Using Virtualization to Improve Software Rejuvenation

Luis Moura Silva[1], Javier Alonso[2], Paulo Silva[1] , Jordi Torres[2], Artur Andrzejak[3]

(1) CISUC, Univ. of Coimbra, Portugal
Email: luis@dei.uc.pt

(2) BSC-UPC, Barcelona, Spain
Email: alonso@ac.upc.edu
torres@ac.upc.edu

(3) ZIB, Berlin, Germany
Email: andrzejak@zib.de

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 and Internet sites. 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 be the less disruptive as possible for the running service and to get a zero downtime for most of the 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 applications benchmarks (Tomcat/Axis and TPC-W). Our results demonstrate that virtualization can be extremely helpful for software rejuvenation and fail-over in the occurrence of transient application failures and software aging.

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 100K 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. This works quite fine to tolerate application crashes. The latest trend goes towards 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 our 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 phenomena of software aging or caused by transient failures. The term software aging describes the phenomena of progressive degradation of the running software that may lead to system crashes or undesired hang ups [4]. This phenomena is likely to be found in any type of software that is complex enough, 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 widely number of cases the only choice is to apply some sort of software rejuvenation.

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][14-21] and it is widely understood that this technique of rejuvenation 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] suffers from clear memory leaks.

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

Sixth IEEE International Symposium on Network Computing and Applications (NCA 2007)
0-7695-2922-4/07 $25.00

server restart would take 20 seconds of downtime, a micro-reboot of a set of components could take only 600 milliseconds.

In [26], it was well explained that it is easier and definitely more valuable to approach the goal of 100% availability by reducing the MTTR instead of paying efforts to increase the MTBF (Mean-Time-Between-Failures) of systems. 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 in the Internet world.

Driven by the results of the ROC project and by this 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 re-engineering the applications or the middleware. These rejuvenation mechanisms should minimize the MTTR of the applications.

In this context we looked into the current state-of-the-art in terms of virtualization technology [27][28] and we realized that it can be a good recipe to optimize the process software rejuvenation. Encouraged by this concept, we have done a prototype implementation of our VM-based rejuvenation approach followed by an experimental study. The results are promising and 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

To motivate our approach for software rejuvenation we start by explaining the seven guidelines that characterize our software-rejuvenation mechanism:

1- Our rejuvenation mechanism should be easily applied in off-the-shelf Application Servers without re-engineering the applications neither the middleware;
2- The mechanism should provide a very fast recovery to cut down 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 rejuvenation; the end-user or the interacting application should see no impact at all when there is a server restart;
4- The software infrastructure should automate the rejuvenation scheme in order to achieve a self-healing system;
5- The mechanism should not introduce a visible overhead during the run-time phase;
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;

Achieving all these guidelines seems to be very demanding but it is still achievable. If the application of our rejuvenation scheme could be automated then we get some contribution for the development of a self-healing system. This is our ultimate goal.

We are mainly focused in the healing of software aging and transient failures. Permanent software bugs, operator mistakes and hardware failures are out-of-scope of our mechanism. However, several studies have reported the large percentage of transient software failures and the importance of software aging in 24x7 applications [8].

The concept of micro-rebooting, from the ROC Project [25], was probably the most advanced contribution in the area. However, that scheme does not fulfill some of our guidelines: (1), (3) and (7). First, the micro-rebooting technique requires the re-engineering of the middleware container and the re-structuring of the applications to make them crash-only. The authors proved their concept in an instrumented version of JBoss, but it can not be easily applied to other application servers. Secondly, in their scheme some of the work-in progress is still being lost during a restart operation. For instance, in [29] the authors present an interesting result: when they applied a restart they lost 3900 requests, but when they used the micro-reboot of a component they would have lost only 78 on-going requests. Some requests were still being lost, which means that some of the end-users will see the impact of that micro-restart, which will undermine their confidence level in the web-site, or may even cause some data inconsistency in enterprise applications.

We feel that when applying a planned restart 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 secondly, it is important that the MTTR for a protective restart is zero, if possible. These are two important goals we want to achieve.

