenation mechanisms should minimize the MTTR of the applications.

In this context, we decided to make use of virtualization technology [27], [28] since it can be a good recipe to optimize the process of software rejuvenation. Encouraged by this concept, we have done a prototype implementation of our VM-based rejuvenation approach followed by an experimental study. The experimental results are presented in this paper.

The rest of the paper is organized as follows: Section 2 elaborates a bit further on our rationale for a new scheme of software rejuvenation; Section 3 describes our software rejuvenation and self-healing techniques and outlines their simplicity; Section 4 presents the results of our experimental study; Section 5 concludes the paper.

2 Rationale for a New Scheme of Software Rejuvenation

Here, we explain the guidelines of our project, where we developed a software rejuvenation technique.

1. Our rejuvenation mechanism should be easy to apply in off-the-shelf Application Servers without reengineering the applications or the middleware.
2. The mechanism should provide a very fast recovery to reduce the MTTR to the minimum: If possible, we should achieve a zero downtime even in case of restart.
3. The mechanism should not lose any in-flight request or session data at the time of a restart or rejuvenation; the end user should see no impact when there is some restart in the server.
4. The software infrastructure should automate the rejuvenation scheme to improve the self-healing abilities of the server.
5. The rejuvenation mechanism should not introduce a visible overhead during runtime.
6. The scheme should not require any additional hardware: it should work well in single server and in cluster configurations.
7. The mechanism should be easy to deploy and maintain in complex IT systems.

We are mainly focused on the healing of software aging and transient failures. Permanent software bugs, operator mistakes, and hardware failures are out-of-scope of our mechanism. Several studies have reported the large percentage of transient software failures and the importance of software aging in 24×7 applications [8] and this is our target failure model.

The concept of microrebooting, from the ROC Project [25], was likely the most advanced contribution in the area. However, that scheme does not fulfill some of our guidelines: points 1, 3, and 7. First, the microrebooting technique requires the re-engineering of the middleware and the restructuring of the applications to make them "crash-only." The microrebooting concept was experimented in an instrumented version of JBoss, and it cannot be easily applied to other application servers. Second, in their scheme some of the work-in-progress can be lost during a restart operation. For instance, in [29], the authors presented an interesting result: when they applied a restart they lost 3,900 requests, but when they used the microreboot of a component they lost only 78 ongoing requests. Even so, some requests were still lost, which means that some of the end users will see the impact of that microrestart. This may undermine their confidence level in the Website, or may cause some inconsistency in enterprise applications.

We feel that when applying a rejuvenation action to avoid software aging, it is mandatory that we do not lose any request at all: A restart should be fully transparent to all the end users. And it is important that the MTTR for a protective restart should be zero, if possible. A clear recipe to avoid downtime when there is a server restart is to use a cluster configuration: When one server is restarted, there is always a backup server that assures the service. However, even in these cluster configurations, the server restart has to be carefully done to avoid losing the work-in-progress in that particular server. Using a cluster for all the application servers in an enterprise represents a huge increase in terms of budget: in [2], it is argued that the adoption of a cluster may represent an increase of 4-10 times more in the budget, when compared with a single-server configuration. If we increase the number of server machines and load balancers, we are also increasing the cost of management and Total-Cost-of-Ownership (TCO).

In order to achieve a low MTTR for planned restarts even in a single-server configuration, we decided to exploit the use of virtualization. With this technique, we are able to optimize the rejuvenation process without requiring any additional hardware. Our rejuvenation mechanism is totally supported by software, and can be easily deployed in existing IT infrastructures. It can be applied on cluster configurations or single-server applications.

3 VM-Rejuv: The Framework

Our rejuvenation mechanism can be applied in any Application Server: it can be Websphere, WebLogic, JBoss, Tomcat, Microsoft.Net, or others. All the persistent state of the application is maintained in a database that should be made reliable by some RAID topology. An exception is made to the state of the application that is maintained in session objects. This session data cannot be lost in a restart operation. We do not require any restructuring of the applications or the middleware container and the scheme should work

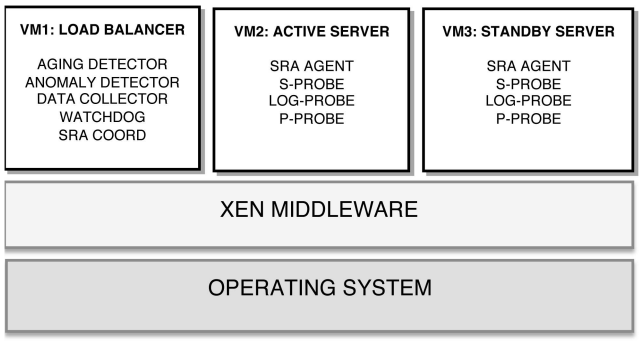

Fig. 1. VM-Rejuv framework.

seamlessly with any server in the market. The main requirement is the use of a virtualization middleware like VMWare [30], XEN [31] or Virtuoso [32].

We have adopted XEN in our experiments, but all the software infrastructure that was developed could be used in any other virtualization middleware. On top of our virtualization layer, we create three virtual machines per application server: one VM to run a software load balancer (VM-LB), one VM where we run the main application server, and a third VM where we execute a replica of the application server that works as a hot standby.

VM-LB will be responsible for the management of fail-over and rejuvenation of the two virtual servers. It will also run some software modules for the detection of aging and transient failures. When something is detected, it will trigger a rejuvenation action in the primary server. In this action, we do not restart the main server right away: first, we start the standby server, the session state is migrated to that server and all the new requests and sessions will be diverted to the standby server. This is important to avoid losing on-going requests. Fig. 1 presents the main modules of our rejuvenation framework. VM1 runs the following software modules:

- a Load Balancer (LB).
- A module that collects system data from the application servers.
- a module that applies some techniques for aging forecast.
- another module that detects transient failures in the target servers.
- a software watchdog.
- the coordinator of the rejuvenation procedure.

In our experiments, the LB module was implemented by using the Linux Virtual Server (LVS [33]). LVS is a 4-layer load balancer that also provides IP fail-over capabilities and a myriad of balancing policies. All the client requests are sent to the LVS that forwards the request to the active application server. For the `Watchdog` functionality, we used the `ldirectord` tool [34]. This tool is bundled together with LVS and has direct access to the LVS server table. It executes HTTP probing to some static HTML object and is used to detect server outages. We have enhanced this tool to avoid the occurrence of false alarms. For the system monitoring, we used Ganglia [35]. It requires the installation of a system probe in every target server (`S-Probe`). The Ganglia probe does the monitoring of several system parameters like CPU, memory usage, swap space, disk usage, number of threads, I/O traffic, connection to the database, among other system parameters. All the collected data is assembled and sent to a main daemon running in VM1 (`Data Collector`). This module applies some detection rules and provides the data feed for two of the other modules: the `Aging Detector` and the `Anomaly Detector`. These two modules use some detection techniques based on the data collected by different sensors:

1. system-level parameters, collected by Ganglia.
2. communication protocol errors (HTTP errors, TCP-IP errors, communication time-outs) that are collected by the watchdog and the P-Probes.
3. error codes and anomaly detection by runtime searching in the log files.
4. application-specific sensors (HTML errors in message responses) that are collected by the P-Probes.
5. performability metrics like the continuous surveillance of the overall throughput and latency as well as the fine-grain latency per service component, collected by the P-Probes.

The `Aging Detector` is a core module in our system: in the first version of our project this module detects potential anomalies and evidences of software aging by applying some reactive rules to the monitoring data. The surveillance of external QoS metrics has been proved relevant in the analysis of Web server and services [37] and it has been highly effective in the detection of aging and fail-stutter failures [38] in a previous study that was presented in [21]. We have used similar techniques in this paper. In the future, we plan to enhance this module with statistical learning techniques [23] and time-series analysis [39] to provide some more accurate forecast of the occurrence of software aging.

The `Anomaly Detector` is a complement of the above module: It detects application-level anomalies, protocol errors, log errors, and threshold violations in system parameters. System data is collected in runtime and the patterns for default and acceptable behavior continuously monitored. If some of the system parameters exceed (above or below) the expected values, some triggers will be launched by this module. The module also includes direct detection of errors in log files and communication messages.

In the other virtual machines (VM2 and VM3), we execute two replicas of the Application Server; both servers should have access to a database in the back-end. In each of these VMs (VM2 and VM3), we install a Software Rejuvenation Agent (`SRA-Agent`) that is responsible for the rejuvenation operation. This module is directly coordinated by the `SRA-Coord`. There are three other probes that should be installed in both VMs:

1. `S-Probe` corresponds to the Ganglia probe;
2. `Log Probe` is a simple software module that performs some anomaly analysis at the logs of the container;

3. `P-Probe`, that is a small proxy installed in front of the application server: It filters some error conditions and collects some fine-grain performance metrics, like throughput and latency. This `P-Probe` is able to distinguish latency variations per service component that is externally accessible by the end users. In the case of a Java-based container like Tomcat, we used the *servlet filter* capability [36] to implement this functionality.

