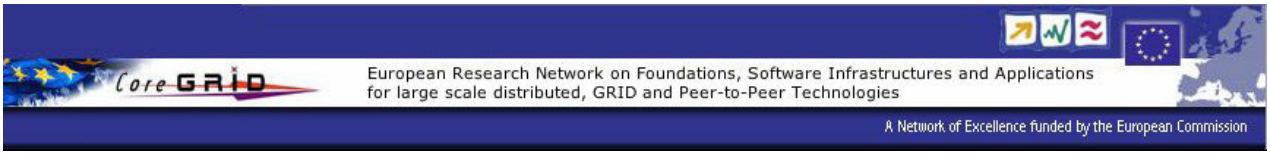

Using Micro-Reboots to Improve Software Rejuvenation in Apache Tomcat

Paulo Silva[1], *Luis Silva*[1], *Artur Andrzejak*[2]

[1]*CISUC - Centre for Informatics and Systems of the University of Coimbra*
Polo II - Pinhal de Marrocos
3030-290 Coimbra, Portugal
luis@dei.uc.pt

[2]*Zuse-Institute Berlin*
Takustr. 7, 14195 Berlin, Germany
andrzejak@zib.de

CoreGRID Technical Report
Number TR-0099
September 17, 2007

Institute on Architectural issues: scalability, dependability, adaptability (SA)

CoreGRID - Network of Excellence
URL: http://www.coregrid.net

CoreGRID is a Network of Excellence funded by the European Commission under the Sixth Framework Programme

Project no. FP6-00426

Using Micro-Reboots to Improve Software Rejuvenation in Apache Tomcat

Paulo Silva[1], Luis Silva[1], Artur Andrzejak[2]

[1]CISUC - Centre for Informatics and Systems of the University of Coimbra
Polo II - Pinhal de Marrocos
3030-290 Coimbra, Portugal
luis@dei.uc.pt

[2]Zuse-Institute Berlin
Takustr. 7, 14195 Berlin, Germany
andrzejak@zib.de

CoreGRID TR-0099[*]

September 17, 2007

Abstract

As software complexity increases so does the difficulty in solving all software defects before the production stage, even with advanced software testing tools. Those software defects are often the cause for application crashes. To tolerate application crashes the industry has adopted several clustering techniques: server-redundancy, load-balancers and server-failover. The latest trend goes towards the development of self-healing techniques that automate the recovery procedures and prevent the occurrence of unplanned failures. In recent years a particular software problem has been studied: software aging. Software aging is described as the progressive degradation of the running software that may lead to crashes. To solve software aging a mechanism has been proposed: software rejuvenation. In essence software rejuvenation is a restart operation that causes the software to return to maximum performance thus avoiding the software crash. In this report we study an enhanced rejuvenation technique: Micro-Rebooting. A Micro-Reboot is a reboot done in a more fine-grained component than the whole application. Our experimental study aims to fill that void by studying the feasibility of using Micro-Reboots in Apache Tomcat. We implemented a prototype Micro-Reboot framework in that Web-Server and did an experimental study to evaluate its effectiveness.

[*] *This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).*

1 Introduction

In recent years the industry and academia have paid attention to the important issue of availability in business-critical applications. These applications have increased in complexity, making the already difficult goal of zero software defects almost impossible to reach. Even with advanced commercial tools [1-2] to detect the root cause of software bugs some defects are not easily spotted. To approach 100% availability, [4] explains that it is easier to reduce the MTTR (Mean-Time-To-Repair) close to zero than paying efforts to increase the MTBF (Mean-Time-Between-Failures). Particularly in Internet systems, long-running applications, it is very difficult to guarantee that the MTBR will be equal or bigger than the total execution time of the application.

Two particular types of failures have been largely identified [5]: transient failures and software aging. Internet systems have been identified [5] as highly prone to transient failures, which can be quite difficult to spot and predict due to their indeterminist behaviour. Software aging describes the phenomena of progressive degradation of the running software that may lead to system crashes [6]. Software aging has been observed not only in desktop operation systems but in other systems: telecommunication [7], web-servers [8-10], enterprise clusters [11], OLTP systems [12] and spacecraft systems [13]. This problem has even been reported in military systems [14] causing the loss of lives.

The latest trend to achieve higher availability is the development of self-healing techniques [3] that automate the recovery procedures and whenever possible would prevent the occurrence of unplanned failures. The most natural procedure to combat software aging is to apply the well-know technique of software rejuvenation [6]. Software rejuvenation is achieved by restarting the application, thus restarting its performance to the standard levels and effectively solving software aging.

One enhanced rejuvenation technique has been introduced by the ROC Project from Stanford and Berkeley [15]: Micro-Rebooting. Micro-Reboots are more fine-grained restarts: instead of restarting the whole application we restart a smaller application component thus allowing for faster recovery and lower disruption to end-users. Their motivation was to lower the MTTR thus lowering the end-user noticed service disruption. Micro-Rebooting proved [17-19] to be a very effective technique to solve both transient failures and software aging.

Motivated by the good results of Micro-Rebooting when applied to the JBoss [16], we decided to implement a prototype Micro-Reboot framework applied to the popular web-server: Apache Tomcat [20] and conduct an experimental study to evaluate its effectiveness in this scenario.

2 Micro-Reboot Framework

The design of our Micro-Reboot framework started by devising some orienting guidelines:

1. It should be easily applied to Apache Tomcat without requiring any knowledge of the installed web-applications or their re-engineering.
2. The mechanism should provide a fast recovery time by doing a more fine-grained restart compared to a full Web-Server restart. A faster recovery means that a higher availability could be reached by reducing the MTTR.
3. Micro-Rebooting one web-application should not disrupt other web-applications in the Web-Server.
4. The disruption to the end-users during a Micro-Reboot should be minimal as the overhead of the framework.
5. The framework should automate the rejuvenation scheme in order to achieve a self-healing system.
6. The mechanism should be easy to deploy and maintain in complex IT systems.

Based on those guidelines we then designed the framework, shown in Figure 1.