A clear recipe to avoid downtime when there is a server restart is to use a hardware-supported 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 server. Using a cluster for all the application servers in an enterprise represents a huge increase in terms of budget: in [2] is argued that the adoption of a cluster may represent an increase of 4 to 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 increasing the cost of management and TCO (Total-Cost-of-Ownership).

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. Obviously, it can either work on cluster configurations or single-server applications.

3. `VM-Rejuv`: The Framework

We target our rejuvenation mechanism to any Application Server. It can be Websphere, Bea WebLogic, JBoss, Tomcat, Microsoft .Net, or others. The clients communicate with the server through TCP-IP HTTP or SOAP. All the persistent state of the application is maintained in a database that should be made reliable by some RAID scheme. Exception is made to some important state of the application that is maintained in session-objects. This state is important for the end-users and cannot be lost in a restart operation. We do not require any re-structuring of the applications nor any change to the middleware container. The scheme should work 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 we may have used any virtualization middleware. On top of our virtualization layer we create 3 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 instantiate a replica of the application server that works as a hot-standby.

The VM-LB also runs some other software modules that will be responsible for the detection of software aging or some potential anomaly. When this happens this module will trigger a rejuvenation action. In this action we do not restart the main server right away: first, we start the standby server, all the new requests and sessions are sent by the LB to this second server, the session-state is migrated from the primary to the secondary server and we wait for all the on-going requests to be finished in the primary server. When we are able to do this we can restart the main server without losing any in-flight request or session state. We call this a "clean" restart.

Figure 1 represents our framework (called for the time being `VM-Rejuv`).

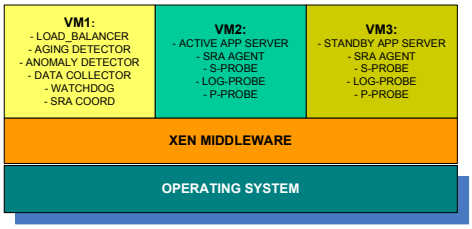

Figure 1: `VM-Rejuv` Framework

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 potential anomalies in the target servers;
- a software watchdog;
- and the coordinator of the rejuvenation procedure.

In our experiments the LB module was implemented by 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 in the time-out policies 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`) that applies some threshold filters. This module 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 data collected by different sensors: (i) System-level parameters, collected by Ganglia; (ii) Communication protocol errors (HTTP errors, TCP-IP errors, communication time-outs) that are collected by the watchdog and the P-Probes; (iii) Error codes and anomaly detection in logs; (iv) Application-specific sensors (HTML errors in message responses) that are collected by the P-Probes; (v) 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: for the time being it just uses some simple threshold techniques based on the conditions observed on the target server. 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: basically, it detects application-level anomalies, protocol errors, log errors and threshold violations in system parameters. This module is still an on-going development stage.

In the other virtual-machines (VM2 and VM3) we instantiate the Application Server, that may have access to a database in the back-end. Together with the server we install a `SRA-Agent` (Software Rejuvenation Agent) that is responsible for the rejuvenation operation. This module is directly coordinated by the `SRA-Coord`. Three other probes should also be installed in both VMs: the `S-Probe` corresponds to the Ganglia probe; the `Log-Probe` is a simple software module that performs some anomaly analysis at the logs of the container; a `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 that server migration process no in-flight request is lost, since we have a window of execution where we have both servers running in active mode. When all the requests are finished in the main server, then the local SRA conducts a server restart. Before this, all session data should be migrated to the standby server and all the in-memory caches should be refreshed to the database. To implement the migration of session data we made use of some existing session-replication mechanism already existing in the application server we have used in our study (Tomcat) that is in fact quite similar to the replication schemes implemented by other containers. If we save all this session state to the standby server we might be able to achieve one of the fundamental features of crash-only software [40].

All these software modules were implemented by using open-source tools, like LVS, ldirectord and Ganglia. The deployment of this framework is straightforward and it does not require any change to the applications or the middleware containers. As soon as these software modules are made robust for third-users they will be made available for the scientific community.

4. Experimental Study

To study the behaviour of our rejuvenation scheme we have used two different application benchmarks and we have conducted an extensive set of experiments in a dedicated cluster.