The `SRA-Coord` module works in coordination with the two `SRA-Agents` to achieve a clean restart procedure. During the server migration process, the main concern is to avoid losing any ongoing request that is being executed in the primary server. There is a window of execution where we have both servers running in active mode and we need to assure a clean restart. The primary server is able to be restarted only when all its session data has been migrated to the secondary server, all the in-memory caches should be flushed to the database and the server should have no more ongoing requests. When all these conditions are executed, the local SRA conducts a rejuvenation of the server. By saving all the session state to the standby server, we might be able to achieve one of the fundamental features of crash-only software [40]. In a nutshell, crash-only software is structured in such a way that it is able to handle failures just by restarting without requiring any complex recovery procedure.

All these software modules were implemented by using open-source tools, like LVS, `ldirectord`, and Ganglia. The deployment of this framework is straightforward and does not require any change to the applications or the middleware containers.

Although we have explained the main modules of our framework, the reader should take into account that the main focus of this paper is to present an experimental study of our rejuvenation mechanism.

4 Experimental Study

To study the effectiveness of our rejuvenation scheme, we used two client/server application benchmarks and conducted an experimental study in a dedicated cluster or machines.

4.1 Experimental Environment

4.1.1 Application Benchmarks

We selected two application benchmarks that represent typical Web-based applications: one of them used SOAP as the communication protocol, while the other is based on HTTP protocol.

Tomcat/Axis. This refers to a synthetic Web service that implements a simple shopping store with a database back-end (MySQL). The client application may search for products, add them to a shopping cart and run for checkout. This application used SOAP as the communication protocol. We chose Tomcat/Axis [41] as a SOAP Router since we already knew from a previous study [21] that Axis 1.1.3 was suffering from memory leaks.

TABLE 1
Configuration Parameters of the Machines

	SERVERS: KATRINA & WILMA	SERVERS: TANIA & NELMA	CLIENT MACHINES
CPU	Dual AMD64 Opteron (2000MHz)	Dual Core AMD64 Opteron 165 (1800MHz)	Intel Celeron (1000MHz)
Memory	4GB	2GB	512MB
Hard disk	160GB(SATA2)	160GB(SATA2)	
Swap Space	8GB	4GB	1024MB
Operating System	Linux 2.6.16.21-0.25-smp	Linux 2.6.16.21-0.25-smp	Linux 2.6.15-p3-Netboot
Java JDK	1.5.0_06, 64-bit Server VM	1.5.0_06, 64-bit Server VM	1.5.0_06-b05 Standard Edit.
Tomcat JVM Heap Size	512MB	512MB	
Other Software	Apache Tomcat 5.5.20, Axis 1.3, MySQL 5.0.18	Apache Tomcat 5.5.20, Axis 1.3	

TPC-W. TPC-W is a benchmark that simulates a transaction-oriented Web application and has been used for performance benchmarking of web servers [42]. It simulates three basic profiles for browsing, shopping, and ordering. In this particular case, the benchmark was implemented in Java by a team of CMU [43] and made use of Tomcat and MySQL. Most of the load of this application is kept in the database layer rather than in the Web front-end.

From the initial experiments with TPC-W, we did not detect any visible problem of software aging. To measure the effectiveness of our detection mechanisms, we had to inject synthetic aging in the application. For this purpose, we implemented a small fault injector that works as a resource "parasite": It consumes system resources in competition with the application. This fault injector has support for several resources: CPU, memory, disk, threads, database connections, and IO traffic. We have only used the memory consumption option with an aggressive configuration to speed up the effects of (synthetic) aging in our experiments. This technique to inject memory leaks is similar to the one described in [18].

4.1.2 The Workload Test Tool

To collect some performance metrics, we used a multiclient tool called QUAKE [44] that was implemented in our Laboratory. This tool permits the launching of simultaneous multiple clients that execute requests in a server under test. The tool allows several communication protocols, like TPC-IP, HTTP, RMI, and SOAP. It conducts automated test runs according to some predefined parameters and workloads. The workloads can vary among Poisson, Normal, Burst, and Spike. In this study, we just used the burst distribution to speed up the occurrence of software aging. It was extremely useful to study the behavior of our rejuvenation mechanism with automated test runs and automated result collection.

4.1.3 The Experimental Setup

In our experiments, we used a cluster of 12 machines: 10 machines running the client benchmark application, one Database Server (Katrina or Wilma), and our main server (Tania or Nelma) running XEN and the three virtual machines. All the machines are interconnected with a 100 Mbps Ethernet switch. The detailed description of the machines is presented in Table 1.

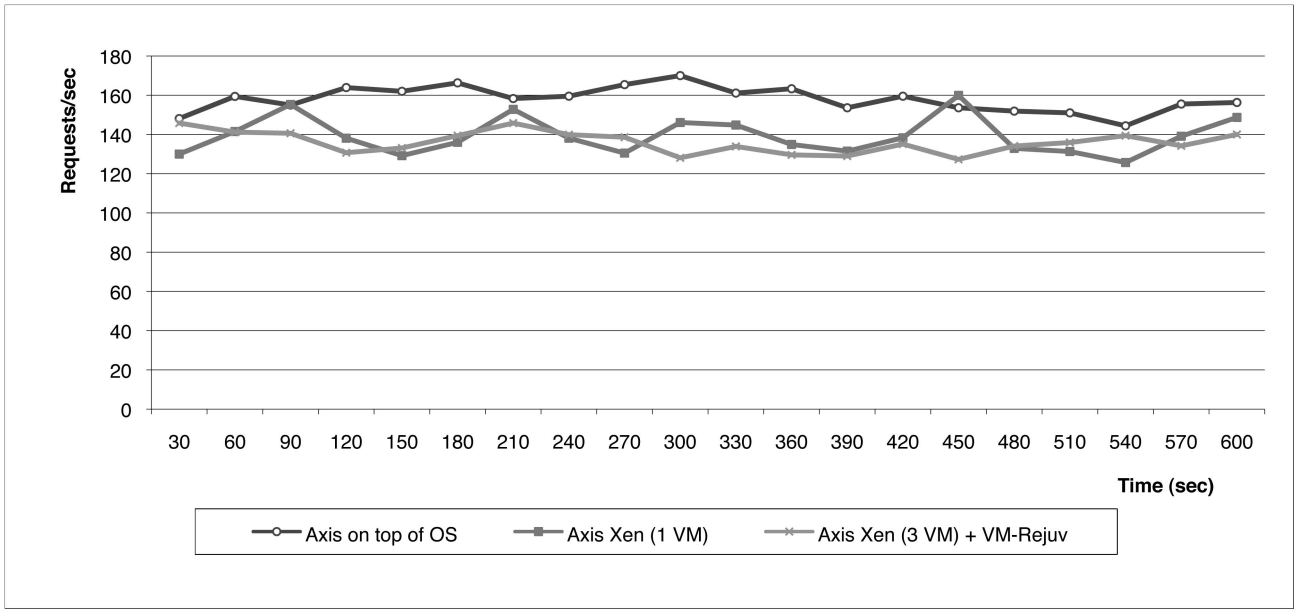

Fig. 2. Comparing the throughput of Tomcat/Axis on top of OS, on top of XEN, and using `VM-Rejuv`.

We used Xen version 3.0.2 [31] configured with three virtual machines: two virtual machines with 700 MB memory for the application servers and a third VM with 256 MB memory to run LVS, the `ldirectord` watchdog and our software modules. Both virtual machines run Linux 2.6.16.21-0.25-xen with 1,024 MB swap space and one virtual CPU. In some of the experiments, where we needed to test with two active application servers, we used the Katrina server and five virtual machines.

4.2 Experimental Results

4.2.1 What Is the Overhead of Using XEN and VM-Rejuv?

In the first experiment, we decided to evaluate the performance penalty for using a virtualization middleware (Xen) and our `VM-Rejuv` framework. We started to use the Tomcat/Axis application benchmark. We executed several short-time runs (15 minutes) in burst mode. From these runs we removed the initial 5 minutes, considering as the warm-up period. So, the total run length was 10 minutes. The comparison of throughput is presented in Fig. 2. As can be seen there is some overhead for using XEN and our `VM-Rejuv` framework, when compared with a simple run on top of the operating system.

Table 2 gives a more precise comparison in terms of average throughput and total number of requests. From that data, we can measure an overhead in terms of total number of requests served in that time interval. In the case of the Tomcat/Axis benchmark, we can see that XEN introduced an overhead of 12 percent, while our mechanism added an additional overhead of 2 percent.