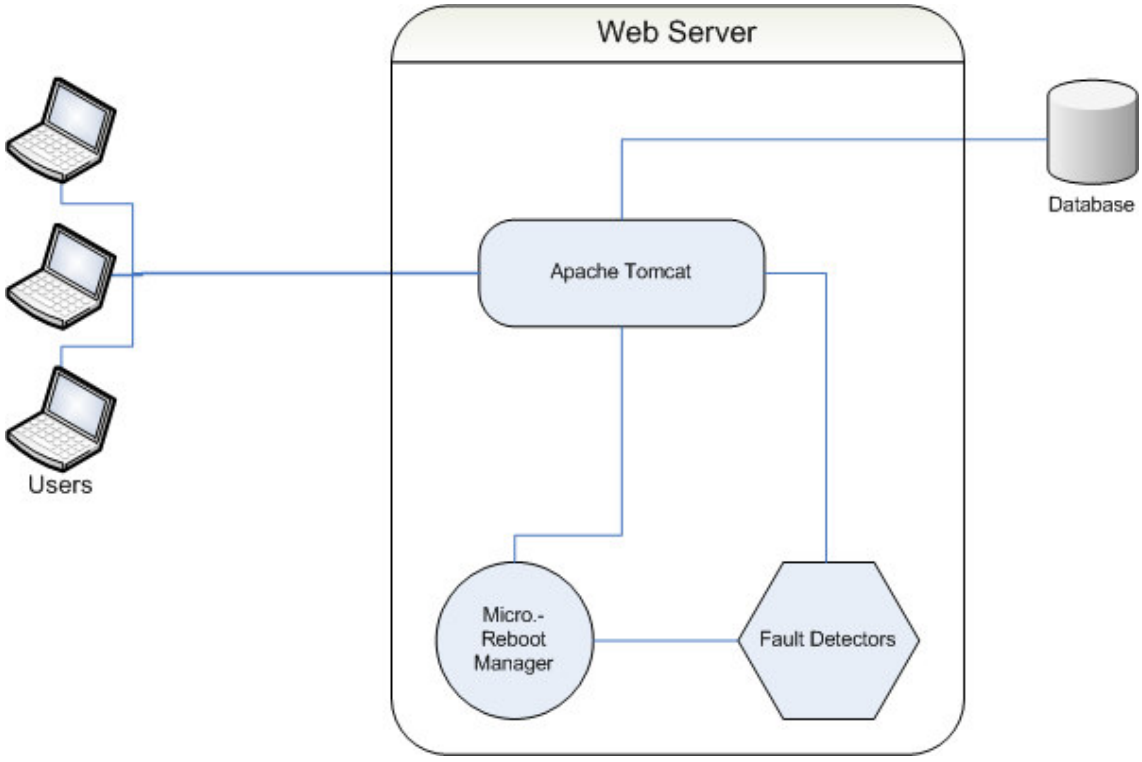

Figure 1: Micro-Reboot framework

To the usual scenario in a Web-Server we added the fault detectors, the Micro-Reboot Manager and slight modifications to Apache Tomcat.

In Apache Tomcat the Micro-Reboot was implemented by slightly improving an existing Manager web-application [21]. That Manager can be used to deploy, start, stop or reload web applications. To do the Micro-Reboot we use the reload feature of the Manager with slight modifications to guarantee that it cleanly stops the web-application and then restarts it. In Apache Tomcat a web-application is loosely translated to one web-site, which means in a shared hosting environment, Apache Tomcat usually hosts several web-applications.

As our focus was in the study of the software rejuvenation we used a simple set of fault detectors. Those fault detectors constantly monitor Apache Tomcat and the target system, analysing vital data like memory and CPU usage. Another important fault detector is the component analyzer, described in [22], that analyses thrown exceptions in the web-application components.

The Micro-Reboot Manager (MRM) controls when a Micro-Reboot should be done and to which application. When it receives an UDP message from the fault detectors it analyses it, and decides if a Micro-Reboot should be done. If the MRM receives a fault of the same type, to the same web-application and it is already Micro-Rebooting that application, it obviously ignores that message. When the MRM decides to apply a Micro-Reboot it contacts the Apache Tomcat Manager and issues a reload operation on the desired web-application.

During the experimental study we noticed that, while Micro-Reboots are fast, there are still some missed requests that occur during the Micro-Reboot. To avoid those, we devised a new version for our framework, the Micro-Reboot Framework version 2, presented in Figure 2.

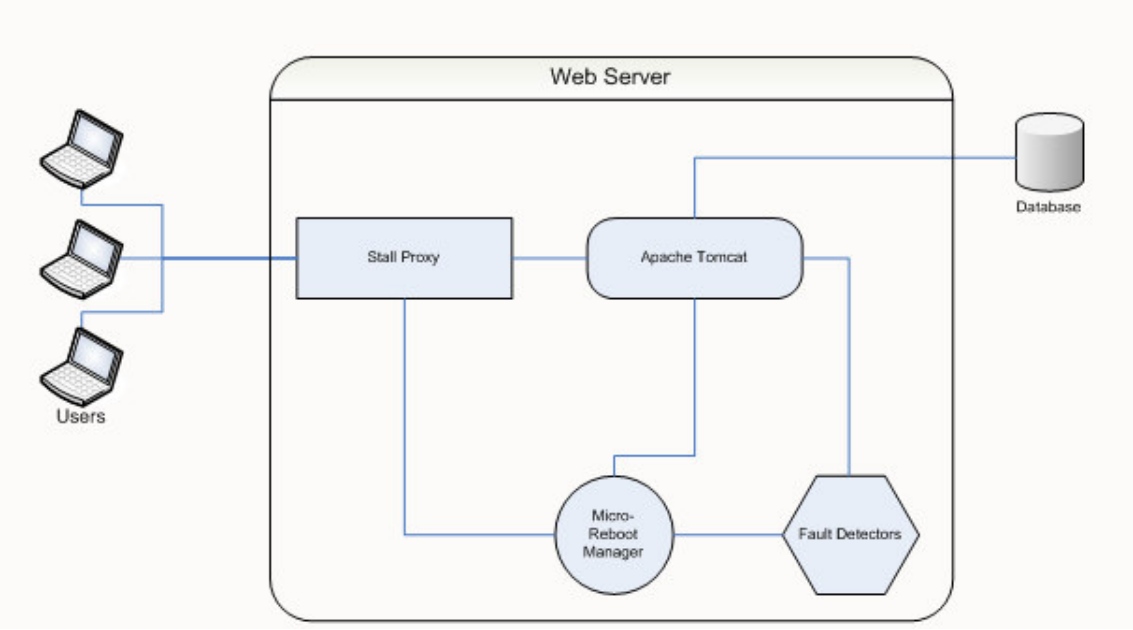

Figure 2: Micro-Reboot Framework version 2

This new version adds a stall proxy between end-users and the Web-Server that is used to hold the requests during a Micro-Reboot, to avoid missed requests during the rejuvenation operation. Any request that reaches the proxy when a Micro-Reboot is executing is held until the Micro-Reboot ends. That way, the end-user instead of receiving a missed request, because a Micro-Reboot is taking place, only notices that the request is slower because of the waiting period.

When the Micro-Reboot Manager (MRM) receives the fault detection message from the fault detectors it first informs the Stall Proxy to stop passing requests to the Web-Server. Only after that the Micro-Reboot is executed. The Micro-Reboot Manager then checks Apache Tomcat, to detect when the Micro-Reboot has ended (by trying to access the web-application), and when the Web-Server is ready a message to the proxy is sent, so that requests can resume.

We implemented the prototype version of the Micro-Reboot framework and proceeded to its evaluation.

3 Experimental Environment

For our set of experiments we used the JPetStore application benchmark. To achieve meaningful results we ran every scenario, at least, 10 times and we present the average results. In this section we describe the whole experimental environment.

3.1 Tomcat/JPetStore

To access the validity of micro-reboot in Apache Tomcat we used iBatis JPetStore [23], a pet store e-commerce website. JPetStore The customer can browse through a catalog of pets that vary by category and pet type. If the customer sees an item he likes, he can add it to his shopping cart. When the customer is done shopping; he can checkout by submitting an order that includes payment, billing and shipping details. Before the customer can checkout, they must sign in or create a new account. PetStore 1.1 [24] was made available by Sun as a sample J2EE application and originated many Pet Store versions using different technologies. iBatis JPetStore is a 3-tier application making use of JSP and Struts in its presentation layer, JavaBeans in its business layer and DAO in the data access layer. For our experiment JPetStore was configured to use a MySQL database back-end.

3.2 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 [25] 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 our study, we just

used the burst distribution, to speed up the occurrence of software aging. Burst distribution means that clients send requests without any interval between them, thus executing the maximum possible number of requests per second.

For our experimental study we adapted the QUAKE tool to support the JPetStore application benchmark. Our benchmark simulates a usual browsing session made by a user. The benchmark uses all available JPetStore operations, effectively covering all JPetStore components. The benchmark is done in burst mode, which means clients do not have a waiting period between requests.

3.3 Experimental Setup

In our experiments we used a part of the CISUC cluster composed by 3 client machines running the Quake client benchmark tool and one server, Wilma. All machines are interconnected with a 100Mbps Ethernet switch. The detailed description of the machines is presented in Table 1.