4.1 Experimental Environment

4.1.1 Application Benchmarks

The applications were the following:

(1) **Tomcat/Axis**: in this case we have used 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 in accessed through SOAP. We have chosen Tomcat/Axis [41] since we already

knew from a previous study [21] that Axis v.13 is suffering from memory leaks;

(2) **TPC-W**: this is a Java implementation of the TPC-W benchmark that was implemented in CMU [42]. This container was also Tomcat and the database was MySQL. This benchmark resembles a online book store. 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 aging problem. This way, we have 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 synthetic technique to inject memory leaks is similar to the one described in [18].

4.1.2 The Workload-Test Tool

To speed-up the occurrence of software aging in our applications and to collect some performance measures we used a multi-client tool, called QUAKE [43] that was implemented in our Lab. 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 pre-defined 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 running the client benchmark application, one Database Server (Katrina or Wilma) and our main server (Tania or Nelma) running XEN and three virtual machines. All the machines are interconnected with a 100Mbps Ethernet switch. The detailed description of the machines is presented in Table 1.

We used Xen version 3.0.2 [31] configured with 3 virtual machines: 2 virtual machines with 700MB memory for the application servers and a third VM with 256MB memory to run LVS, ldirectord watchdog and our software modules. Both virtual machines run Linux 2.6.16.21-0.25-xen with 1024MB swap space and 1 virtual CPU. In some of the experiments (sub-sections 4.2.5 and 4.2.6) where we needed to test with two active application servers we used the Katrina server and 5 virtual machines.

	Servers: Katrina, Wilma	Servers: Tania, Nelma	Clients achines
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	

Table 1: Configuration parameters of the 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 it as the warm-up interval. So, the total run length was 10 minutes. The comparison of throughput is presented in Figure 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.

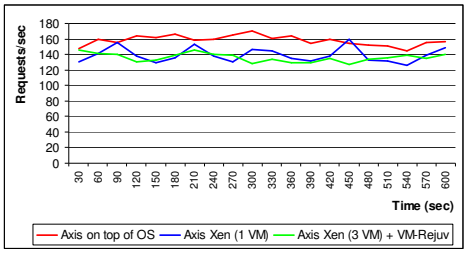

Figure 2: Comparing the Throughput of Tomcat/Axis Application on top of OS, on top of XEN, and on top of XEN using VM-Rejuv

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 introduces an overhead of 12% while our mechanism adds an additional overhead of 2%.

	Average Throughput (request/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%

Table 2: Comparing the Overhead (Tomcat/Axis)

The results for TPC-W are rather different. In this run, we have changed the code of the TPC-W clients to eliminate the random "think-time" between requests. The specification includes a random time between 7 and 70 seconds. In this run, we set the "think-time" to a fixed value, corresponding to the minimum of seven seconds. Results are presented in Table 3.

	Average Throughput (request/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%

Table 3: Comparing the Overhead (TPC-W)

As it can be seen, the overhead of using XEN is almost negligible. Since this application is not using a continuous burst distribution the resulting overhead is not significant.

4.2.2 Is `VM-Rejuv` an Effective Technique?

The next step was to evaluate the effectiveness of our automated rejuvenation mechanism. We used the two application benchmarks (Tomcat/Axis and TPC-W) and we configured the rejuvenation mechanism to be triggered when there is some threshold violation in one of the external QoS metrics. In this case we are observing the throughput of the application: 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 ten simultaneous clients we observed a crash in the application server in less than 5 hours. We measured the throughput in time-runs of 4 hours when using the application and no rejuvenation, comparing with two scenarios where we applied rejuvenation when there was a violation of the throughput SLA (Service-Level Agreement). In these experiments we set-up two values for the SLA: 50% and 75%. When the observed external throughput decreased to lower than 50% or 75% of the maximum value (observed when the system was running at the beginning) the `VM-Rejuv` applies a rejuvenation action. Results are presented in Figure 3. That 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 v1.3. Without the usage of our scheme the application would crash approximately after 4.5 hours of execution.