The results for TPC-W were a bit different. In the specification of TPC-W [42], the client code includes a "think time" between requests of the order of 7-70 seconds. This benchmark tried to simulate the access pattern of a Web user to an e-commerce site and this "think time" goes along with that pattern. In this run, we set the "think time" to a fixed value, corresponding to the minimum of 7 seconds. Results are presented in Fig. 3 and the summary in Table 3.

As can be seen, the overhead of using XEN is 1.1 percent and the overhead of `VM-Rejuv` is almost negligible (0.2 percent). This difference in the overhead comes from the fact that this benchmark is not using a continuous burst distribution, as in the previous case of Tomcat/Axis. But it is important to note that the "think time" between requests that is used in TPC-W represents a more likely workload of Websites.

TABLE 2
Comparing the Overhead of Virtualization and VM_Rejuv (Tomcat/Axis)

	AVERAGE THROUGHPUT (REQUESTS/SEC)	TOTAL NUMBER REQUESTS	OVERHEAD
On top of OS	157.9	94,743	---
On top of Xen	139.2	83,541	12%
VM-Rejuv	136.1	81,651	14%

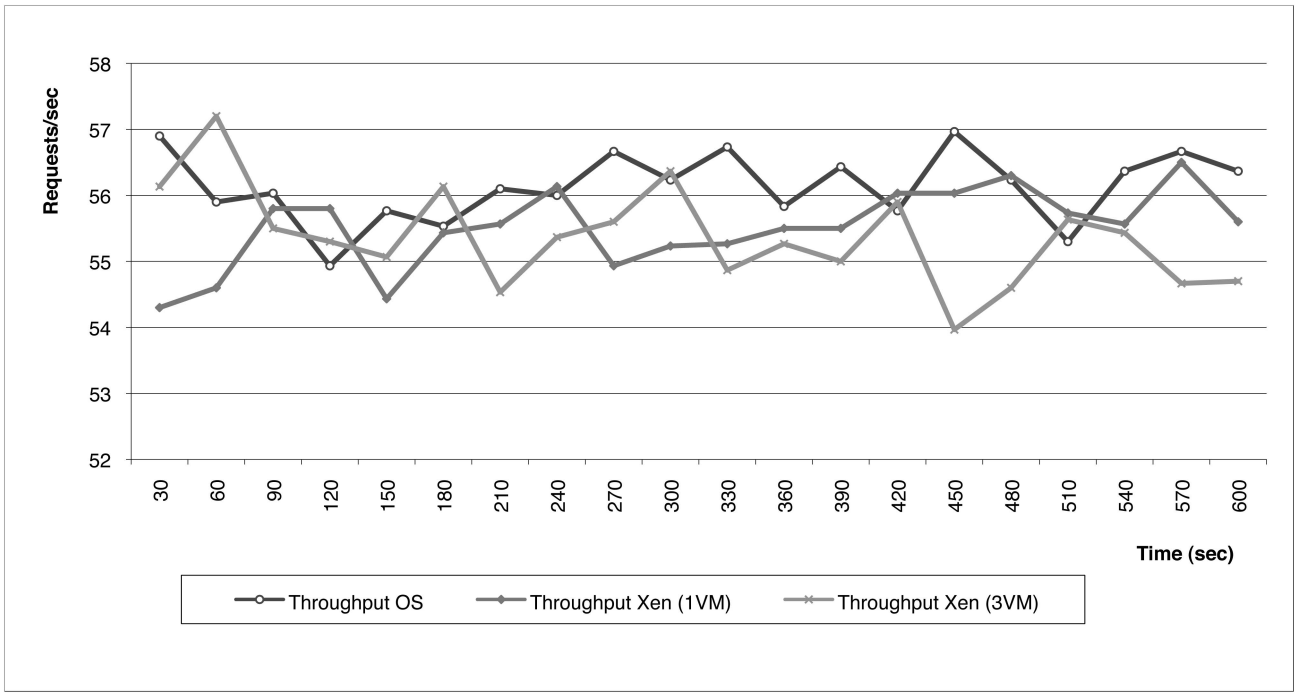

Fig. 3. Comparing the throughput of TPC-W on top of OS, on top of XEN, and using `VM-Rejuv`.

4.2.2 Is VM-Rejuv an Effective Technique?

The next step was to evaluate the effectiveness of our automated rejuvenation mechanism. We made use of the application benchmarks and configured the rejuvenation mechanism to be triggered when there is some threshold violation in one of the external QoS metrics. In this case, we just observed the throughput of the application: Basically, if the application starts to get slower over time, there is a high probability we are facing a fail-stutter behavior [38].

In the case of Axis, we already knew that version 1.3 is suffering from severe memory leaks [21]. When doing some experiments with a burst workload and 10 simultaneous clients, we observed a crash in the application server in less than 5 hours. So, we measured the throughput in time runs of 4 hours when using the application and no rejuvenation. After that, we compared with two scenarios where we applied our rejuvenation scheme when there was a violation in the throughput SLA (Service-Level Agreement). In these experiments, we set up two values for the SLA: 50 percent and 75 percent. When the external throughput decreases to lower than 50 percent or 75 percent of the maximum value (measured when the system was running at the beginning), the `VM-Rejuv` applies a rejuvenation action. Results are presented in Fig. 4.

This figure shows the effectiveness of our automated rejuvenation. If the system starts to get slower a rejuvenation action is applied and for some time the maximum performance figures are restored. We do avoid a crash of the application due to the internal memory leaks of Axis 1.1.3. Without the usage of our scheme the application would crash approximately after 4.5 hours of continuous burst execution. Fig. 5 presents related data from the same experiment: this time the figure presents the observed latency (in milliseconds) when applying an action of rejuvenation when there was a violation of 50 percent and 75 percent, compared with the normal case when there is no rejuvenation. We can see that when applying the mechanism of rejuvenation, the latency of requests is kept to an acceptable value. Otherwise, the latency would start growing until the client starts receiving some time-out replies due to the unavailability of the SOAP server application. This data can be represented in this format due to the burst nature of the associated workload.

TABLE 3
Comparing the Overhead of Virtualization and VM_Rejuv (TPC-W)

	AVERAGE THROUGHPUT (REQUESTS/SEC)	TOTAL NUMBER REQUESTS	OVERHEAD
On top of OS	56.13	33,682	---
On top of Xen	55.51	33,308	1.1%
VM-Rejuv	55.36	33,217	1.3%

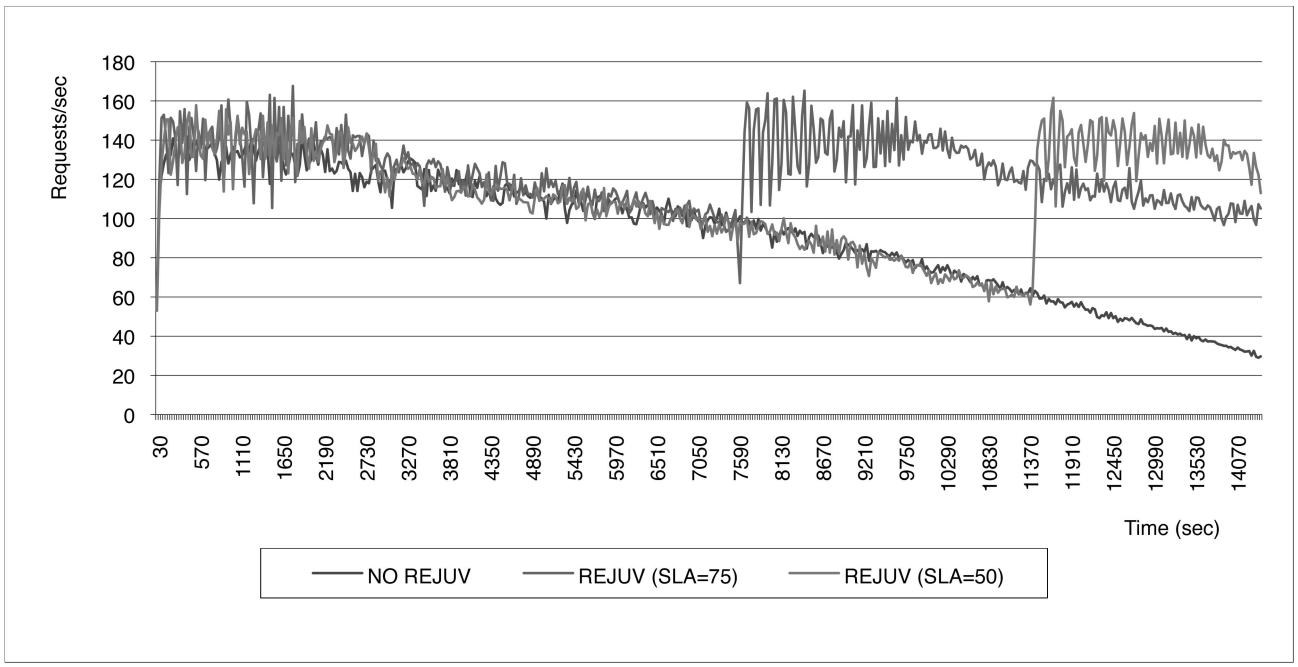

Fig. 4. Throughput of requests when applying rejuvenation to Tomcat/Axis.