	Server: Wilma	Client machines
CPU	Dual AMD64 Opteron (2000MHz)	Intel Celeron (1000MHz)
Memory	4GB	512MB
Hard Disk	160GB (SATA2)	
Swap Space	8GB	1024MB
Operating System	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-b05 Standard Edition
Tomcat JVM heap size	1024MB	
Other software	Apache Tomcat 5.5.20, MySQL 5.0.18	

Table 1: Detailed experimental setup description

3.4 Fault Detection and Injection

An orthogonal problem to software rejuvenation is the fault detection and diagnosis. As in our Lab we had another research work focused on fault detection, we used the fault-detectors proposed in that work. The fault detectors include system-metric monitoring, log analyzers and a component analyzer that monitors Java Exceptions thrown by the web application.

The same research work also led to the implementation of a fault injection tool, JAFL, that we conveniently used to inject faults in JPetStore during our experimental study.

4 Experimental Results

4.1 What is the overhead of using the Micro-Rebooting framework?

In our first experiment we measure the performance penalty of the micro-reboot framework. We individually assess the overhead of the fault detectors and of the whole framework. Tomcat/JPetStore benchmark was used for this experiment. We run several short-time runs (15 minutes) and we remove the first 5 minutes, considering it is the warm-up period. The total run length was then 10 minutes.

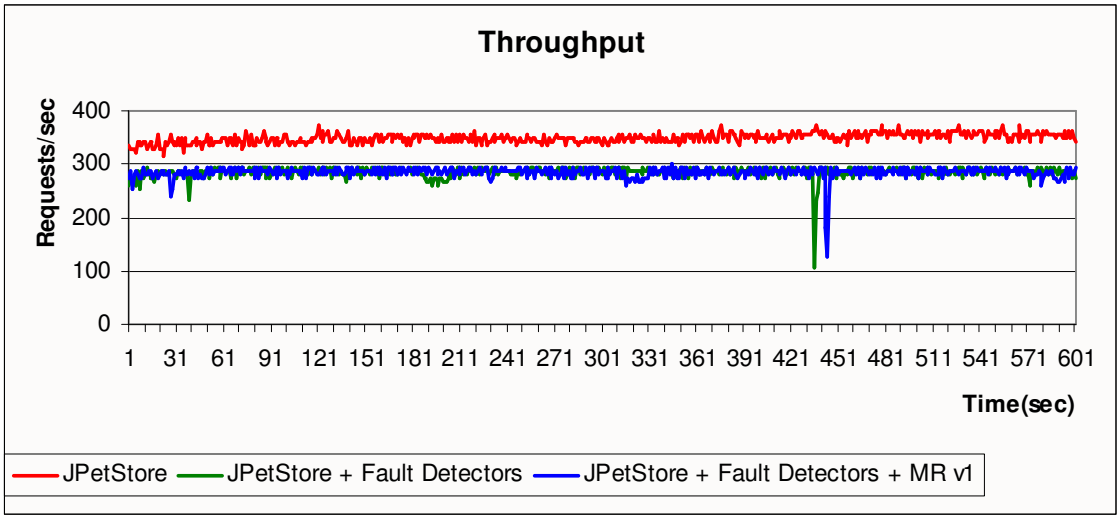

Figure 3: Comparing throughput of Tomcat/JPetStore benchmark plus fault detectors and with the whole Micro-Reboot v1 framework

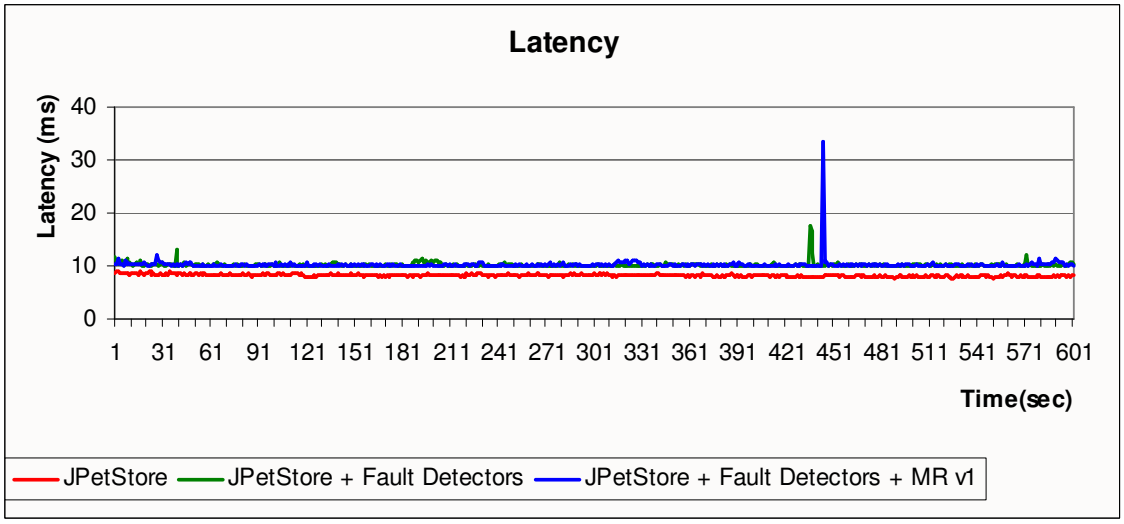

Figure 4: Comparing latency of Tomcat/JPetStore benchmark plus fault detectors and with the whole Micro-Reboot v1 framework

Figure 3 and Figure 4 presents the throughput and latency comparison. We can clearly see the overhead caused by the Fault Detectors. Adding the Micro-Reboot Manager does not add a significant overhead. Table 2 shows more detailed numbers from these experiments. We measure overhead in terms of total number of requests served.

	JPetStore	**JPetStore with Fault Detectors**	**JPetStore with Fault Detectors and Micro-Reboot framework v1**
Total Number Requests	210049	170863	170639
Average Throughput (requests/sec)	350.08	284.77	284.40
Average Latency (ms)	8.22	10.21	10.23
Overhead		18.66%	18.76%

Table 2: Comparing the Overhead of Tomcat/JPetStore

We can see that the introduction of the Fault Detectors causes an 18.66% overhead, while adding the Micro-Reboot Manager has a negligible overhead.

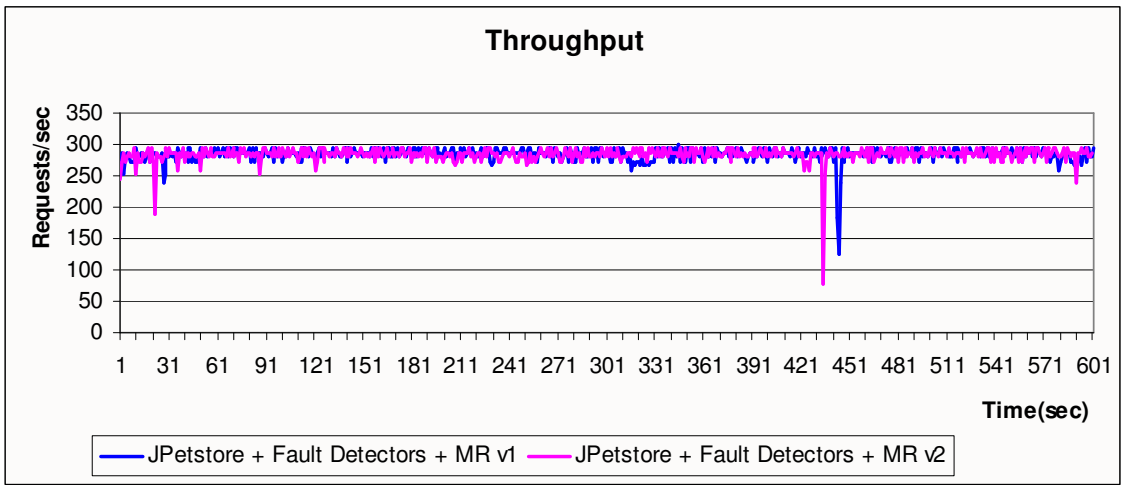

Figure 5: Comparing throughput of Tomcat/JPetStore benchmark with Micro-Reboot framework v1 and v2 (with an added stall proxy)

	JPetStore with Fault Detectors and Micro-Reboot framework v1	JPetStore with Fault Detectors and Micro-Reboot framework v2
Total Number Requests	170639	170520
Average Throughput (requests/sec)	284.40	284.20
Average Latency (ms)	10.23	10.25
Overhead		0.07%

Table 3: Comparing the Overhead of Tomcat/JPetStore with Micro-Reboot framework v1 and v2

The Micro-Reboot framework version 2 adds a stall proxy. Figure 5 and Table 3 shows the comparison between version 1 and version 2. In Table 3 the overhead is measured in terms of total number of requests served. As we can see the introduction of the proxy adds a minimal overhead.

