An index based load balancer to enhance transactional application servers' QoS

J.A. Alvarez-Bermejo, J. Roca-Piera
Dpt. Arquitectura de Computadores y Electrónica
Campus La Cañada de San Urbano
Universidad de Almería.
Almería. España

Abstract— **The Web is the preferred interface for the new generation of applications. Web services are an ubiquitous concept for both, developers and managers. These Web applications require distribution systems of web requests that allow and support the dynamism of these environments, to provide service availability and resource usage, commonly heterogeneous. Web services provide an entry point to the Web application business logic. Therefore, the design of appropriate load balancing strategies, taking into account the dynamic nature of the application servers' activity, is essential. In this work we present a load balancing policy and its integration in-between static and dynamic layers of any web application that uses application servers. The strategy gets the status report of each application server, used to later distribute web requests. Results that show how the strategy succeed are presented.**

Keywords- QoS; web applications; load balancing; application servers.

I. INTRODUCTION

A number of services are now on the Web, and the number raises in a continuing trend. The Web is not anymore the unique reason for an application to be used. Many other things should come into consideration, such as the user experience. The Web infrastructure is turning into a determining factor, therefore, corporations try to design high availability and fault tolerant services [1] making it necessary to provide the applications with strategies to offer the fastest path to reach the service provided [2], this has a two-fold consequence: clients get what they need when they need it (user experience), corporations' popularity climb the ranks due to their efficiency. The need of efficient web load balancing strategies is a direct drawback. These load balancing strategies must deal with a heterogeneous environment, where the servers and their workloads do not follow any strict patterns and are submitted to an unpredictable flow of requests.

High availability services [2], [3] reduce the response time when servers are overloaded or suffering from networking problems. These services should therefore make use of load balancing strategies to derive requests to alternative servers or networks. When referring to transactions that a web service might serve, then it would be ideal to have replicated application servers in order to make an efficient distribution of the workload. As this is unaffordable for a number of applications' owners, then policies to optimize the usage of existing servers (or event virtual servers) is advisable. High availability is even harder to ensure when the network that provides the services is the Internet. Thus when defining high availability and high throughput in a Web system where the Internet is an important component, there is no other option than focusing on the server's side availability and throughput.

The core side of a web application (built on the service oriented architecture) is the web service. The web service is related to some kind of computation that, usually, is wrapped into a transaction. There exist many solutions to improve the Quality of Service (QoS) of the web services, such as the one described in [23]. Our approach is not on the sequence of web services but on providing them the best of the executing scenarios available with the minimum hardware duplication as possible, making it a portable solution for a wide variety of applications. QoS must be understood as an issue where Load Balancing plays an important role. The most popular Load Balancing strategy is based on the number of requests submitted to each server. In contrast we show how a load balancing strategy integrated into the application server, considering factors such as the number of busy threads, or the fact that this load balancer is transparent to the business logic, improves different performance parameters.

This work shows how transaction's response times improves when monitoring the application server's health and reacting accordingly.

A. HPC Transactional Web Applications.

The *RojaDirecta* saga should make clear why the software industry -and the content industry- is looking for new enforcement tools [24]. When the Internet was not so popular, sharing licenses was hard and the methods employed to protect software were merely based in built-in passwords. As the Internet began to be used as a medium to distribute licensed software (p2p networks are the proof), the licensing protection mechanism began to be weaker. As the Internet acquired more relevance, protecting licenses and software became harder. Software is illegally cracked and then redistributed; in a high percentage *malware* is included in the distributed packages. As a solution, many authentication services were moved to Internet servers -and implemented as web services-, using licensed software means that it might be necessary establishing a first connection to a server in order to authenticate the software and let it run.
Implementing a license protection mechanism is a hot spot in software factories (and when done it is a high computational demanding task) like the antivirus industry. As an example, an antivirus need a license to retrieve (as fast as possible)

updated virus signatures databases in order to keep the system protected. Antivirus companies invest so much effort in building such databases. Duplicated license harness this useful effort. Checking the concurrent usage of licenses that would trigger the alarm system, is a costly operation and the requests should check the application servers' health to schedule the transaction in the best possible scenario. An example, see fig. 0, of a Web Service composed by several transactions operating concurrently with the streaming of multimedia content to preserve user experience.