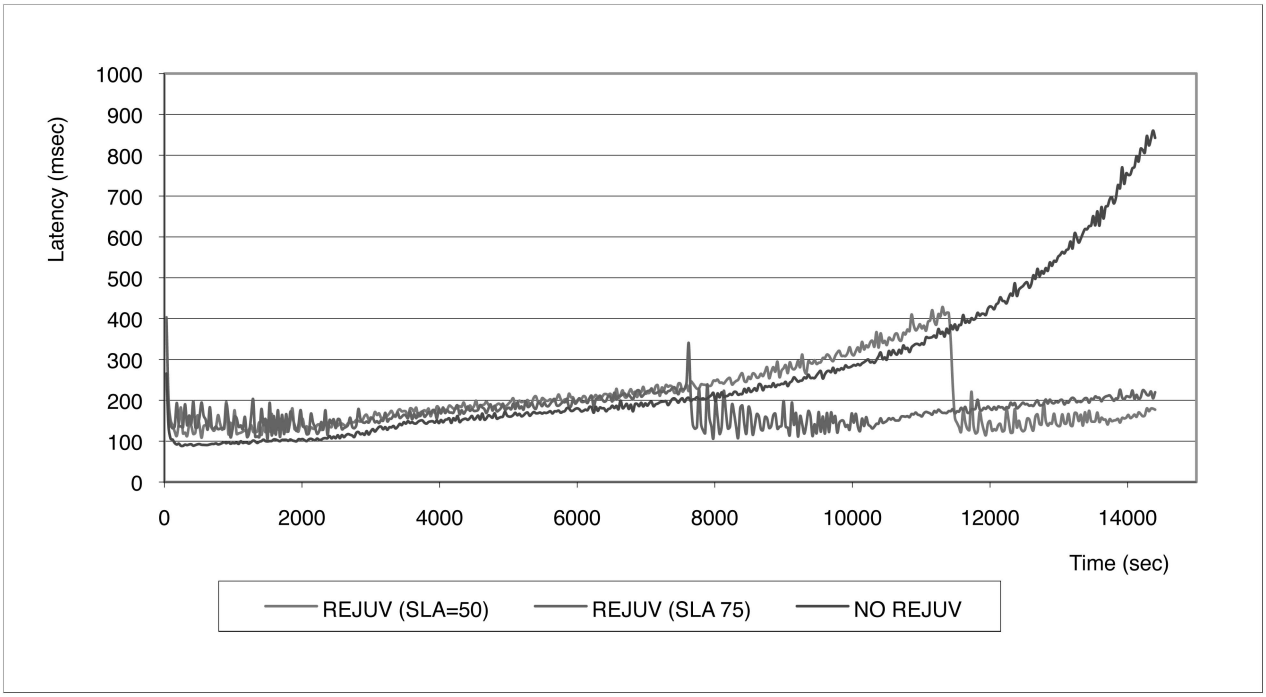

Fig. 5. Latency of requests when applying rejuvenation to Tomcat/Axis.

Fig. 6 presents the results for a similar experiment with TPC-W. In this case, we used a memory fault loader with parasitic behavior similar to the one presented in [18], to resemble the occurrence of similar scenario of software aging due to memory leaks. In these experiments, we injected a memory leak of 1 Kb per request, which is an aggressive leak. The idea was to observe very quickly the crash of the application to compare the effects of our rejuvenation mechanism. In Fig. 6, we can observe that TPC-W application with that synthetic memory leak died after 1.475 hours of execution.

In Fig. 7, we present the latency of requests, measured in milliseconds, when applying the rejuvenation mechanism compared with default case. Without the rejuvenation scheme, the latency of requests grows considerably and the client starts receiving time-outs before the crash of the server.

With the automated rejuvenation scheme, we were able to avoid a crash and kept a sustained level of performance, within the limits defined for the Throughput SLA.

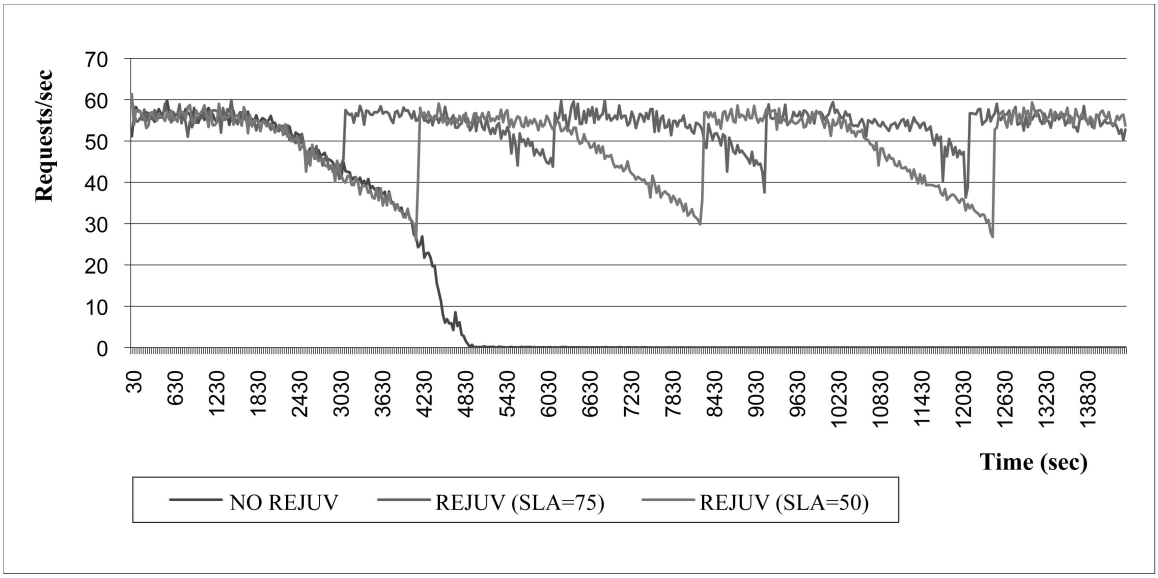

Fig. 6. Throughput of requests when applying rejuvenation to TPC-W.