These results were obtained while using a burst workload so they should be seen as the maximum observable overhead.

4.2 Is Micro-Rebooting an effective technique?

To be considered useful, our Micro-Rebooting framework needs to effectively apply an automated rejuvenation action thus solving the observed fault. To evaluate this we configured the fault injector to throw exceptions in a JPetStore component, namely the ViewCategory operation in the CatalogBean. After 120 seconds of usage the fault injector starts injecting faults so the ViewCategory operation always returns an exception. In this experiment we used a NullPointerException but similar results were obtained with other types of Exceptions. This behaviour simulates a transient fault.

We then ran a 10 minute experiment. If the web-server is successfully rejuvenated the fault injector is restarted and only after 120 seconds another fault is injected. This means that in the 10 minute run, we can observe 4 periods where the fault injector is active (120, 240, 360 and 480 seconds).

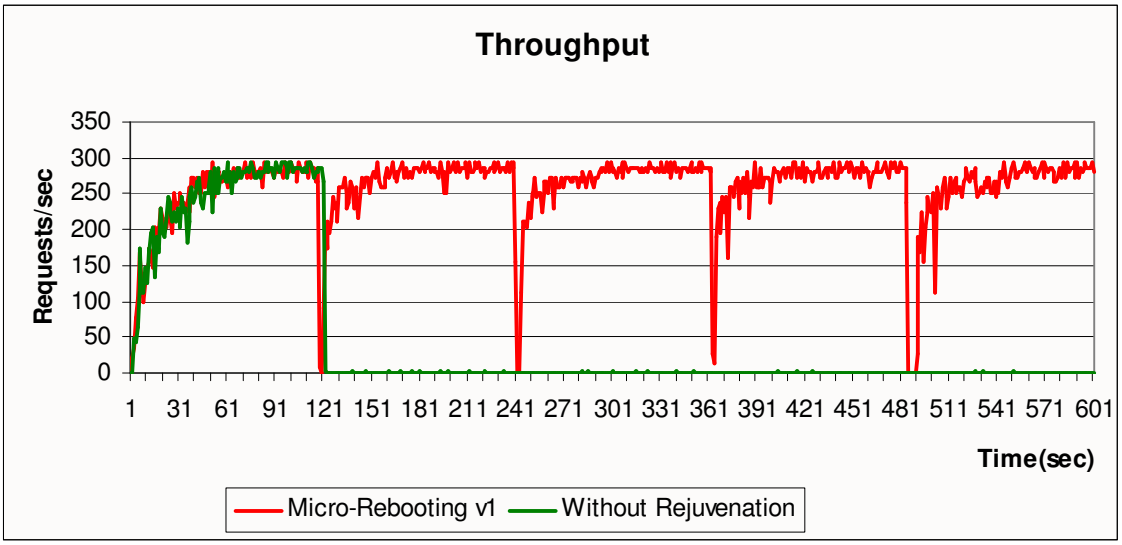

Figure 6: Applying Micro-Reboots to Tomcat/JPetStore with a transient fault

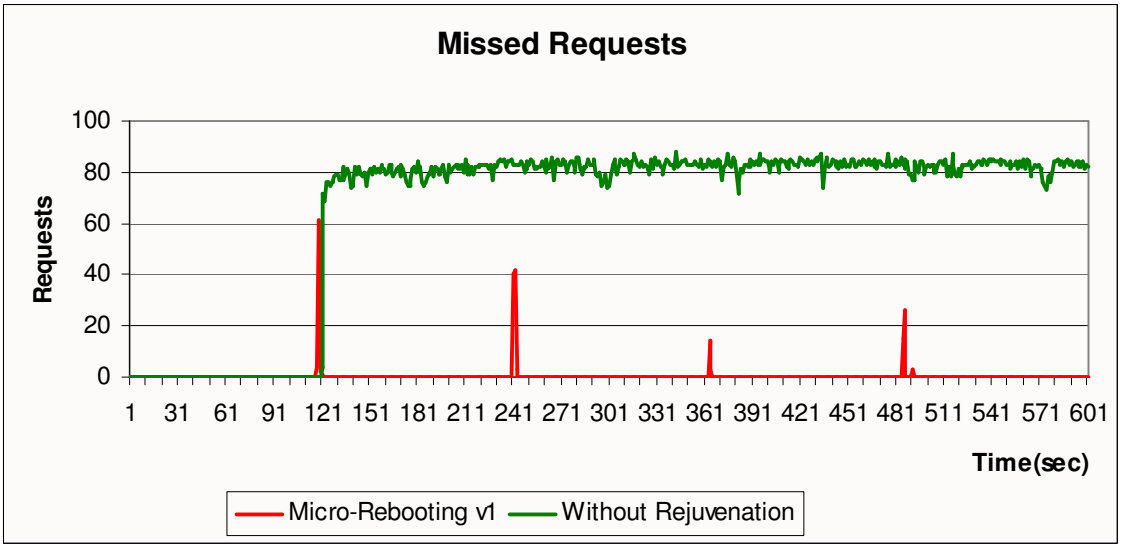

Figure 7: Comparing missed requests with and without the Micro-Reboot framework while applying a transient fault in Tomcat/JPetStore

Figure 6 clearly shows the effectiveness of our Micro-Reboot framework. Without it after 120 seconds we see a throughput drop to very minimal levels, between zero and 5 requests per second. Throughput is not always zero because some requests do not use the component where the fault is injected. Using Micro-Reboots we can see that the throughput drops while the Micro-Reboot takes place but after that the maximum performance is restored.

Figure 7 show the number of missed requests. In this experiment, without any rejuvenation, we measured 275072 missed requests against 209 obtained when Micro-Reboot rejuvenation is applied. But the obtained missed requests when Micro-Rebooting are not all rejuvenation related. There is a small time-window between the injection of the fault and the fault detection, that causes missed requests. That is why fault detection and diagnosis is an orthogonal problem to software rejuvenation.

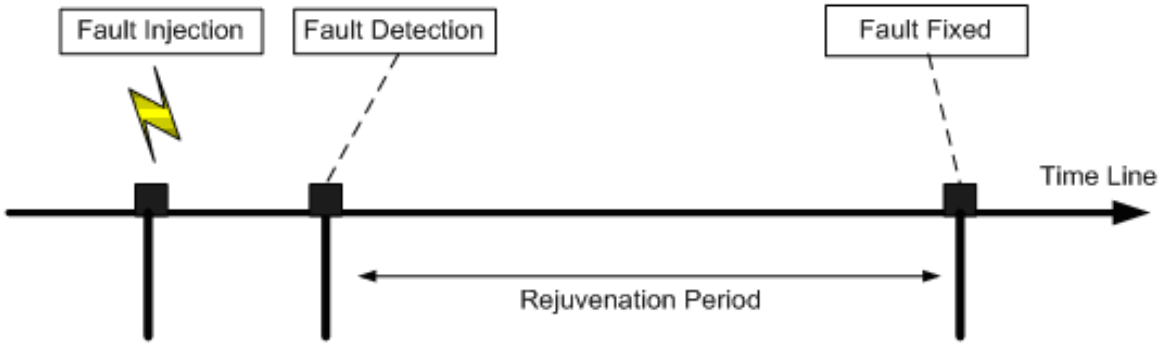

Figure 8 shows the time line where we exemplify this behaviour. Between the fault injection and fault detection there is a gap. Between fault detection and the start of the rejuvenation the gap is minimal. In our experiments that was lower than 10ms, whereas the latency of the fault detection was around 1 second.

