High-Available Grid Services through the use of Virtualized Clustering

Javier Alonso[#], Luis Silva[*], Artur Andrzejak[+], Paulo Silva[*] and Jordi Torres[#]

[#] *Computer Architecture Department, Technical University of Catalonia*
Barcelona Supercomputing Center
C/ Jordi Girona 1-3, Building C6, Campus Nord. 08034 Barcelona, Spain
alonso@ac.upc.edu torres@ac.upc.edu

[*] *Dep Eng Informática, University of Coimbra,*
3030-Coimbra, Portugal
luis@dei.uc.pt

[+] *Computer Science Research Department*
Zuse Institute Berlin (ZIB)
Takustr. 7, 14195 Berlin, Germany
andrzejak@zib.de

Abstract

Grid applications comprise several components and web-services that make them highly prone to the occurrence of transient software failures and aging problems. This type of failures often incur in undesired performance levels and unexpected partial crashes. In this paper we present a technique that offers high-availability for Grid services based on concepts like virtualization, clustering and software rejuvenation. To show the effectiveness of our approach, we have conducted some experiments with OGSA-DAI middleware. One of the implementations of OGSA-DAI makes use of Apache Axis V1.2.1, a SOAP implementation that suffers from severe memory leaks. Without changing any bit of the middleware layer we have been able to anticipate most of the problems caused by those leaks and to increase the overall availability of the OGSA-DAI Application Server. Although these results are tightly related with this middleware it should be noted that our technique is neutral and can be applied to any other Grid service that is supposed to be high-available.

I. INTRODUCTION

Grid computing deals with the usage of large scale and heterogeneous resources in geographically dispersed sites. The target applications which are executed over Grid environments usually require a huge execution time and computation capacity. If there is a failure in one or more Grid services used by the application, all the work that has been done can be simply lost.

Unfortunately these complex environments are highly prone to the occurrence of transient failures [1]. Some of these failures are easy to fix by the system administrator provided there is a deterministic pattern and it is easy to find the cause of the error.

However, there are a large number of failures that are quite difficult to spot due to the indeterminist nature and difficulty to re-create them during the final acceptance tests of the software. One particular case is the occurrence of software aging. This problem has been reported in telecommunication systems [2], web-servers [3], enterprise clusters [4], OLTP systems [5] spacecrafts systems [6] and even military systems [7].

Since it is impossible to guarantee that the mean time between failures (MTBF) will be bigger than the total execution time of a single application, it is needed to provide some mechanisms to guarantee the availability of the application despite the occurrence of transient failures, more than the support for fault-tolerance.

In the last twenty-five years we have seen a considerable amount of research in the topic of fault-tolerance and high-availability. Techniques like server redundancy, server fail-over, watchdogs, replication schemes, checkpoint and rollback have been extensively studied in the literature. However, most of those techniques only act when a failure has happened. If the system anomaly does not result in a crash but rather in a decrease in the performance levels or some instability in the quality of service (QoS) metrics most of those techniques are not even triggered.

We felt it would be important to extend the failure model and to consider the occurrence of fail-stutter failures [8]. In this paper we will present a software solution to achieve high-available and high-responsive application servers that provide Grid services. Our technique makes use of several concepts: a virtualization layer that is installed in every application server that is running a Grid service; the use of primary-backup scheme for server replication; the implementation of system surveillance scheme to detect anomalies in a timely-reactive manner; the adoption of software rejuvenation and a clean migration mechanism to transfer the execution between servers without losing on-going requests. All of these concepts and techniques have been merged in one package and provide a cost-effective solution for high-availability, without requiring additional servers and load-balancing machines, which would increase the budget of the infrastructure. Our technique has been experimentally evaluated using a well-known package of Grid middleware: OGSA-DAI. The results are quite promising.

The rest of the paper is organized as follows: Section 2 presents the goals of our technique for high-availability. Section 3 describes the virtualized clustering architecture. Section 4 presents the experimental setup that has been used in our experiments. Section 5 presents a study about the overhead impact of using a virtualization layer in our solution. Section 6 presents some results which show the effectiveness of our solution. Section 7 evaluates the downtime that is achieved using virtualized clustering. Section 8 concludes the paper.