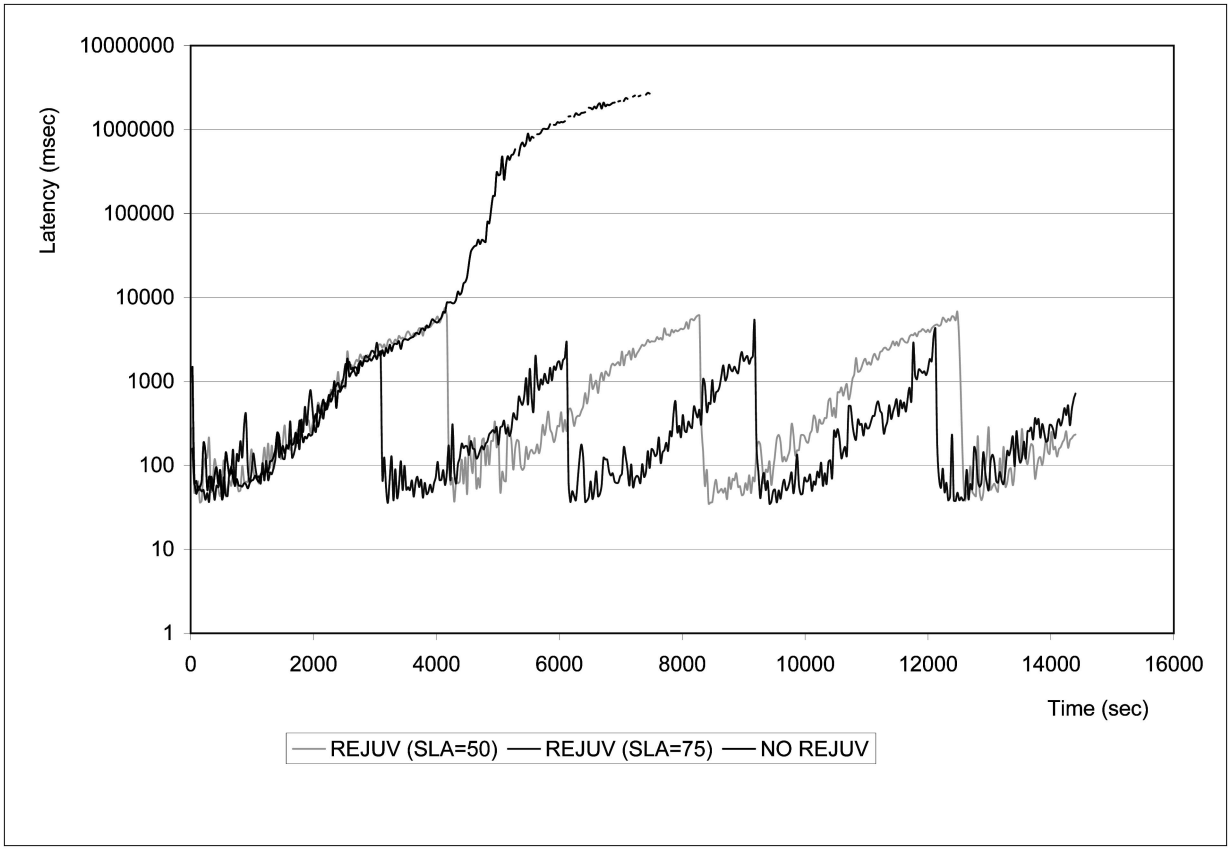

Fig. 7. Latency of requests when applying rejuvenation to TPC-W.

With these results, we start considering this rejuvenation mechanism as a crucial module of our software framework that aims to provide self-healing abilities for the application servers.

4.2.3 What Is the Downtime of Our Rejuvenation Scheme?

In the next step, we wanted to evaluate the potential downtime for the application when there is a restart, as well as the number of failed requests and the potential loss of session data during the rejuvenation process. These results are of extreme importance, according to guidelines 2 and 3 described in Section 2. We conducted four experiments in the same target machine using the Tomcat/Axis application:

1. one run, where we applied a rejuvenation action at the time of 300 seconds,

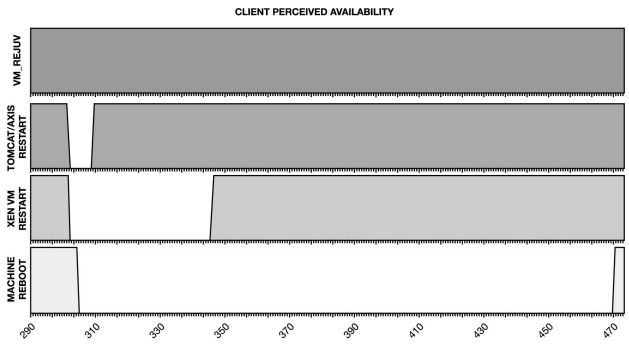

Fig. 8. Client perceived availability for different restart mechanisms (Tomcat/Axis).

2. another run, where we triggered a Tomcat restart at exactly that time,
3. a third run, where we applied a restart in the XEN Virtual Machine, where the main server was running, at the exact same time (300 seconds), and finally,
4. a last one, where we executed a full machine reboot at that execution time.

The results are presented in Fig. 8. That figure presents an execution window of 600 seconds. We adopted that format for an easy comparison with a similar result achieved by George Candea in the work of microrebooting, published in [29]. As can be seen, our rejuvenation scheme achieves a zero downtime. The clients do not even notice there was a rejuvenation action in the active server. In the other cases, there was some visible downtime for the end users: a Tomcat/Axis restart produced a visible downtime of 12 seconds; the restart of a XEN VM resulted in a visible downtime of 56 seconds; and finally a machine reboot represented a downtime of about 200 seconds.

The next step was to analyze the number of failed requests caused by one restart operation. Results are presented in Table 4. That table presents the average throughput during a test run of 10 minutes, considering there was one restart operation. We present the total number of requests executed during that period, the number of failed requests, the number of slow requests, and the perceived downtime, as measured from the client's side.

As can be seen, a server restart using our rejuvenation scheme resulted in zero failed requests: No work-in-progress was lost due to that restart. This was important to fulfill our guideline 3. The implementation of microrebooting produced a very low MTTR (even so, higher than zero) but it allowed the occurrence of some failed requests: as presented in a particular experiment in [29], a microreboot of an EJB component would still cause 78 failed requests. This means a microreboot was not transparent for the end users and may allow a few visible failures. In our case, we do provide a clean restart of the server with no perception for the end user and no impact in the application consistency.

The definition of "slow requests" corresponds to those requests that have a response time higher than 8 seconds. Here, we adopt the same threshold as the one proposed in [29]: It corresponds to the SLA usually used by production Websites as the maximum acceptable response time.

There were no slow requests when we applied a VM-restart or a node reboot: those operations caused refused connections that were counted as failed requests but none of them was classified as a slow request. The time-out mechanism in these two cases is completely different from the case when the machine is available but the application server (Tomcat) is restarting. This is related to the TCP-IP mechanism for time-outs [45], and explains why the number of failed requests when there is a machine reboot was not proportional to the downtime.

The major achievement of our rejuvenation mechanism was the zero downtime. This result has a strong impact if we want to do some simple calculations on the resulting availability when we apply prophylactic restarts (rejuvenation actions). Without considering unplanned failures, if we want to achieve a 99.999 percent availability in a whole year in a single-server machine, then we have the following limits for the number of restarts: we can only do one node reboot in that whole year; or, five restarts in the XEN virtual machine where the server is running; or, alternatively, if it works out, 25 restarts in the Tomcat/Axis application server. If the target application server is installed with our `VM-Rejuv` framework then we can apply a huge number of planned restarts per year within the five nine figure: the only constraint is the minimum acceptable interval between restarts.

4.2.4 Does Our Rejuvenation Scheme Maintain All the Session Data? What Is the Overhead for That?

In this section, we do present some results about the session replication scheme of Tomcat. To avoid losing session data, we need to replicate the session objects from

TABLE 4
Comparing Downtime and Failed Requests with Different Restart Mechanisms

	VM-REJUV	TOMCAT RESTART	XEN-VM RESTART	MACHINE REBOOT
Average Throughput (req/sec)	143.8	134.7	128.8	97.3
Total Requests	86313	80847	77292	58401
Failed Requests	0	476	536	1902
Slow Requests	0	10	0	0
Downtime (msec)	0	12490	56143	200722

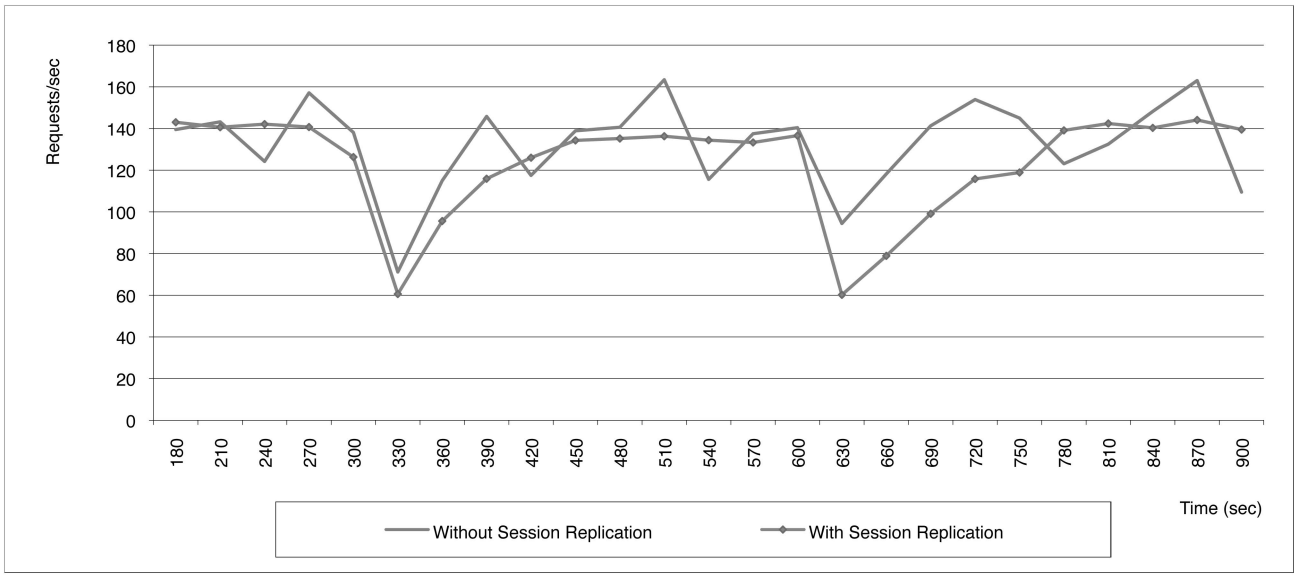

Fig. 9. Comparing the overhead of session replication in Tomcat/Axis (session objects of 8 Kb).

the primary to the secondary server. This replication scheme may introduce some visible overhead, but it is of great importance if we do not want to lose any session data when there is a restart operation.