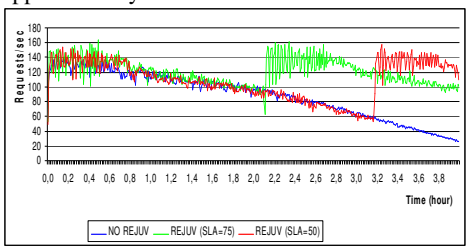

Figure 3: Applying Rejuvenation to Tomcat/Axis.

Figure 4 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 1kb per request, which was an aggressive measure. The idea was to observe very quickly the crash of the application to compare the effects of our rejuvenation mechanism. In Figure 4 we can observe that TPC-W application with that synthetic memory leak died after 1.475 hours of execution.

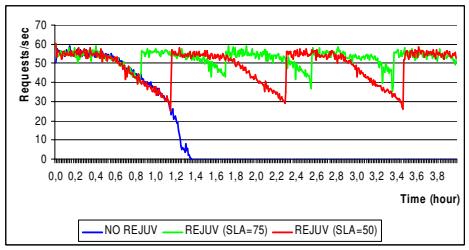

Figure 4: Applying Rejuvenation to TPC-W.

With the automated rejuvenation scheme we were able to avoid a crash and we kept a sustained level of performance, within the limits defined for the Throughput-SLA. This mechanism has proved to be very effective to achieve some self-healing capabilities for the application servers, without any additional hardware and any change in the applications.

4.2.3: What is the Downtime of our Rejuvenation Scheme?

In the next step, we want to evaluate the potential downtime for the application when there is a restart, as well as the number of failed-request and the potential loss of session data during the rejuvenation process. These results would be of extreme importance, according to guidelines (2) and (3) described in section 2.

We conducted four different experiments in the same target machine and using the Tomcat/Axis application:
(i) one run where we applied a rejuvenation action at the time of 300 seconds;
(ii) another run where we triggered a Tomcat restart at exactly that time;
(iii) a third one, were we applied a restart in the XEN Virtual Machine where the main server was running, at the exact same time (300 seconds);
(iv) and finally a last one, where we executed a full machine reboot at that execution time.

The results are presented in Figure 5. It 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 micro-rebooting, 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 XEN VM restarted resulted in a visible downtime of 56 seconds and finally a machine reboot represented a downtime of about 200 seconds.

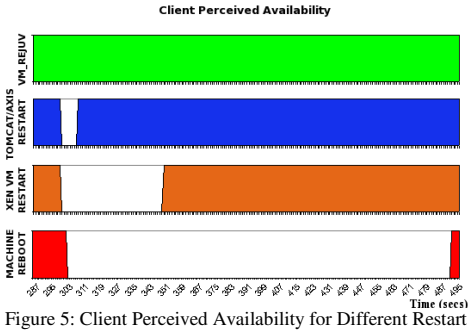

Figure 5: Client Perceived Availability for Different Restart Mechanisms (Tomcat/Axis).

The next step was to analyze the number of failed requests caused by one restart operation. Results are presented in Table 4.

	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

Table 4: Comparing Downtime and Failed Requests with different restart mechanisms.

Table 4 presents the average throughput during a test-run of 10 minutes, considering there was one restart operation, the total number of requests during that execution time, the number of failed requests, the number of slow requests and the perceived downtime, as measured from the client's side.

As it 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 is a very important achievement towards our guideline (3). The implementation of micro-rebooting produced a very low MTTR (even so, higher than zero) and it allowed some failed requests: as presented in a particular experiment in [29] a micro-reboot of a EJB component would still cause 78 failed requests. This means a micro-reboot 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 by George Candea in [29]: it corresponds to the SLA usually used by production web-sites as a maximum acceptable response time.

Curiously, there were no slow requests when we applied a VM-restart or a node reboot: those operations did cause refused connections that were counted as failed request. The time-out mechanism when the node is available (scenario 2 and 3) is quite different from when it is not available (scenario 4). This is related to the TCP-IP mechanism for time-outs, and explains why the number of failed requests in scenario 4 (machine reboot) was not proportional to the downtime.