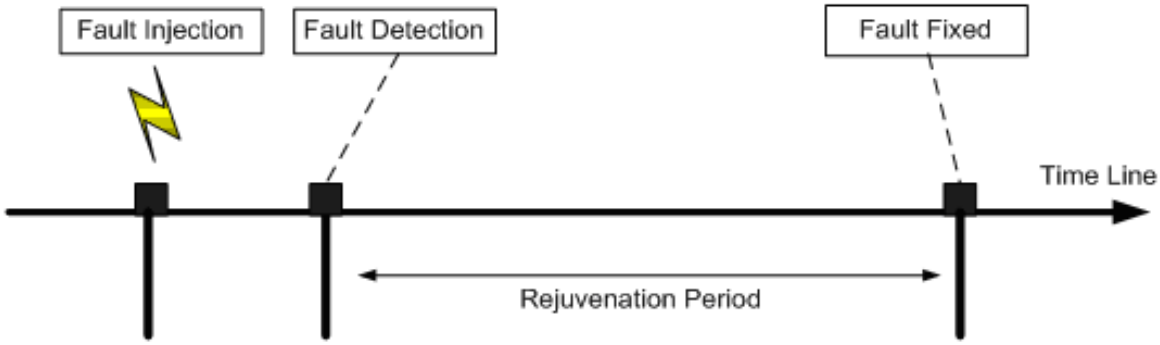

Figure 8: Fault injection versus fault detection time line

From the 209 missed requests obtained with Micro-Rebooting only 36 are caused because of the software rejuvenation operation. The remaining 173 were obtained during the fault injection/detection window.

To access the effectiveness of Micro-Reboots in other category of faults we used the memory consumption fault injector to simulate a software aging behaviour. After 120 seconds we inject a memory leak of 512KB per request, which is a very aggressive measure. Our purpose was to accelerate the crash of the application to observe the effects of our rejuvenation mechanism. The fault detector that monitors Tomcat memory usage was configured to send an alert when memory usage reached 900MB.

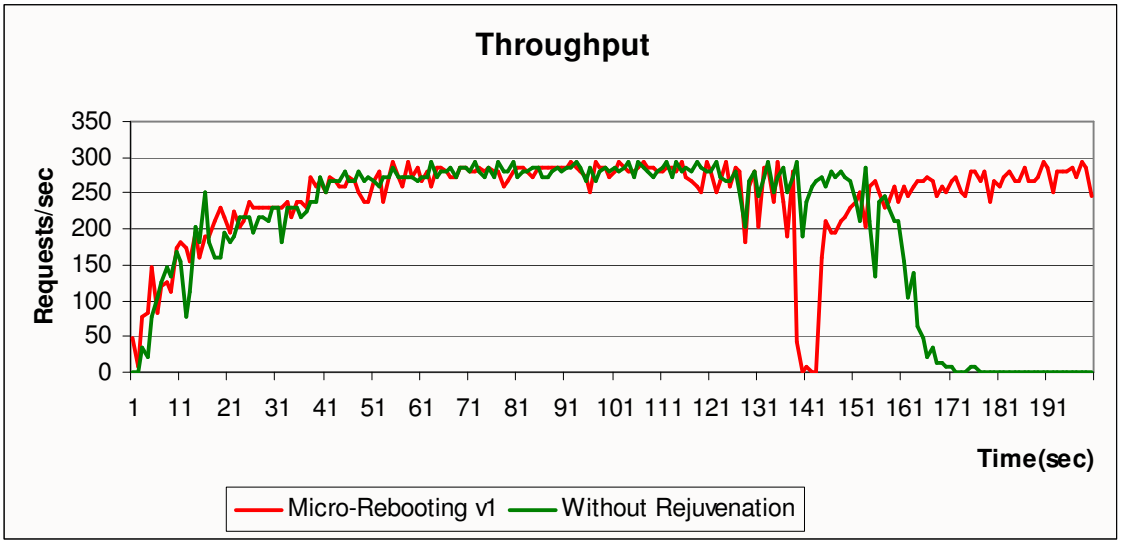

Figure 9: Applying Micro-Reboot to Tomcat/JPetStore with a memory exhaustion fault

As the fault detector is configured to send an alert before the maximum memory usage is reached, 1GB, we can apply the rejuvenation operation before the fault crashes Apache Tomcat. [Candea2004b] coined this as *microrejuvenation* because Micro-Reboots are used to prevent the web-server crash, before the end-user notices the problem.

Figure 9 presents the results of this experiment. We can see that if no rejuvenation is done, the throughput degrades over the time, until it reaches 0. Without Micro-Reboot we measured 638 missed requests against only 6. After the Micro-Reboot the throughput returns to the maximum performance.

4.3 Micro-Rebooting one web-application disrupts other web-applications?

It is very common to have several web-applications per Web-Server, usually called shared hosting. Our Micro-Reboot approach aims to be able to restart only the faulty web-application without disruption to the other web-applications running in the same Web-Server.

To test this claim we installed two JPetStore instances in the same Web-Server and configured 3 additional clients that executed requests to the second instance. After 120 seconds we Micro-Reboot one of the JPetStore instances. The throughput results are show in Figure 10.

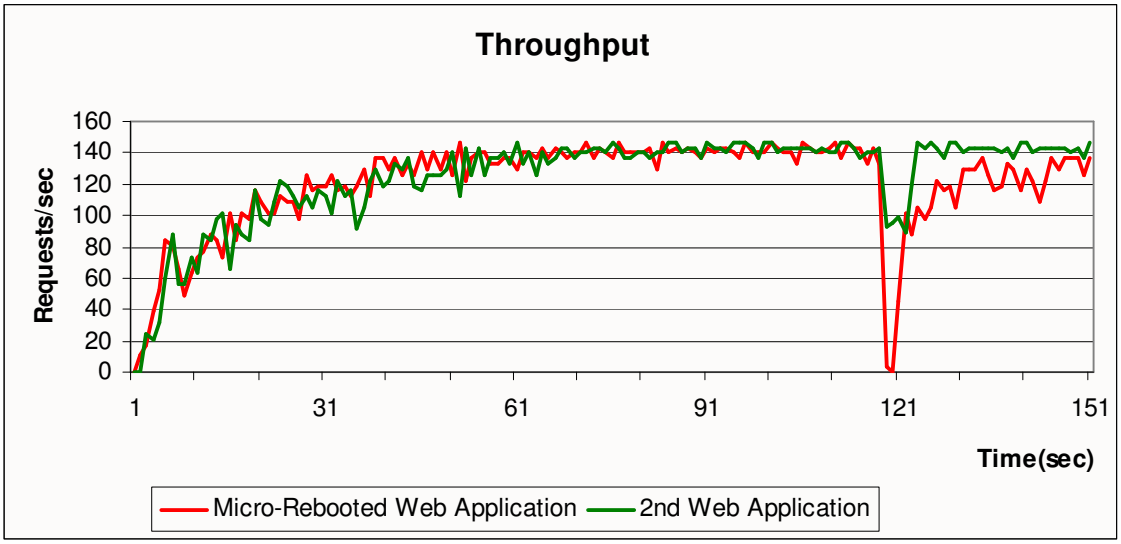

Figure 10: Comparing the throughput of two web-applications installed in the same Web-Server while one is Micro-Rebooted

We can see that the Micro-Rebooted web-application has a throughput drop when the Micro-Reboot is executed but in the second web-application the drop does not reach zero. The second application also does not have any missed requests. Still, we observe a small performance drop because when Apache Tomcat Micro-Reboots an application it uses CPU resources thus limiting the resources available to the other web-application. At most and in an overloaded Web-Server we can expect that Micro-Rebooting one web-application causes requests to other web-applications to be slower.