We have conducted an experiment of 15 minutes with Tomcat/Axis configured with the session objects of 8 Kb (typical session object size is less than this value). The results are presented in Fig. 9. The figure presents two scenarios: 1) one with session replication and 2) another without it. We just presented the last 750 seconds of the experiment, since we removed the warm-up time.

When using session replication, we did not lose any session data and the application served a total of 107,301 requests during those 15 minutes. Without session replication, we observed 15 session errors and the application was only able to serve 117,750 requests. The difference was around 8.8 percent. With session objects of 1 Kb, we got an overall difference of 4.9 percent in the number of requests in that time period. To summarize: there is some overhead but it is mainly visible when we are executing a continuous burst workload, which was the case of Tomcat/Axis. In the case of TPC-W, the overhead was negligible.

4.2.5 Is Our Rejuvenation Scheme Useful in a Cluster Configuration?

So far, we have been testing our rejuvenation in single servers. One key question still remains:

Is our scheme any useful when we have a cluster configuration?

When we use a cluster, we have a load balancer box to support for IP fail-over in the occurrence of a server crash. Clusters are used to increase performance and to tolerate failures. Our rejuvenation scheme is meant to avoid failures due to software aging. So, it has a high potential of being used in cluster configurations.

Suppose we have an application server that degrades over time due to aging. If we replicate that application through N servers, it will degrade in all the servers, probably with different decay functions, but it will decay in the overall. If we use our rejuvenation scheme, we can mitigate the impact of these "fail-stutter" failures. We also have a way to restart a server without losing any work-in-progress, something that sometimes is not achieved when IT managers apply planned restarts in servers belonging to a cluster.

To demonstrate the potential of our rejuvenation scheme in clusters, we have set up the following experiment: we executed the Tomcat/Axis application in a cluster configuration (using three virtual machines in XEN: one LB and two active servers). In order to speed up the visualization of the aging, we configured the JVM to 64 Mb. The throughput curves of both servers are presented in Fig. 10. Since both servers run the same application, they will suffer from aging at a similar pace, if the load balancer is using a round-robin strategy.

During this experiment, Server S1 had a crash after 1 hour of execution. LVS does the migration of all the requests to S2. We decided not to restart S1 automatically to observe the behavior of S2 without interference. Server S2 had a crash after 1.29 hours.

When S1 had a crash, we can see in the figure that the throughput of S2 went temporarily down to almost zero. This was caused by the sudden migration of all the new requests to S2, supported by the fail-over mechanism of the load balancer. But after that new peak, the performance degrades until server S2 also crashes and the cluster remains totally unavailable.

In Fig. 11, we can see the performance decay in the overall throughput of the cluster, down to zero, assuming there is no automatic restart in any crashed machine of the cluster. We can also see the overall throughput when using our rejuvenation scheme (with a trigger at 75 percent of the throughput SLA).

Our rejuvenation mechanism was able to maintain a sustained level of performance: the cluster did not face any

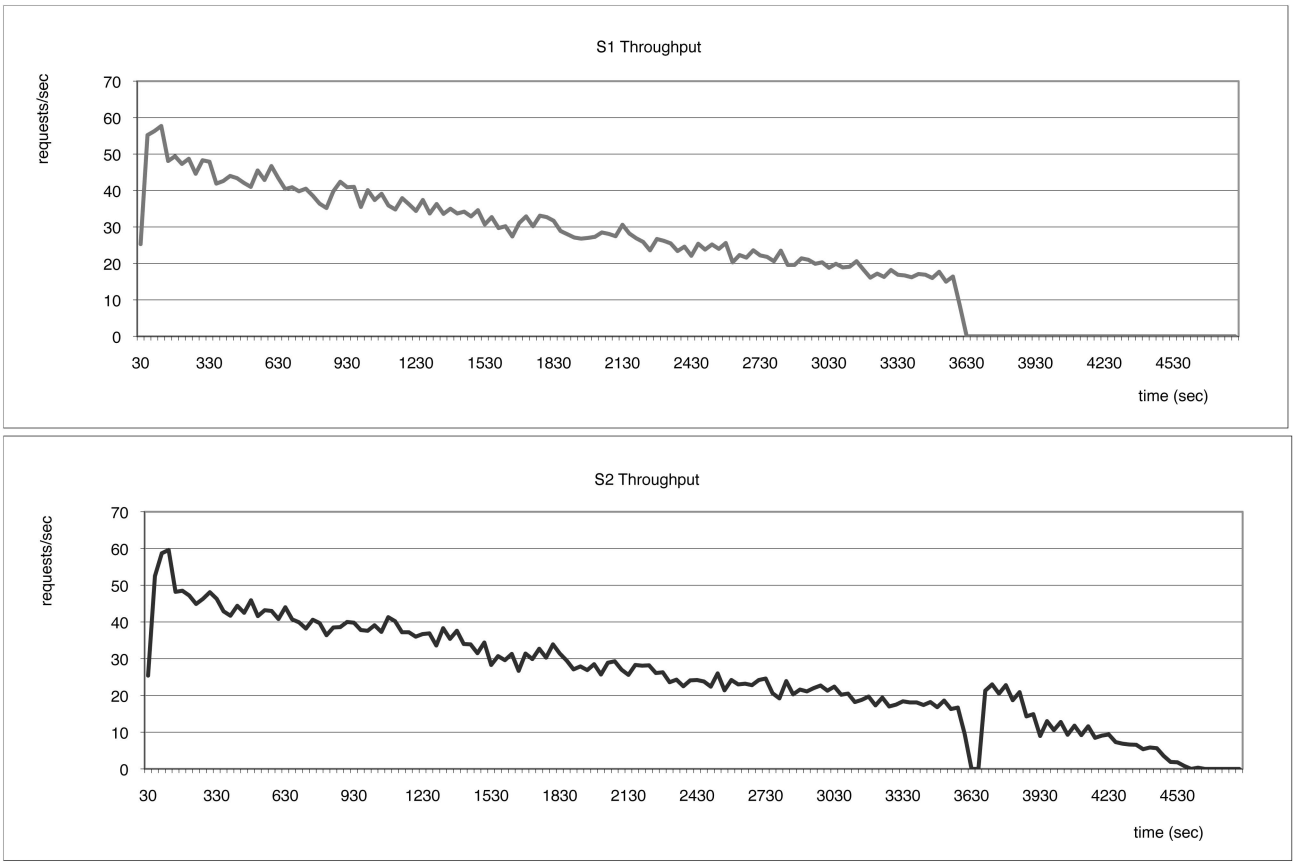

Fig. 10. Throughput of two servers in a cluster, suffering from severe aging (Tomcat/Axis).

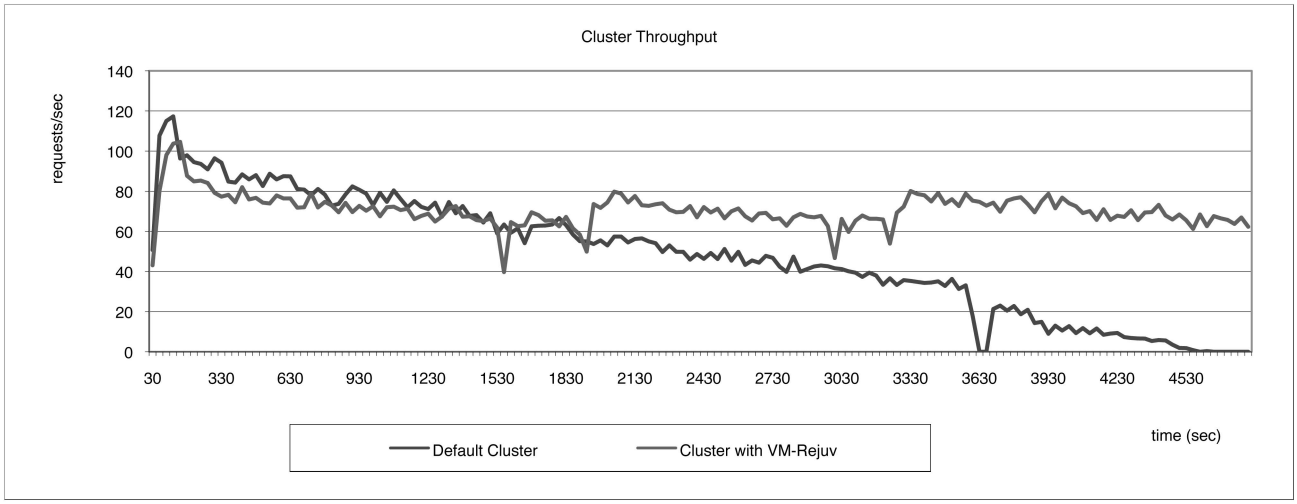

Fig. 11. Comparing the cluster throughput, with and without rejuvenation (Tomcat/Axis).

performance failure or a complete crash (as in the default case). There was no failed request when using rejuvenation, while in the fail-over mode of the default cluster we noticed 2,223 failed requests. During the 4,800 seconds of the experiment, the default configuration of the cluster was able to serve a total of 231,044 requests from the clients. When using our rejuvenation technique, the cluster managed to serve 339,186 requests. This means that our mechanism promoted an increase of about 46 percent in the performance of the application.