Our main achievement in this experiment was a zero downtime of our rejuvenation mechanism. 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) to combat the software aging problem. Without considering unplanned failures, if we want to achieve a 99.999% 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 1 node-reboot in that whole year; or, 5 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 sub-section we do present some results to about the session-replication scheme of Tomcat. To avoid losing session data, we need to replicate the session-objects from the active to the backup server. This replication scheme would bring some overhead, but 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 8kb (typical session-object size is less than this value). The results are presented in Figure 6. The Figure presents two scenarios: (i) one with session replication; (ii) another without it. We just presented the last 750 seconds of the experiment, since we removed the warm-up time.

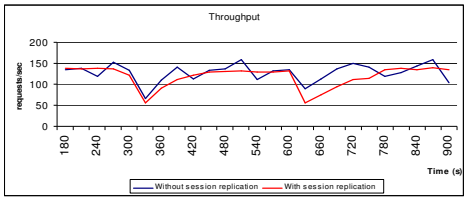

Figure 6: Comparing the overhead of session replication with session objects of 8kb (Tomcat/Axis).

In the first case we did not lose any session data and the application served a total of 117,750 requests, during those 15 minutes. In the second case we observed 15 session errors and the application was only able to serve 107,301 requests. This represents an overhead of 8.8%. With session objects of 1k we got an overhead of 4.9%. To summarize: there is some overhead and this is the price to pay if we want to maintain the consistency of the application.

4.2.5: Is our Rejuvenation Scheme useful in a Cluster Configuration?

So far we have been testing our rejuvenation in single servers. One pertinent 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 and support for IP fail-over when there is a server crash. And here is the critical point: clusters are usually used to tolerate failures, by using redundancy. 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 in-flight request, 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 ran the Tomcat/Axis application in a cluster configuration (using 3 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 Figure 7.

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.

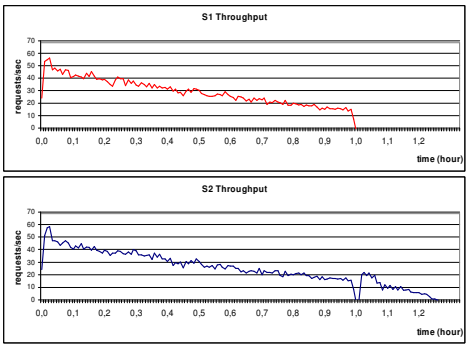

Figure 7: Throughput of two servers in a cluster, suffering from severe aging (Tomcat/Axis).

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.

In Figure 8 we can see the performance decay in the overall throughput of the cluster, down to the zero level. We can also see the overall throughput when using our rejuvenation scheme (with a SLA verification of around 75% of the throughput).

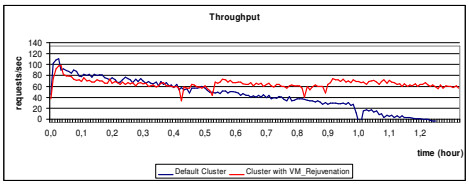

Figure 8: Comparing the cluster throughput, with and without rejuvenation and no automatic restart (Tomcat/Axis)

Our scheme is able to maintain a sustained level of performance: the cluster did not face any performance failure or a complete crash (as in the default case). There was no failed request when using rejuvenation in this experiment, while in the default fail-over mechanism of LVS 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. By using our rejuvenation scheme the cluster managed to serve 339,186 requests. This means that our mechanism promoted an increase of about 46% 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. The results of 5 hours of execution are presented in Figure 9. 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 observed an improvement of 66% in the overall performance.

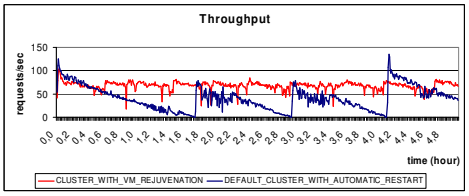

Figure 9: Comparing the cluster throughput, with and without rejuvenation, but automatic restart (Tomcat/Axis).

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 of the application running in the cluster.

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 that resemble a "blind" restart: when there is a decision to trigger a rejuvenation action the server is restarted without any extraordinary concern.