4.4 Is Micro-Rebooting better than a full Web-Server Restart?

We introduced Micro-Reboots as a better rejuvenation technique compared to a full Web-Server restart. To evaluate this we ran a similar experiment to section 4.2 but, instead of doing a Micro-Reboot, we execute a full Web-Server restart when a fault is detected.

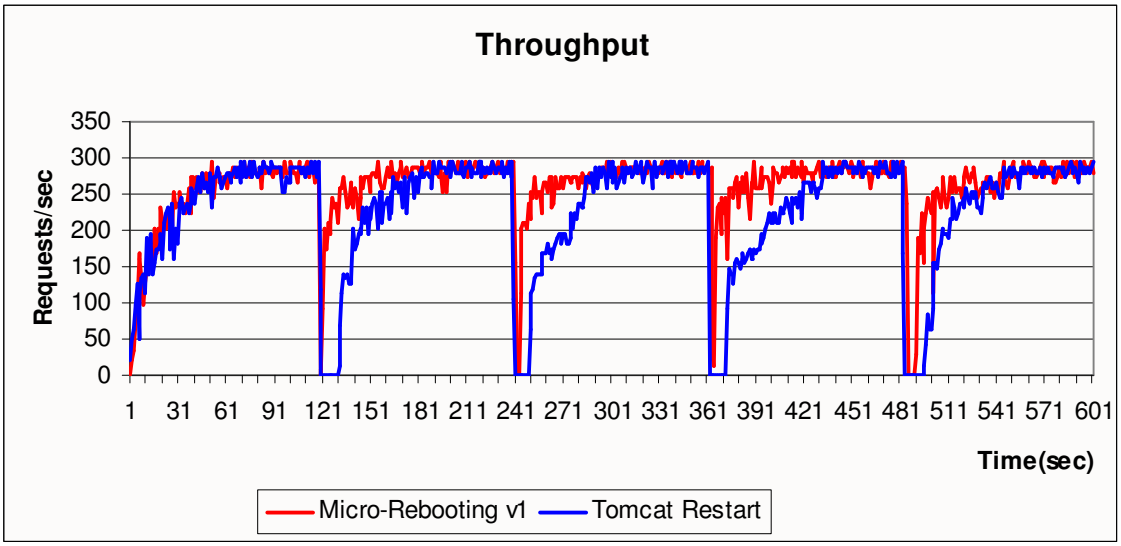

Figure 11: Comparing the throughput of a full Web-Server restart against Micro-Rebooting with a transient fault in Tomcat/JPetStore

Figure 11 presents the results obtained with a transient fault. When a Tomcat restart is used we notice a larger period of time where the throughput is very low. In this 10 minute experiment using a full Web-Server restart causes 14434 missed requests against 1498 of the Micro-Reboot solution. We can also notice that after a restart the warm-up period (the time to achieve maximum performance) is longer than with a Micro-Reboot.

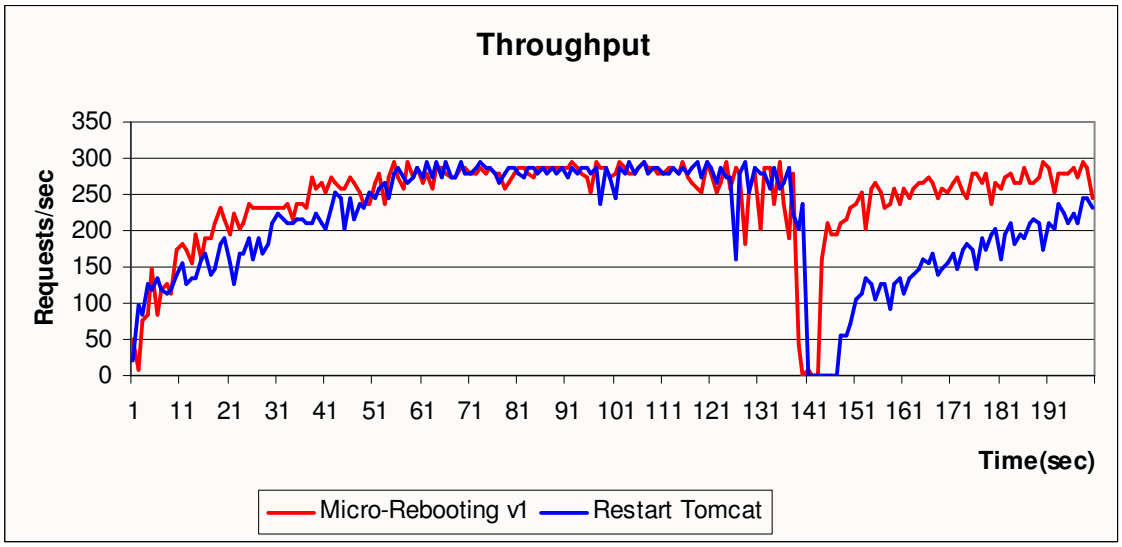

Figure 12: Comparing the throughput of a full Web-Server restart against Micro-Rebooting with a memory exhaustion fault in Tomcat/JPetStore

Similar results are obtained with the memory exhaustion fault, as shown in Figure 12. In this experiment we can also see that after a restart it takes longer to achieve maximum performance, than with a Micro-Reboot. With a Micro-Reboot warm-up period is around 10 seconds while after a restart the warm-up takes 1 minute.

4.5 What is the downtime of Micro-Reboots?

After proving the effectiveness of Micro-Reboots we wanted to evaluate the downtime associated with Micro-Reboots compared to a full Web-Server restart. These are very important results because we previously stated that

Micro-Reboots are a fast recovery mechanism that can be useful in achieving high-availability by reducing the Mean-Time-To-Repair.

For this experiment we apply the rejuvenation action at the time of 120 seconds. The rejuvenation can be a Micro-Reboot or a full Web-Server restart. We present the results in Figure 13. We adopted a similar format as published in [Candea2004b]. A solid bar indicates that none of the clients perceive the service as unavailable. A gap indicates that at least some requests during that time period failed.

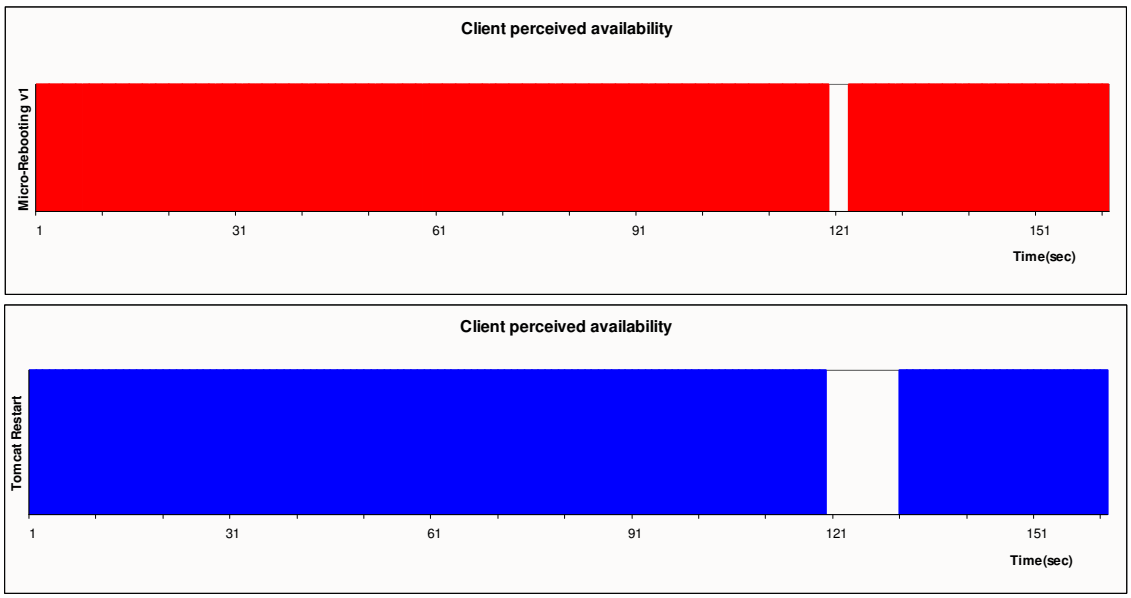

Figure 13: Client perceived availability for different rejuvenation mechanisms in in Tomcat/JPetStore