We have done a second experiment where we included an automatic restart in the default cluster configuration: when the `ldirectord` detects a crash, it restarts the failed server automatically. Fig. 12 presents a time window of 5 hours of execution.

The figure presents the cluster throughput when we use our rejuvenation scheme compared to the default configuration with automatic restart. Without our rejuvenation, there were several crashes in the cluster and we lost 2,640 requests. With our rejuvenation scheme, we had no failed request. Comparing the total number of requests served, we have

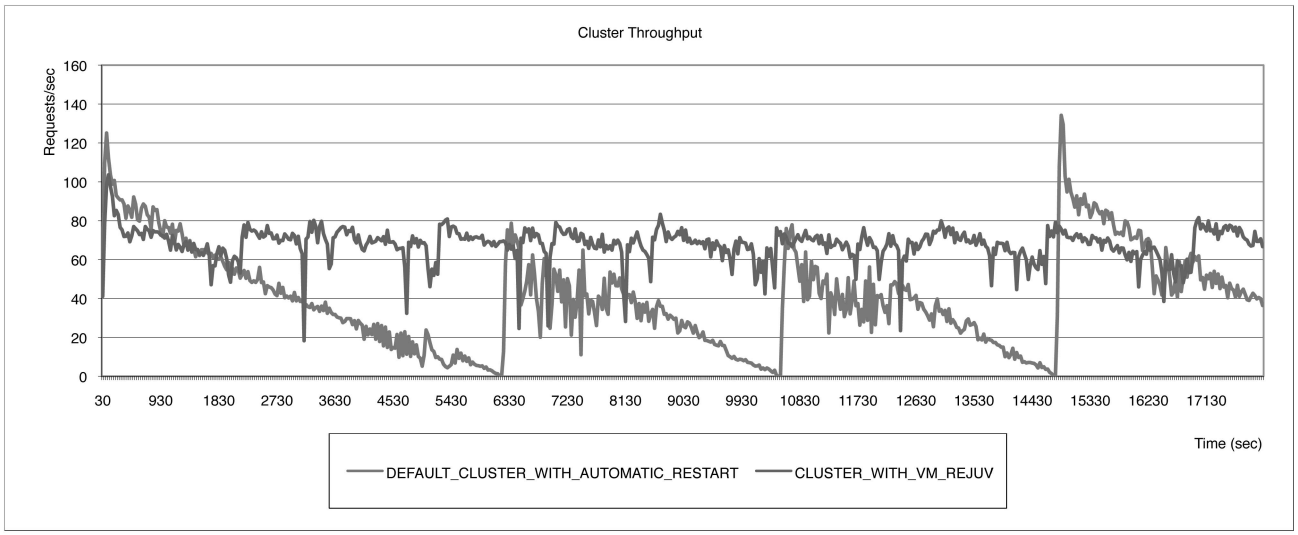

Fig. 12. Comparing the cluster throughput, with and without rejuvenation, but automatic restart (Tomcat/Axis).

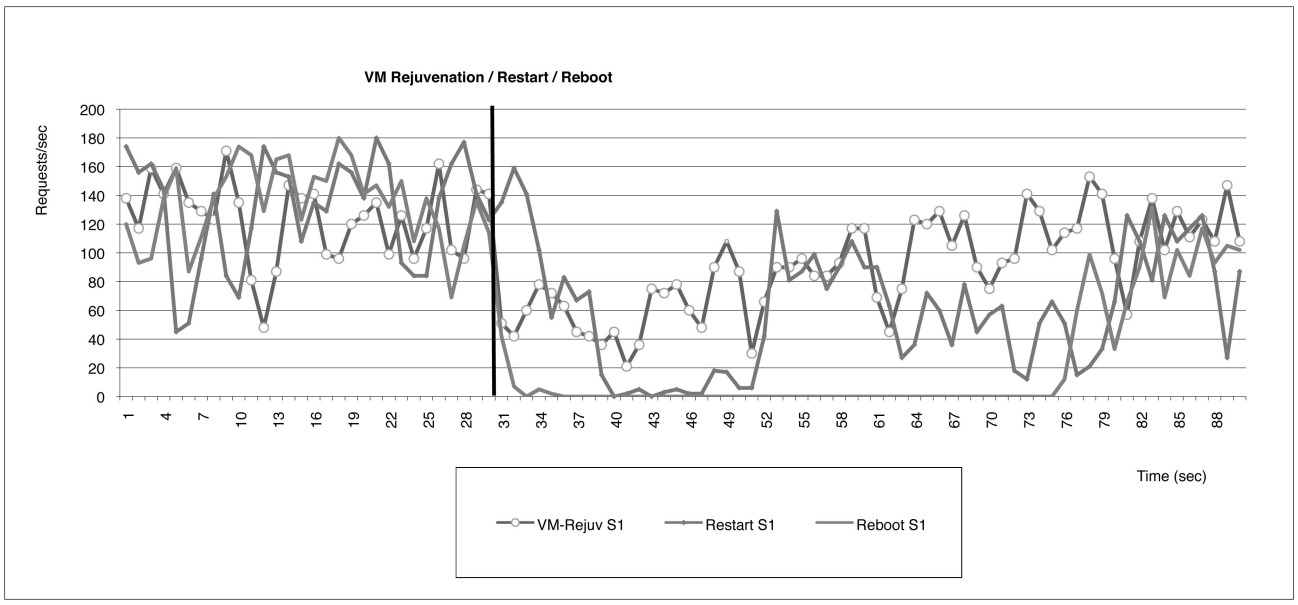

Fig. 13. The impact of a "clean" restart when compared with a "blind" restart and a "blind" reboot (Tomcat/Axis).

observed an improvement of 66 percent in the overall performance.

This result proves the high potential of using our rejuvenation scheme in cluster configurations: it does not only avoid crashes due to aging, but also increases the performance if the application is suffering from some "fail-stutter" effect.

4.2.6 How Important Is the Use of a "Clean" Restart Instead of Applying a "Blind" Restart?

In this experiment, we decided to compare the importance of applying our "clean" restart mechanism when compared with typical procedures: When there is a decision to trigger a rejuvenation action, the server is restarted without any extraordinary concern. We call this procedure a "blind" restart.

In this experiment, we used a cluster with two active servers. Our goal was to compare the effect of a "clean" restart from our `VM-Rejuv` scheme with a "blind" restart of Tomcat and a "blind" reboot of one active server. In this configuration, our rejuvenation mechanism was also used in the cluster configuration with two active servers. Results are presented in Fig. 13.

This figure only presents a "zoom-in" of the results of around 90 seconds. After 30 seconds, we applied a restart action in S1 using our `VM-Rejuv` scheme, a "blind" restart of server S1, and a "blind" reboot of server S1. In our scheme, we had zero failed requests as opposed to the other schemes: The "blind" restart produced 244 failed requests, while the "blind" reboot led to 341 failed requests. These failed requests happened even when we had a cluster configuration.

It can also be seen in the figure that there was a "window" with a very low throughput right after the "blind" restart. It was even more visible after the "blind" reboot. That happens because when server S1 is restarted

there are several requests that get an exception and the LB applies a fail-over to server S2. However, the XEN layer gives more CPU to server S1 during the restart phase. During that phase S2 does not get enough CPU and some of the requests are not executed or executed with extra delay.

5 Conclusions

In this paper, we presented a simple but effective approach for software rejuvenation: It can be applied to off-the-shelf application servers, it achieves a zero downtime without losing any work-in-progress and does not incur in a significant performance overhead. It can be used in a single server or in a cluster configuration without any additional cost. We just require the use of a virtualization layer and the installation of some software modules. The goal of our project is to study the application of the software rejuvenation technique as an important piece of the puzzle that is necessary to provide self-healing abilities for application servers.