In this experiment we used a single server running XEN and we have configured 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 and a "blind" reboot of an active server. In the configuration of VM-Rejuv we needed to use 5 virtual-machines: one for the LB, two for the active servers and other two for the standby servers. In the two other scenarios you just needed to use 3 virtual machines: one for the LB and two for the active servers. Results are presented in Figure 10.

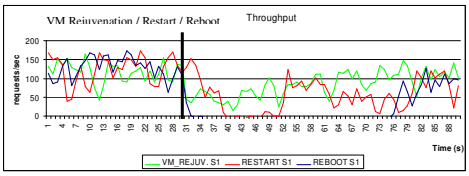

Figure 10: The impact of a "clean" restart when compared with a "blind" restart and a "blind" reboot (Tomcat/Axis).

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 much 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. And this is why we see that fall in the throughput curves.

5. Conclusions

In this paper, we have presented a simple approach for enhanced software rejuvenation that has proved to be highly effective: 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 penalty. It can be applied to single-server or cluster configurations without any additional cost. We just require the use of a virtualization layer and the installation of some software modules.

The achievements of this study allow us to think that this approach to software rejuvenation can be a partial contribution for the development of critical technologies to add self-healing capabilities to off-the-shelf Enterprise Servers and Web-Servers.

Acknowledgments

This research work is carried out in part under the FP6 Network of Excellence Core-GRID funded by the European Commission (Contract IST-2002-004265) and by the Ministry of Science and Technology and European Union (FEDER Funds) under contract TIN2004-07739-C02-01.

Bibliography

[1] J.Hennessy, D.Patterson, "*Computer Architecture: A Quantitative Approach*", Morgan & Kaufmann Publishers, 2002.

[2] E.Marcus, H.Stern, "*Blueprints for High Availability*", Wiley, 2003

[3] J.Kephart, D.M.Chess. "*The Vision of Autonomic Computing*", IEEE Computer, 36(1), 2003

[4] Y.Huang, C.Kintala, N.Kolettis, N. Fulton. "*Software Rejuvenation: Analysis, Module and Applications*", Proceedings of Fault-Tolerant Computing Symposium, FTCS-25, June 1995

[5] A.Avritzer, E.Weyuker. "*Monitoring Smoothly Degrading Systems for Increased Dependability*", Empirical Software Eng. Journal, Vol 2, No 1, pp. 59-77, 1997

[6] Apache: http://httpd.apache.org/docs/

[7] Microsoft IIS: http://www.microsoft.com/

[8] V.Castelli, R.Harper, P.Heidelberg, S.Hunter, K.Trivedi, K.Vaidyanathan, W.Zeggert. "*Proactive Management of Software Aging*", IBM Journal Research & Development, Vol. 45, No. 2, Mar. 2001

[9] K.Cassidy, K.Gross, A.Malekpour. "*Advanced Pattern Recognition for Detection of Complex Software Aging Phenomena in Online Transaction Processing Servers*", Proc. of the 2002 Int. Conf. on Dependable Systems and Networks, DSN-2002

[10] A.Tai, S.Chau, L.Alkalaj, H.Hecht. "*On-board Preventive Maintenance: Analysis of Effectiveness an Optimal Duty Period*", Proc. 3rd Workshop on Object-Oriented Real-Time Dependable Systems, 1997

[11] E.Marshall. "*Fatal Error: How Patriot Overlooked a Scud*", Science, p. 1347, Mar.1992

[12] MemProfiler: http://memprofiler.com/

[13] Parasoft Insure++: http://www.parasoft.com

[14] K.Vaidyanathan, K.Trivedi. "*A Comprehensive Model for Software Rejuvenation*", IEEE Trans. on Dependable and Secure Computing, Vol, 2, No 2, April- 2005

[15] S. Garg, A. van Moorsel, K. Vaidyanathan, K. Trivedi, "*A Methodology for Detection and Estimation of Software Aging*", Proc. 9th 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. Symp. Software Reliability Eng., pp. 84-93, 1999.