With a Micro-Reboot clients perceive an average downtime of 2.94 seconds. A full Web-Server restart causes a 10.78s visible downtime. Similarly to Micro-Rebooting in the JBoss application server [Candea2004b], we noticed that Micro-Rebooting in Apache Tomcat is a faster rejuvenation mechanism than restarting the Web-Server.

In JBoss a Micro-Reboot is about 30 times faster than a Web-Server restart, but in Apache Tomcat it is only 3.6 times faster. This is mainly due to the smaller complexity of an Apache Tomcat restart, which takes only around 10 seconds against the 30 seconds of JBoss. As JBoss components are more loosely-coupled the Micro-Reboots can be more fine-grained, which means JBoss Micro-Reboots of smaller components can be as fast as 0.5 seconds, where in Apache Tomcat our Micro-Reboots take about 3 seconds.

4.6 Does a stall proxy reduce missed requests?

We already noticed that during the Micro-Reboot we have some missed requests. To solve this we implemented a second version of the Micro-Reboot framework adding a stall proxy between the end-users and the Web-Server. We conducted a 10 minute experiment where at the time of 120 seconds we executed a Micro-Reboot with the v1 and v2 or our framework. To achieve average results we repeated this experiment 10 times. We focused of the number of missed requests measured during the Micro-Reboot operation. Table 4 presents the obtained results.

	Micro-Rebooting v1	**Micro-Rebooting v2**
Client perceived downtime (msec)	2687	2975
Total Missed Requests	209	188
Missed Requests during the Micro-Reboot	9	0

Table 4: Comparing Micro-Rebooting framework version 1 and version 2

We can see that the stall proxy does avoid the missed requests during the Micro-Reboot. But the clients perceive a higher downtime. This is due to the overhead of Micro-Rebooting version 2. In this version we have to check if Apache Tomcat is operational before allowing clients requests to resume, thus increasing the perceived downtime. We should note that the requests made during a Micro-Reboot take longer because they are stalled in the proxy. But

they still are completed below the 8 second threshold that was suggested in [Candea2004b], as the maximum acceptable time a web-user waits for a request.

The results show that the advantage of using the stall proxy is not very high. The majority of missed requests are still due to the latency of the fault detectors to report the fault.

4.7 When Micro-Reboot does not fix the detected fault

In our previous experiments Micro-Reboots were able to fix the detected fault and successfully retrieve the application to its maximum performance. But there are limitations to the use of Micro-Reboots in Apache Tomcat. We repeated the experiment with the memory exhaustion fault but removed the fault detector that monitors Tomcat memory usage. The memory fault injected was also more aggressive: 1MB per request. Without the monitor the fault detectors only warn the Micro-Reboot Manager when the Web-Server returns the *OutOfMemory* exception. Figure 14 presents the observed throughput of that experiment.

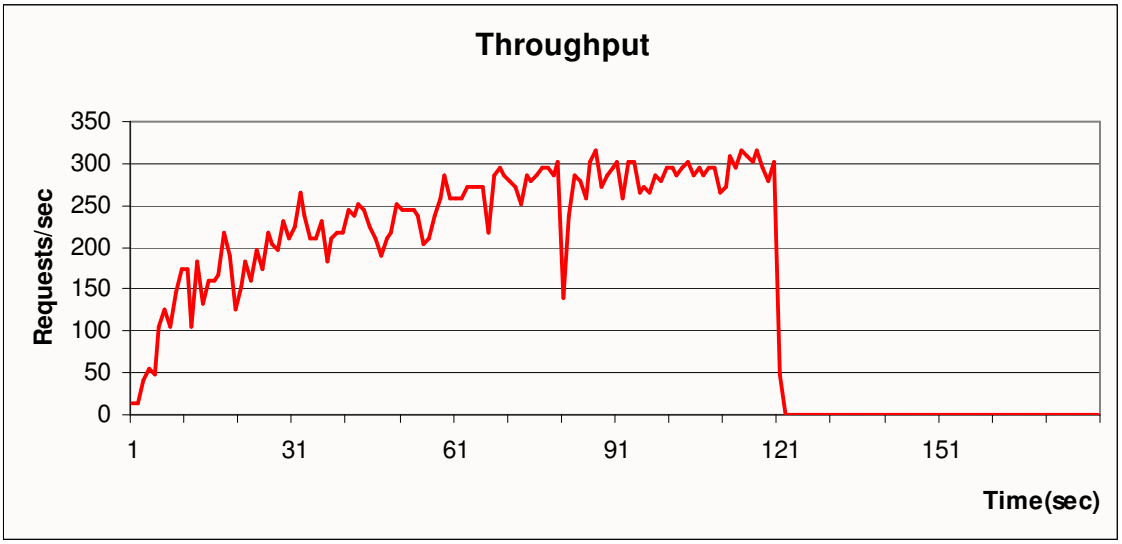

Figure 14: Throughput of JPetStore with a memory exhaustion fault injected at 120s

When the Web-Server returns *OutOfMemory* exceptions it means it has critically reached its maximum memory usage. When that happens Apache Tomcat is unable to successfully Micro-Reboot. It reaches a state only fixed by a full application restart.

To fix this problem we altered our framework to be able to apply a server restart when needed. When the framework receives a fault alert, it first tries to apply a Micro-Reboot. After a configurable amount of time, if the application is still not available we force a full Web-Server restart. In our case we used a 10 second threshold. Figure 15 presents the results of the same experiment with this new framework version. We can see that with the Web-Server restart we were able to fix the fault.

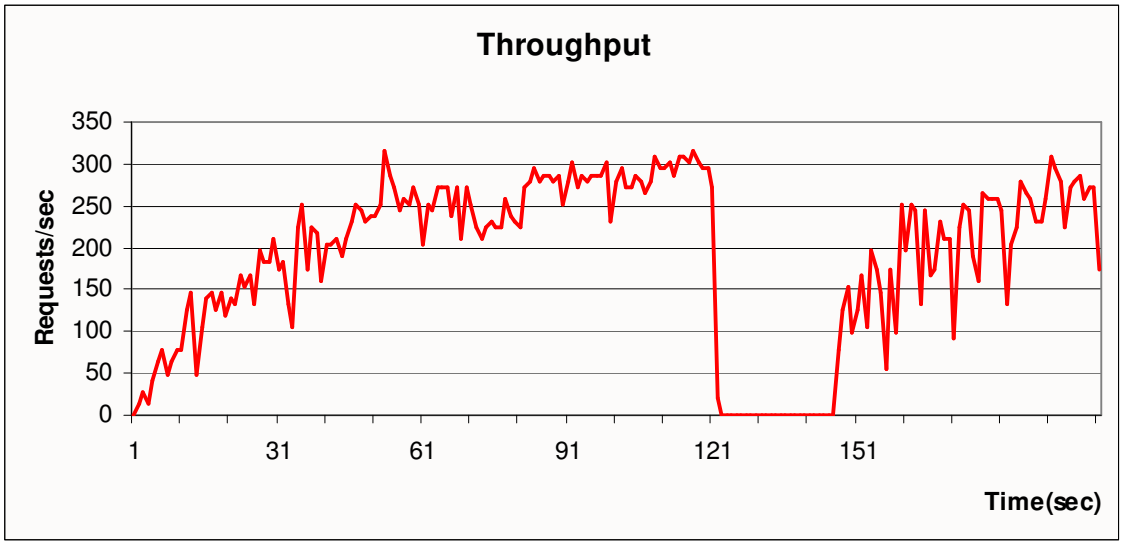

Figure 15: Throughput of JPetStore with a memory exhaustion fault injected at 120s