II. How to achieve our goals

The first step in our approach is to achieve a very effective anomaly detection mechanism. This is achieved by using external surveillance of QoS metrics and internal monitoring of system parameters. The surveillance of external QoS metrics has been proved relevant in the analysis of web-servers and services [9] and it has been highly effective in the detection of software aging and fail-stutter failures [10]. This is tightly combined with the monitoring of system internal metrics. All these monitoring probes should be timely-reactive and able to detect potential anomalies, before the occurrence of a system failure. Even if this detection would lead to some false-alarms it would be important to avoid the occurrence of unplanned crashes.

When a potential anomaly is detected we should apply an automatic recovery action. In most of the cases we apply a rejuvenation action. To avoid losing on-going requests when there is a planned restart or rejuvenation we make use a Grid-service backup that has been running in the same server machine of the primary Grid service. To allow the execution of two replicas of the Grid service in the same machine we will make use of a virtualization layer and we will launch a minimum of three virtual machines per server machine. This way we are not increasing the cost of the infrastructure since we are exploiting the concept of virtualization and server consolidation [11-12].

Since in our technique we inherit some of the concepts of server clustering together with virtualization we decided to name this approach by “Virtualized Clustering”.

The use of Virtualization in every server that is running a Grid Service may be seen as a source of performance degradation since we are introducing a new abstraction layer. However, the results obtained with our proposal are promising and clearly show that the use of virtualization is clearly positive.

III. Virtualized Clustering Architecture

In this section we describe our proposal to offer high availability grid services. As explained, our software solution for high-availability assumes the adoption of a virtualization layer in the machine that provides the Grid service. In our prototype implementation we made use of XEN [13], although it is also possible to use other virtualization middleware like VMWare [14] and Virtuoso [15], among others.

Our solution requires the creation of three virtual machines: one virtual machine with our own load balancer module (VM-LB machine); a second virtual machine where the main grid service runs; and a third virtual machine where we have a replica of the grid service which works as a hot-standby grid service. Figure 1 represents the conceptual architecture of our approach.

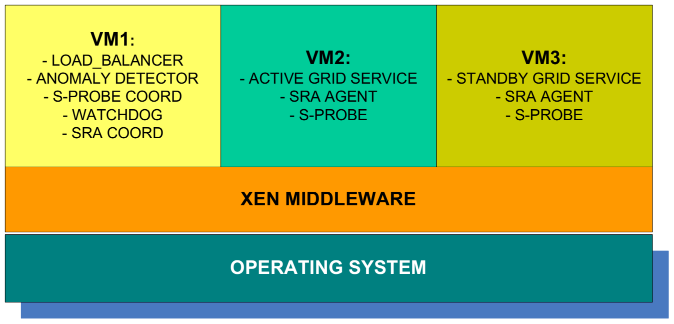

Figure 1: Virtualized Clustering architecture description.

VM1 machine runs a set of components: our Load Balancer (LB); a module that collects system data from the grid service; a module that detects potential anomalies; a software watchdog; and a last module that is the coordinator of the self-recovery actions. In our prototype we have used some open source modules modified to achieve our targets. The LB module was implemented using *Linux Virtual Server* (LVS) [16]. LVS is a 4-Layer load-balancer that provides IP fail-over capabilities and a myriad of balancing policies.

The *S-probe coordinator* collects data from the Grid service and the underlying operating system. It requires the installation of some probes that collect some system metrics like CPU, memory usage, swap usage, disk usage, and number of threads, among other system parameters. This module provides data feed to the last module: *Anomaly-Detector*.

The *Anomaly-Detector* is a core module in our system: for the time being, we are using some simple threshold techniques based on the observed conditions and system metrics of the target grid service. In the future, we plan to enhance this module with statistical learning techniques and time-series analysis to provide more accurate forecasts of the occurrence of potential anomalies.

When the anomalies are not detected in time and the Grid service ends up to crash this is directly detected by the *Watchdog* module. For this module we used a modification of *ldirectord* tool [17]. This tool modifies LVS tables directly and it executes a test over a static HTML page to check if the server is alive. We have enhanced this tool with some improvements in the time-out policies to avoid the occurrence of false-alarms

In the other two virtual machines, we installed the Grid service. In every VM we install a *SRA Agent.* This agent has the responsibility for the recovery action. This module is coordinated by the *SRA-Coordinator*. Another module that is installed in every VM is the *S-probe*. This probe collects system metrics and external QoS metrics like throughput and latency from the grid service and sends data to the *S-probe coordinator* which is running in the VM1 machine.