Acknowledgments

This research was supported in part by the FP6 Network of Excellence CoreGRID, funded by the European Commission (contract IST-2002-004265) and in part by the Spanish Ministry of Education and Science (projects TIN2007-60625).

References

[1] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach.* Morgan & Kaufmann, 2002.
[2] E. Marcus and H. Stern, *Blueprints for High Availability.* Wiley, 2003.
[3] J. Kephart and D.M. Chess, "The Vision of Autonomic Computing," *Computer,* vol. 36, no. 1, pp. 41-50, Jan. 2003.
[4] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software Rejuvenation: Analysis, Module and Applications," *Proc. 25th Int'l Symp. Fault-Tolerant Computing,* June 1995.
[5] A. Avritzer and E. Weyuker, "Monitoring Smoothly Degrading Systems for Increased Dependability," *Empirical Software Eng. J.,* vol 2, no. 1, pp. 59-77, 1997.
[6] Apache, http://httpd.apache.org/docs/, 2009.
[7] Microsoft IIS, http://www.microsoft.com/, 2009.
[8] V. Castelli, R. Harper, P. Heidelberg, S. Hunter, K. Trivedi, K. Vaidyanathan, and W. Zeggert, "Proactive Management of Software Aging," *IBM J. Research and Development,* vol. 45, no. 2, Mar. 2001.
[9] K. Cassidy, K. Gross, and A. Malekpour, "Advanced Pattern Recognition for Detection of Complex Software Aging Phenomena in Online Transaction Processing Servers," *Proc. 2002 Int'l Conf. Dependable Systems and Networks,* 2002.
[10] A. Tai, S. Chau, L. Alkalaj, and H. Hecht, "On-Board Preventive Maintenance: Analysis of Effectiveness and Optimal Duty Period," *Proc. Third Workshop Object-Oriented Real-Time Dependable Systems,* 1997.
[11] E. Marshall, "Fatal Error: How Patriot Overlooked a Scud," *Science,* vol. 255, pp. 1344-1347, Mar. 1992.
[12] MemProfiler, http://memprofiler.com/, 2009.
[13] Parasoft Insure++, http://www.parasoft.com, 2009.
[14] K. Vaidyanathan and K. Trivedi, "A Comprehensive Model for Software Rejuvenation," *IEEE Trans. Dependable and Secure Computing,* vol. 2, no. 2, pp. 124-137, Apr.-June 2005.
[15] S. Garg, A. van Moorsel, K. Vaidyanathan, and K. Trivedi, "A Methodology for Detection and Estimation of Software Aging," *Proc. Ninth Int'l Symp. Software Reliability Eng.,* pp. 282-292, 1998.
[16] K. Vaidyanathan and K.S. Trivedi, "A Measurement-Based Model for Estimation of Resource Exhaustion in Operational Software Systems," *Proc. 10th IEEE Int'l Symp. Software Reliability Eng.,* pp. 84-93, 1999.
[17] L. Li, K. Vaidyanathan, and K. Trivedi, "An Approach for Estimation of Software Aging in a Web-Server," *Proc. 2002 Int'l Symp. Empirical Software Eng. (ISESE '02),* 2002.
[18] K. Gross, V. Bhardwaj, and R. Bickford, "Proactive Detection of Software Aging Mechanisms in Performance Critical Computers," *Proc. 27th Ann. IEEE/NASA Software Eng. Symp.,* 2002.
[19] K. Kaidyanathan and K. Gross, "Proactive Detection of Software Anomalies through MSET," *Proc. Workshop Predictive Software Models (PSM '04),* Sept. 2004.
[20] K. Gross and W. Lu, "Early Detection of Signal and Process Anomalies in Enterprise Computing Systems," *Proc. 2002 IEEE Int'l Conf. Machine Learning and Applications (ICMLA '02),* June 2002.
[21] L. Silva, H. Madeira, and J.G. Silva, "*Software Aging and Rejuvenation in a SOAP-Based Server*," *Proc. IEEE Int'l Symp. Network Computing and Applications (NCA),* July 2006.
[22] L. Bernstein, Y.D. Yao, and K. Yao, "Software Avoiding Failures Even When There Are Faults," *DoD Software Tech News,* vol. 6, no. 2, pp. 8-11, Oct. 2003.
[23] A. Andrzejak and L.M. Silva, "Deterministic Models of Software Aging and Optimal Rejuvenation Schedules," *Proc. 10th IFIP/IEEE Int'l Symp. Integrated Network Management (IM '07),* May 2007.
[24] G. Candea, A. Brown, A. Fox, and D. Patterson, "Recovery Oriented Computing: Building Multi-Tier Dependability," *Computer,* vol. 37, no. 11, pp. 60-67, Nov. 2004.
[25] G. Candea, E. Kiciman, S. Zhang, and A. Fox, "JAGR: An Autonomous Self-Recovering Application Server," *Proc. Fifth Int'l Workshop Active Middleware Services,* June 2003.
[26] A. Fox and D. Patterson, "When Does Fast Recovery Trump High Reliability?" *Proc. Second Workshop Evaluating and Architecting System Dependability,* 2002.
[27] R. Figueiredo, P. Dinda, and J. Fortes, "Resource Virtualization Renaissance," *Computer,* vol. 38, no. 5, pp. 28-69, May 2005.
[28] M. Rosenblum and T. Garfinkel, "Virtual Machine Monitors: Current Technology and Future Trends," *IEEE Internet Computing,* vol. 38, no. 5, pp. 39-47, May/June 2005.
[29] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot—A Technique for Cheap Recovery," *Proc. Sixth Symp. Operating Systems Design and Implementation (OSDI '04),* Dec. 2004.
[30] VMware, http://www.vmware.com, 2009.
[31] Xen, http://www.xensource.com, 2009.
[32] Virtuoso, http://www.virtuoso.com, 2009.
[33] LVS, http://www.linuxvirtualserver.org/, 2009.
[34] ldirectord, http://www.vergenet.net/linux/ldirectord, 2009.
[35] Ganglia, http://ganglia.sourceforge.net, 2009.
[36] Essential about Java Servlet Filters, http://java.sun.com/products/servlet/Filters.html, 2009.
[37] D. Menascé, "QoS Issues in Web Services," *IEEE Internet Computing,* vol. 6, no. 6, pp. 72-74, Nov./Dec. 2002.
[38] R. Arpaci-Dusseau and A. Arpaci-Dusseau, "Fail-Stutter Fault Tolerance," *Proc. Eighth Workshop Hot Topics in Operating Systems, (HOTOS-VIII),* 2001.
[39] S. Makridakis, S. Wheelwright, and R. Hyndman, *Forecasting: Methods and Applications,* third ed. John Wiley & Sons, 1998.
[40] G. Candea and A. Fox, "Crash-Only Software," *Proc. Ninth Workshop Hot Topics in Operating Systems,* 2001.
[41] Apache Axis, http://ws.apache.org/axis, 2009.
[42] TPC-W Specification, http://www.tpc.org/tpcw/specs.asp, 2009.
[43] TPC-W in Java for Tomcat and MySQL, http://www.cs.cmu/edu/~manjhi/tpcw.html, 2009.
[44] S. Tixeuil, W. Hoarau, and L.M. Silva, "An Overview of Existing Tools for Fault-Injection and Dependability Benchmarking in Grids," Technical Report TR-0041, CoreGRID, http://www.coregrid.net, 2009.
[45] TCP-IP RFC 793, http://www.ietf.org/rfc/rfc793.txt, 2009.

Luis Moura Silva received the degree in computer engineering from the University of Coimbra in 1990, the master's degree in computer science from the University of Lisbon in 1993, and the PhD degree in computer science from the University of Coimbra in 1997. He is an associate professor in the Department of Computer Science at the University of Coimbra, Portugal. He has published more than 110 scientific papers in the literature. His current research interests include dependable computing, autonomic computing, mobile systems, and large-scale computing.

Javier Alonso received the degree in computer science from the Universitat Politècnica de Catalunya (UPC) in 2004. He is currently working toward the PhD degree on the topic of self-healing techniques for Web-based applications. He has a temporary position in the Computer Architecture Department as assistant professor since 2006.

Jordi Torres received the master's degree in computer science in 1988 and the PhD degree in 1993 from the Technical University of Catalonia (Universitat Politècnica de Catalunya, UPC). Currently, he is a full professor in the Computer Architecture Department at UPC and is a manager for the Autonomic Systems and eBusiness Platforms research line at the Barcelona Supercomputing Center. Further, he is actively working to combine the research from different areas such as autonomic computing, parallel and distributed systems, performance modeling, virtualization, and machine learning, among others, to reasonably stem the difficulties in obtaining more sustainable IT. He has more than 90 publications to his credit in the field. He received the Best UPC Computer Science Thesis Award in 1993.