[17] L.Li, K.Vaidyanathan, K.Trivedi. "*An Approach for Estimation of Software Aging in a Web-Server*", Proc. of the 2002 International Symposium on Empirical Software Engineering (ISESE'02)

[18] K.Gross, V.Bhardwaj, R.Bickford. "*Proactive Detection of Software Aging Mechanisms in Performance Critical Computers*", Proc. 27th Annual IEEE/NASA Software Engineering Symposium, 2002

[19] K.Kaidyanathan, K.Gross. "*Proactive Detection of Software Anomalies through MSET*", Workshop on Predictive Software Models (PSM 2004), Sept. 2004

[20] K.Gross, W.Lu. "*Early Detection of Signal and Process Anomalies in Enterprise Computing Systems*", Proc. 2002 IEEE Int. Conf. on Machine Learning and Applications, ICMLA, June 2002

[21] L.Silva, H.Madeira and J.G.Silva "*Software Aging and Rejuvenation in a SOAP-based Server*", IEEE-NCA: Network Computing and Applications, Cambridge USA, July 2006

[22] L.Bernstein, Y.D.Yao, K. Yao. "*Software Avoiding Failures even when there are faults*", DoD SoftwareTech News, Oct. 2003, Vol.6, Bo.2, pp. 8-11

[23] A.Andrzejak, L.M.Silva. "*Deterministic Models of Software Aging and Optimal Rejuvenation Schedules*", 10th IFIP/IEEE Int. Symposium on Integrated Network Management (IM 2007), May 2007, Munich, Germany

[24] G.Candea, A.Brown, A.Fox, D.Patterson. "*Recovery Oriented Computing: Building Multi-Tier Dependability*", IEEE Computer, Vol. 37, No. 11, Nov.2004

[25] G.Candea, E.Kiciman, S.Zhang, A.Fox. "*JAGR: An Autonomous Self-Recovering Application Server*", Proc. 5th Int Workshop on Active Middleware Services, Seattle, June 2003

[26] A.Fox, D.Patterson. "*When Does Fast Recovery Trump High Reliability?*", Proc. 2nd Workshop on Evaluating and Architecting System Dependability, CA, 2002

[27] R. Figueiredo, P. Dinda, J. Fortes, "*Resource Virtualization Renaissance*", IEEE Computer, 38(5), pp. 28-69, May 2005

[28] M. Rosenblum and T. Garfinkel, "*Virtual Machine Monitors: Current Technology and Future Trends*," IEEE Internet Computing, May 2005, Vol. 38, No. 5.

[29] G.Candea, S.Kawamoto, Y.Fujiki, G.Friedman, A. Fox. "*Microreboot – A Technique for Cheap Recovery*", Proc. 6th Symp on Operating Systems Design and Implementation (OSDI), Dec 2004

[30] VMware, http://www.vmware.com

[31] Xen, http://www.xensource.com

[32] Virtuoso, http://www.virtuoso.com

[33] LVS, http://www.linuxvirtualserver.org/

[34] ldirectord, http://www.vergenet.net/linux/ldirectord

[35] Ganglia, http://ganglia.sourceforge.net

[36] Essential about Java Servlet Filters: http://java.sun.com/products/servlet/Filters.html

[37] D.Menascé. "*QoS Issues in Web Services*", IEEE Internet Computing, Nov-Dec 2002

[38] R.Arpaci-Dusseau, A.Arpaci-Dusseau, "*Fail-Stutter Fault Tolerance*", Proc. 8th Workshop on Hot Topics in Operating Systems, (HOTOS-VIII), 2001

[39] S.Makridakis, S.Wheelwright, R.Hyndman. "*Forecasting: Methods and Applications*", Third-Edition, John Wiley & Sons, 1998

[40] G.Candea, A.Fox. "*Crash-only Software*", 9th Workshop on Hot Topics in Operating Systems, 2001

[41] Apache Axis, http://ws.apache.org/axis

[42] TPC-W in Java for Tomcat and MySQL: http://www.cs.cmu/edu/~manjhi/tpcw.html

[43] S.Tixeuil, W.Hoarau, L.M. Silva, "*An Overview of Existing Tools for Fault-Injection and Dependability Benchmarking in Grids*", CoreGRID Technical Report, TR-0041, http://www.coregrid.net