The *SRA-Coordinator* works in coordination with the two *SRA-Agents* installed in grid services machines. When this module decides to apply a planned restart to clear up the potential anomaly of the system we should apply a careful migration mechanism between the primary and the backup Grid service. This procedure is implemented by the up-front Load-Balancer that assures a transparent migration for the client applications that are assessing that Grid Service. This operation should also preserve data consistency and assure that no in-flight request should be lost during the migration phase. For this reason we have a window of execution where we have both servers running in active mode: the hot-standby service is made active and will receive the new requests while the old grid service should finalize the requests that were still on-going. This way we do not lose any in-flight request and we assure the data consistency.

Our software solution has been implemented in Linux, and in two of the modules we have used some open-source tools, like *LVS* or *Ldirectord.* The deployment of our framework does not require any change to the grid services or the middleware containers. They are also neutral to the virtualization layer. As soon as these software modules are made robust for third-users they will be made available for the scientific community.

IV. EXPERIMENTAL SETUP

To evaluate our Virtualized Clustering framework we elected as benchmark Grid service the OGSA-DAI middleware [18].

OGSA-DAI is a package that allows remote access to data-resources (files, relational and XML databases) through a standard front-end based on Web services. The software includes a collection of components for querying, transforming and delivering data in different ways, and a simple toolkit for developing client applications. OGSA-DAI provides a way for users to Grid-enable their data resources.

The front-end of OGSA-DAI is a set of Web-services that in the case of WSI requires a SOAP container to handle the incoming requests and translate them to the internal OGSA-DAI engine. This SOAP container is Tomcat/Axis 1.2.1 [19].

The detailed description of the OGSA-DAI internal is out-of-scope of this paper. At the moment OGSA-DAI middleware is used in several important Grid projects [20], including: AstroGrid, BIoDA, Biogrid, BioSim-Grid, Bridges, caGrid, COBrA-CT, Data Mining Grid, D-Grid, eDiaMoND, ePCRN, ESSE, FirstDIG, GEDDM, GeneGrid, GEODE, GEON, GridMiner, InteliGrid, INWA, ISPIDER, IU-RGRbench, LEAD, MCS, myGrid, N2Grid,OntoGrid, ODD-Genes, OGSA-WebDB, Provenance, SIMDAT, Secure Data Grid, SPIDR, UNIDART and VOTES.

This list is clear representative of the importance of OGSA-DAI and its relevance as a Grid Service.

Although OGSA-DAI is being used by several Grid projects and a considerable community it includes an internal point of concern, from the point of view of availability. OGSA-DAI WSI makes use of Apache Axis 1.2.1, as the SOAP router. This implementation of Axis is highly prone to internal memory leaks, as has been reported in [21]. When the service developers of OGSA-DAI make use of session-scope in their Grid services those memory leaks result in a clear scenario of software aging, with performance degradation and even system crashes.

For this reason it is quite reasonable testing our framework with OGSA-DAI WSI, when the services are configured with session scope. Without our framework the OGSA-DAI server would crash or present request failures in that described scenario and after some time usage. This would undermine completely the availability of the Grid Application that would make use of an OGSA-DAI service. To increase the availability of this service we have installed our framework in an OGSA-DAI server and we conducted an experimental evaluation with some synthetic workload.

To simulate a set of clients we have used the QUAKE tool [22]. This is a tool of dependability benchmarking that was implemented in our Labs and it facilitates the launching of simultaneous multiple clients that execute requests in a server under test with a programmed workload.

In our experiments we used a cluster of 5 machines: 3 running the client benchmark application, one Database server (Tania) running two virtual machines and our main server (Katrina) running three virtual machines. All machines are interconnected with a 100Mbps Ethernet switch. The detailed description of the machines is presented in Table 1.

TABLE I
DETAILED EXPERIMENTAL SETUP DESCRIPTION

	Katrina	Tania	Clients machines
CPU	Dual AMD64 Opteron (2000MHz)	Dual Core AMD64 Opteron 165 (1800MHz)	Intel Celeron (1000MHz)
Memory	4GB	2GB	512MB
Hard Disk	160GB(SATA2)	160G(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-b05 Standard Edition
Tomcat JVM heap size	1024MB		
Other software	Apache Tomcat 5.0.28, Axis 1.2.1 OGSA-DAI WSI 2.2	MySQL 5.0.18	

V. VIRTUALIZED CLUSTERING OVERHEAD

As we described before, our solution runs over a virtualization layer. This new added abstraction layer over the stack increases the system overhead. For this reason, our first experiment was focused on calculating the overhead introduced by our solution. After that, we developed an accurate mathematical study to measure the advantages *versus* the disadvantages of using virtualization.