4.8 What happens to session data after a Micro-Reboot?

Internet systems have some characteristics [17] that make restart-based recovery relatively safe. Most requests are stateless and all the important, persistent state is kept safe outside the Web-Server, usually in a Database Server that makes sure that data is consistently stored. But some session data is still used and kept in Web-Servers. While session data is not critical to the application consistency it is very used to store temporary data. It is commonplace to use session data to keep users logged in a web application or to store shopping cart information, as JPetStore does.

When we apply a restart or a Micro-Reboot it is important to store session data to avoid service disruption to the end-users. Apache Tomcat has already a built-in mechanism, session persistence [26], that stores session data before a restart/Micro-Reboot is done. [18] suggests that web-applications must manage the session state externally, with a mechanism that guarantees its persistency, like [27].

For our experiments we used the session persistence mechanism of Apache Tomcat to guarantee that session data is reloaded when the Micro-Reboot ends. To analyse the overhead of this mechanism we ran a 3 minute experiment where we did a Micro-Reboot at the time of 120 seconds. We then ran the same experiment with the session persistent mechanism off. We show the results in Table 5.

	Session Persistence Off	**Session Persistence**
Total Number of Requests	41254	40929
Overhead (%)		0.78

Table 5: Comparing the overhead of session persistence (JPetStore)

The overhead is minimal. In the case of JPetStore it is under 1%. Without session persistence users have to re-login after a Micro-Reboot is issued, so we believe using session persistence is a fair trade to reduce service disruption.

4.9 Results Discussion

Our experimental study proofs that Micro-Rebooting can be applied to Apache Tomcat successfully, reducing the MTTR thus making a contribution to higher availability. Micro-Reboots can solve both transient faults and software aging. The measured overhead is minimal if we consider only the Micro-Reboot framework. In our study the bigger overhead was the fault detectors used. The use of a stall proxy is still an open issue to further discussion. While it has a minimal overhead and reduces the number of missed requests, some can argue that it increases the complexity of

our framework and can have scalability issues. If we really want to achieve zero missed requests in a Micro-Reboot, further work has to be done in the fault detection and diagnosis, a very interesting research topic.

5 Conclusions and Future Work

This experimental study of Micro-Rebooting in Apache Tomcat showed that Micro-Reboots are indeed a very effective software rejuvenation technique. A Micro-Reboot is 3.6 times faster than a full Web-Server restart so we can reach higher availability by reducing the MTTR. The overhead of our framework is minimal, but the fault detectors caused an 18.66% overhead in performance. Micro-Reboots also showed to be a great solution for shared hosting scenarios. In that cases, if an application needs to be restarted all applications suffer from downtime. With Micro-Reboot only the users of the faulty application suffer the service disruption. Micro-Rebooting in Apache Tomcat also showed some limitations. Tomcat can, sometimes, reach a state where it can not even Micro-Reboot, requiring a full Web-Server restart. In those cases our framework first tries to Micro-Reboot. When that does not work it applies a full restart.

We believe the software fault detection and prediction is still an important research area. The Micro-Reboot experimental study clearly showed that while Micro-Reboots cause only a few missed requests, the latency between the fault injection and fault detection causes a significant amount of missed requests. This certainly encourages the focus on more advanced fault detection and prediction tools. If we combine these software rejuvenation techniques with excellent prediction systems we can approach 100% availability. Micro-Reboots can also be used to better tolerate false positives as the rejuvenation procedure is faster.

Many system operators are already familiar to the Web-Server restart operation, needed to fix hard-to-detect software defects. We believe that the introduction of a Micro-Reboot framework in Apache Tomcat can successfully replace full restart operations, thus reducing the downtime without adding infrastructure costs or requiring re-engineering of the web applications. There are already tools [28-30] that monitor Apache Tomcat and apply automated restarts. The integration of our framework in those tools should be a very useful task to system operations.

References

[1] MemProfiler: http://memprofiler.com/
[2] Parasoft Insure++: http://www.parasoft.com
[3] J.Kephart, D.M.Chess. "*The Vision of Autonomic Computing*", IEEE Computer, 36(1), 2003
[4] A.Fox, D.Patterson. "When Does Fast Recovery Trump High Reliability?", Proc. 2nd Workshop on Evaluating and Architecting System Dependability, CA, 2002
[5] Oppenheimer, D., Archana Ganapathi, and David A. Patterson. "Why do Internet Services fail, and What can be done about it?" 4th USENIX Symposium on Internet Technologies and Systems (USITS'03), March.2003.
[6] Y.Huang, C.Kintala, N.Kolettis, N. Fulton. "*Software Rejuvenation: Analysis, Module and Applications*", Proceedings of Fault-Tolerant Computing Symposium, FTCS-25, June 1995
[7] A.Avritzer, E.Weyuker. "*Monitoring Smoothly Degrading Systems for Increased Dependability*", Empirical Software Eng. Journal, Vol 2, No 1, pp. 59-77, 1997
[8] Apache: http://httpd.apache.org/docs/
[9] Microsoft IIS: http://www.microsoft.com/
[10] M. Grottke, L. Li, K. Vaidyanathan, K. S. Trivedi "Analysis of Software Aging in a Web Server", IEEE Transactions on Reliability, Vol. 55, No. 3, pp. 411-420, 2006
[11] 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
[12] 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
[13] 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
[14] E.Marshall. "*Fatal Error: How Patriot Overlooked a Scud*", Science, p. 1347, Mar.1992
[15] G.Candea, A.Brown, A.Fox, D.Patterson. "*Recovery Oriented Computing: Building Multi-Tier Dependability*", IEEE Computer, Vol. 37, No. 11, Nov.2004

[16] JBoss: http://jboss.org
[17] 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
[18] G. Candea and A. Fox, “End-User Effects of Microreboots in Three-Tiered Internet Systems”, Stanford Technical Report, arXiv.org:cs.OS/0403007, March 2004
[19] 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
[20] Apache Tomcat: http://tomcat.apache.org/
[21] Apache Tomcat Manager: http://tomcat.apache.org/tomcat-5.5-doc/manager-howto.html
[22] Nuno Rodrigues, Décio Sousa, Luis Silva, Artur Andrzejak. “A Fault-Injector Tool to Evaluate Failure Detectors in Grid-Services”. *In Coregrid Workshop, Greece, June, 2007*
[23] iBatis Jpetstore, http://ibatis.apache.org/javadownloads.html
[24] Sun Microsystems, Java Pet StoreDemo, http://developer.java.sun.com/developer/releases/petstore/
[25] 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, 2006
[26] Apache Tomcat Session Persistence, http://tomcat.apache.org/tomcat-5.5-doc/config/manager.html
[27] B. Ling, E. Kiciman, and A. Fox. Session state, “Beyond soft state”, *NSDI*, 2004.
[28] Tomcat Watchdog, http://aujava.wordpress.com/2006/08/16/watchdog-for-tomcat/
[29] Lambda Probe, http://www.lambdaprobe.org
[30] Monit and Tomcat, http://www.nabble.com/monit-and-tomcat-t3908009.html