A. Virtualized Clustering Overhead

Our first experiment was to measure the performance penalty due to use a virtualization layer (XEN) and our virtualized clustering technique.

We executed several short time runs (10 min) on burst mode and we collected the results of three configurations: OGSA-DAI on top of the Operating System, OGSA-DAI on top of one virtual machine, and a last configuration where we executed OGSA-DAI with all the components of the virtualized clustering framework to measure the overhead of our approach.

Figure 2 shows the throughput of the three configurations in those test-runs of 10 minutes. It can be seen that there is some overhead for using XEN and our virtualized clustering framework.

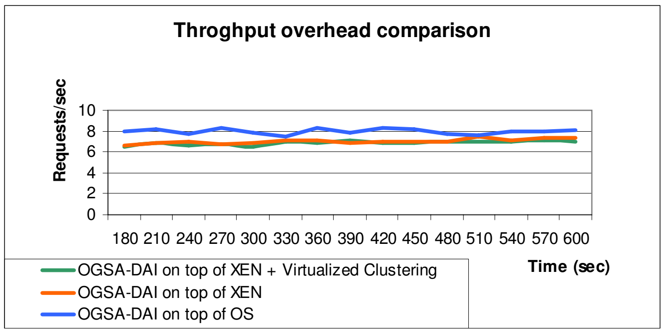

Figure 2: Throughput Comparison.

Table 2 gives more detailed numbers obtained in these experiments: average latency, average throughput, total number of requests and throughput overhead.

TABLE II
OVERHEAD COMPARISON RESULTS

	Average latency (ms)	Average Throughput (req/sec)	Total Number Requests	Throughput Overhead
On top of OS	299,16	7,94333333	2383	
On top of Xen	358,89	7,13666667	2141	10,2%
Virtualized Clustering	363,00	6,96666667	2090	12,3%

By comparing the total number of requests we can see that the virtualization overhead introduces a 10% of overhead and our solution adds a 2% additional overhead. These results have been achieved with a burst workload, so they should be seen by the reader as the maximum observable overhead.

One could argument that the virtualization middleware introduces a visible overhead in the performance Grid service. Due to the introduction of an overhead we decided to make a study to calculate the advantages of virtualization.

B. Average Performance under Virtualization Overhead

The usage of a virtualization layer obviously decreases application performance, while on the other hand it reduces the Mean Time to Recovery (MTTR) in case of a fault or rejuvenation need. This gives rise to a question what is more beneficial for the long-term performance: given a large time interval with many rejuvenation cycles, will an application A_{phys} running on a physical server serve more requests than an identical application A_{virt} running in a virtual machine, or the other way around?

To treat this problem we introduce the following definitions. Assume that under the full load (burst request distribution) A_{phys} can serve at most P_p requests per second which we call its *instantaneous performance* (and could be called the maximum throughput), and let P_p depend on the time since last rejuvenation. Under the same conditions let $P_v=P_v(t)$ be the number of requests per second served by A_{virt}. The *average performance* $\underline{P}_p$ is defined and computed by summing up the number of served requests over a large time interval and dividing it by the length of this interval (analogously $\underline{P}_v$). The Mean Time to Failure (MTTF) t gives us the expected time until an application must be rejuvenated, and we assume that it is equal for both A_{phys} and A_{virt}. However, each application has a different MTTR which we designate as r_p and r_v, respectively. We introduce the ratio $q= P_v/P_p$ called the *performance ratio*, which is less than 1 due to virtualization overhead. Even if the instantaneous performances vary over time, we might assume that q is constant since both servers age similarly.

First, to compute the average performance under a burst distribution with a fixed request rate above $\max(P_p,P_v)$ it is sufficient to consider just one failure and recovery cycle while assuming that MTTF t is not an expectation but a fixed time. We obtain then:

$$\underline{P}_p = \frac{\int_0^t P_p(t)dt}{t+r_p},$$

$$\underline{P}_v = \frac{q\int_0^t P_p(t)dt}{t+r_v}.$$

The major question to be answered is: what is the minimum level q_{min} of the performance penalty under which the average performance of the virtualised application A_{virt} is still matching the average performance of A_{phys}?

By setting $\underline{P}_p = \underline{P}_v$ and solving by q we find out that under the above conditions

$$q_{min} = \frac{t+r_v}{t+r_p}.$$

These solutions look different if the applications are not running under their respective full load. For example, if the request rate never exceeds P_v, then there is no difference in the throughput between applications. In the latter case q_{min} might take any value above 0 and the ratio of the average performances depends only on the ratio of the MTTR's r_v and r_p. Furthermore, if the requests arrive according to some more complex distribution, e.g. the Poisson distribution, then only time intervals with arrival rates above the momentary levels $P_v(t)$ and $P_p(t)$ contribute to the differences between average performances.

To compute q_{min} in the experimental setting of the OGSA-DAI server we have compared the throughput of the non-virtualized case ("On top of OS") vs. the virtualized case without the rejuvenation service ("On top of XEN") and with this service ("Virtualized Clustering"). To safeguard against the effects of the initialization phase and the aging effects we compared the service rates after the initial 5 minutes since start for the duration of 5 minutes. Table 3 shows that the performance ratio q was 2141/2383 = 0.8984 for the case "On top of XEN", and 2090/2383 = 0.8770 for the rejuvenation control scenario.

TABLE III
MEASURED VIRTUALIZATION OVERHEAD IN THE TIME INTERVAL 5-10 MIN

	Number requests	**Average throughput**	**overhead**	**performance ratio q**
On top of OS	2383	7.943	-	-
On top of XEN	2141	7.137	10.2%	0.8984
Virtualized Clustering	2090	6.967	12.3%	0.8771

TABLE IV
MINIMAL REQUIRED PERFORMANCE QMIN DEPENDING ON THE MTTF T FOR R =0 AND R=12.5 SECONDS

MTTF t	75	100	125	150	500	2000	3000
q_{min}	0.857	0.889	0.909	0.923	0.976	0.994	0.996

To confront this value with the computed minimal required performance q_{min} to match the average performance in both modes, we use as values of the MTTR parameters $r_v = 0$ and $r_p = 12.5$sec. Under the assumption of a variable MTTF t we obtain Table 4 with the values for q_{min}. It shows that only for very short rejuvenation cycles (below 125 sec) the average performance in the virtualized case does not fall behind the case of A_{phys}. Since the aging effects are noticeable after much longer time (above 30 minutes), the average performance of A_{virt} is in general worse than the one of A_{phys}. However, for the cost of about 12.3% average performance -with the worst case only in the burst mode - we completely eliminate outages caused by software aging, which is a fair trade.

VI. HOW EFFECTIVE IS OUR APPROACH?

After the reasoning about the virtualization overhead, we have to demonstrate the effectiveness of our approach. The main goal of this experiment was to demonstrate that our solution can apply an automatic rejuvenation action without losing any client request.

We started by conducting several experiments to observe the behaviour of OGSA-DAI and the underlying system metrics. We concluded that external QoS metrics were very unstable and it would be quite difficult to apply some threshold analysis to trigger automatic recovery action. After that, we studied in detail the underlying system metrics and we observed that memory usage was increasing with time until the maximum memory usage allowed by the configuration. When this happen the OGSA-DAI server had an unpredictable behaviour with very unstable performance and sometimes it resulted in system crashes.

Since we knew the cause of the anomaly was an internal memory leak on Axis layer, we decided to apply a threshold trigger associated with memory usage. Thereby, we have configured our *S-Probe* to obtain the memory usage every 5 seconds and when it would get higher than a certain threshold we would apply a rejuvenation action in the main Grid service to avoid a possible failure or crash.

We ran two configurations: OGSA-DAI without our solution to obtain reference curves; and then OGSA-DAI with our solution. In this latter case, we have configured the threshold to trigger when memory usage gets to a certain limit: 50% of maximum memory. Both experiments were executed with a workload on burst mode.

Figures 3 and 4 show the throughput and the latency results, respectively. We can observe that our solution was able to achieve a higher throughput and a lower latency.

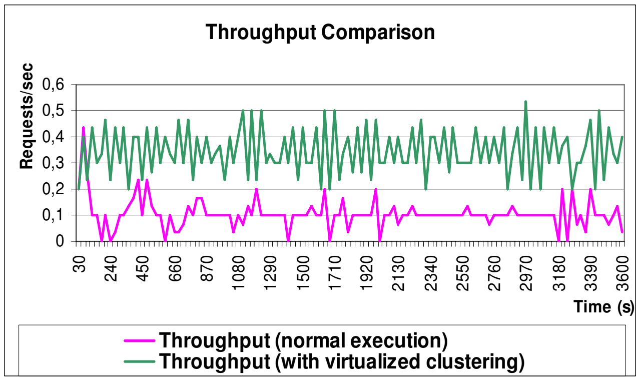

Figure 3: Throughput comparison with and without our solution.

In Figure 4 it is clear that the usage of our virtualized clustering approach was able to provide a very stable latency, while in the normal case the OGSA-DAI server (with session-scope) presented a very unstable latency and in some cases we even registered some missed requests.

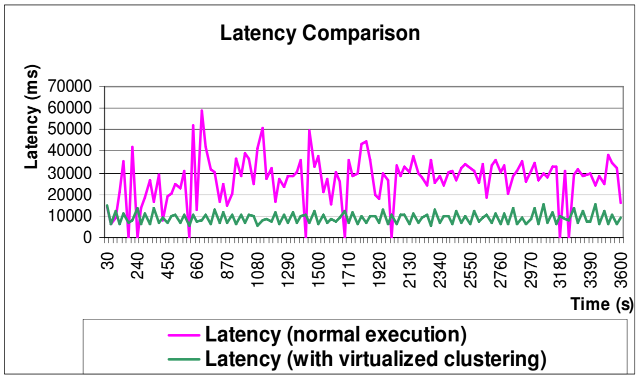

Figure 4: Latency comparison with and without our solution.

The reader could ask himself, what is the reason for this difference between throughput results? Because you can suppose that in both experiments the throughput will be more or less the same, because the hardware is the same. However, this is not enough to achieve the same throughput. Because a normal execution of OGSA-DAI consumes the memory quickly and after that the throughput goes down like we can observe around second 30 in figure 3. Our solution avoids that fact, improving the results of the OGSA-DAI behaviour.

Table 5 presents more detailed numbers of this experiment. We can see three main things:

- Our approach was able to avoid any missed request. Without our framework we have observed 25 missed requests during that timeframe of the experiment;
- Our approach was able to reduce the average latency by a factor of 3;
- Our approach was able to improve the average throughput by a factor of 3.

Even if we had to pay the overhead of virtualization we were able to improve the performance of OGSA-DAI server by a considerable factor, just by applying some planned restarts to avoid performance failures. With this data we are able to argument about the cost-effectiveness of our solution for high-available Grid services.

TABLE V
DETAILED VALUES FROM 1 HOUR OF EXECUTION

	Avg. Latency	**Avg. Throughput**	**Avg. Memory Usage**	**Missed Requests**
OGSA-DAI (normal execution)	29013,2 ms	0,105 req/s	1100784,4 Bytes	25
OGSA-DAI (with virtualized clustering)	9049,5 ms	0,345 req/s	313384 Bytes	0

VII. DOWNTIME ACHIEVED WITH THE VIRTUALIZED CLUSTERING SOLUTION

After showing effectiveness of our solution to obtain a high available and highly-stable grid service, we wanted to evaluate the potential downtime for the grid service when a recovery action was executed as well as the number of failed requests during this process.

This experiment was very important to achieve one of our main goals. As mentioned we wanted to develop a solution that when a recovery action is done any work-in progress would not be lost to guarantee that client application would not see any request failure and would notice the OGSA-DAI server as always available.

We run four different experiments to compare our technique with other simpler restart solutions:

- One runs where we applied a recovery action at the time of 300 seconds.
- Another run where we triggered an OGSA-DAI restart at exactly that time.
- A third runs, where we applied a restart in the XEN Virtual machine where the main server was running, at the same time.
- And finally a last run, where we executed a full machine reboot at that time.

The results are presented in Figure 5, which presents an execution window of 600 seconds. We adopted that graph format to show the client perception about the availability of the grid service when a "restart" is done.

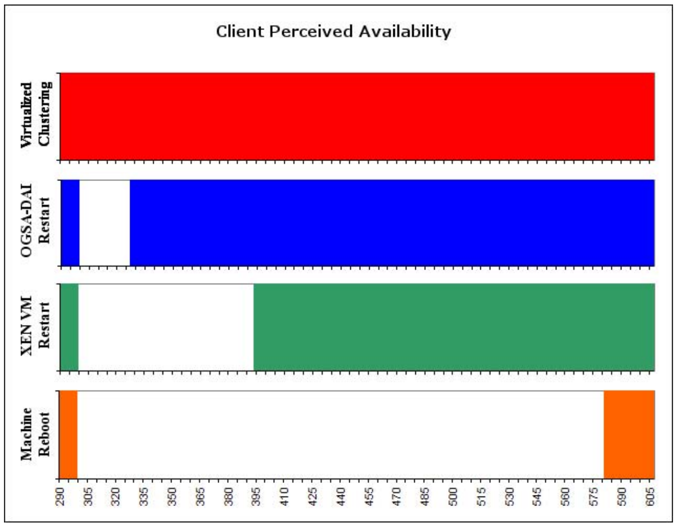

Figure 5: Client perceived availability for different restart mechanisms.

We can observe that when our solution is triggered there is no perceived downtime: from the point of view of the clients the Grid service is always available. This is achieved due to the execution window where both grid services (primary and hot-standby) are running at simultaneous time.

The restart of the OGSA-DAI server had a visible downtime of 25 seconds. A restart of the XEN Virtual Machine resulted in a downtime of 92 seconds. Finally, a restart of the full operating system incurred in a downtime of 279 seconds.

The next step was to measure the impact of every "restart" process according to the number or requests lost. The results are presented in Table 6. We included the average throughput obtained in every experiment during a test-run of 10 minutes. Assuming that there is one unique restart action in those 10 minutes we present the total requests during the execution, the number of failed requests, the average throughput in that test-run of 10 minutes and the observed downtime.

TABLE VI
COMPARING DOWNTIME AND FAILED REQUESTS WITH DIFFERENT RESTART MECHANISMS.

	Virtualized Clustering	**OGSA-DAI Restart**	**Xen VM Reboot**	**Machine Reboot**
Average Throughput (req/sec)	1.205	1.165	1.085	0.647
Total Requests	723	699	651	388
Failed Requests	0	70	758	2353
Downtime (msec)	0	25479	92514	279454

When applying our mechanism we were able to achieve the minimum MTTR as possible: zero. The Grid service was always available and no downtime was perceived by the client applications. We were also able to register zero failed requests. Other restart solutions incurred in several failed requests (from 70 up to 2353) and a visible downtime.

.

This experiment has shown that our virtualized clustering approach is a quite promising solution to achieve a high available grid service, with zero downtime and zero failed requests and without incurring any additional cost in terms of servers and IT infrastructure.

VIII. CONCLUSIONS

In this paper, we have presented a software solution for is highly effective for software aging, fail-stutter and performance failures. Our solution makes use of a virtualization layer that although it introduces some overhead it pays-off in terms of consolidation and ease of deployment. We have proved the effectiveness of our solution in the case of OGSA-DAI, but are convinced that similar results can also be observed in other Grid Services that present some latent failures.

Another important achievement is the fact that our solution is completely independent the server under test. We can use our framework with legacy software without requiring any change.

The results presented in this paper encouraged us to consider that this approach for high-availability can be a partial contribution for the deployment of dependable grid services and we are currently enhancing our solution with support for the real-time forecast of failures and critical anomalies.

ACKNOWLEDGMENT

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

References

[1] 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.

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

[3] 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)

[4] 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.

[5] 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.

[6] 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.

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

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

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

[10] 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.

[11] Renato J. Figuereido, Peter A. Dinda, José A. B. Fortes. "*A Case For Grid Computing on Virtual Machines*". Proc. of the 23rd Int. Conf. on Distributed Computing Systems, p. 550, May 19-22, 2003.

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

[13] (2007) Xen Source website [Online]. http://www.xensource.com/

[14] (2007) VMWare website. [Online]. http://www.vmware.com/

[15] (2007) Virtuoso website. [Online]. http://www.virtuoso.com/

[16] (2007) LVS website. [Online]. http://www.linuxvirtualserver.org/

[17] (2007) Ldirectord website. [Online]. http://www.vergenet.net/linux/ldirectord/

[18] (2007) OGSA-DAI website. [Online]. http://www.ogsadai.org.uk/

[19] (2007) Apache Axis. [Online]. http://ws.apache.org/axis

[20] (2007) OGSA-DAI Projects, [Online]. http://www.ogsadai.org.uk/about/projects.php

[21] W. Hoarau, S. Tixeuil, N. Rodrigues, D. Sousa and L. Silva. "*Benchmarking the OGSA-DAI Middleware*". CoreGrid Technical Report, No. TR-0060. October 5, 2006. http://www.coregrid.net

[22] 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