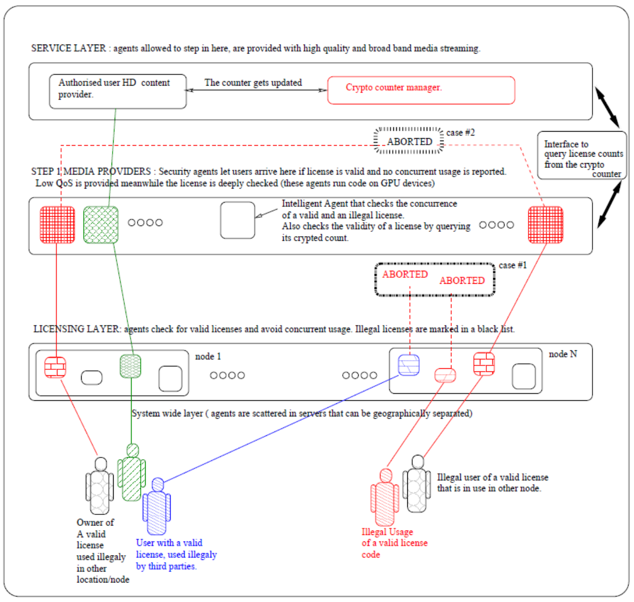

SERVICE LAYER : agents allowed to step in here, are provided with high quality and broad band media streaming.
Authorised user HD content provider.
The counter gets updated
Crypto counter manager.
ABORTED
case #2
Interface to query license counts from the crypto counter
STEP 1 MEDIA PROVIDERS : Security agents let users arrive here if license is valid and no concurrent usage is reported. Low QoS is provided meanwhile the license is deeply checked (these agents run code on GPU devices)
Intelligent Agent that checks the concurrence of a valid and an illegal license. Also checks the validity of a license by querying its crypted count.
ABORTED
ABORTED
case #1
LICENSING LAYER: agents check for valid licenses and avoid concurrent usage. Illegal licenses are marked in a black list.
node 1
node N
System wide layer (agents are scattered in servers that can be geographically separated)
Owner of A valid license used illegaly in other location/node
User with a valid license, used illegaly by third parties.
Illegal Usage of a valid license code
Illegal user of a valid license that is in use in other node.

Figure 0. Transactional Licensing Watchdog WebService

Figure 0 shows an implementation of a licensing protection mechanism, used to provide multimedia content through the Internet, when a user logs, the transaction is started and the protocol is ignited to avoid illegal usage of licenses. Obviously as the fast the protocol executes, the sooner we can provide multimedia content to the legally validated user. In the meanwhile, for the sake of the user experience, content should be provided with low quality. To avoid the streaming of too much content to an illegal validated user, the protocol should run in the best and healthy application server. A more detailed description on the cryptographic protocol used to detect illegal licenses can be found in [25]. We propose in this paper a method and a feasible architecture to provide throughput when accessing services through the Internet that may need using licenses that cannot be replicated, duplicated, shared, or even cracked. This paper is structured as follows; section 2 is devoted to giving a description of the current transactions logic infrastructure. Section 3 underlines the needs raised within the former infrastructure description; a solution proposal is therefore sketched. Section 4 gathers results from the solution proposed and are evaluated against several metrics considered of interest to what we are measuring and conclusions are exposed at the end of the section.

II. A Three Layer Architecture for Service-Oriented Web Applications

The most commonly used architecture to build a web service-oriented application is that in which three layers are used in a way that the Web server is the first layer, the middle layer is that where the dynamic engines (application servers) are present (i.e. JSP, ASP, PHP,..) this layer is commonly called the business layer. The back-end layer is, obviously, the data storage layer.

Layers are integrated to form a whole, each layer serves to each other as a compliment to fulfil the web request: the client sends a request to the Web server (Apache web server in our case). The Web server formats this request using the second layer's dynamic logic, which in turn may have used access to data by interacting with the third layer. So for the sake of simplicity, stating the difference among web server and application server (see reference [4]) is as simple as indicating that web servers only act as a mere intermediary among client and the information that is being requested. The Application server is the logic used by the web server to build a response for each request from the client side, the second layer is its advisable homeland. One of the most popular and widely used technologies for this second layer (application servers) is Java Enterprise Edition, aka J2EE [4], [5]. Two important components of a J2EE application server are:

- Servlet Engine: for providing an architecture independent environment to let the dynamic execution entities (java server pages) prepare, collect, results that are to be sent to the Web Server.
- Entreprise Bean Engine: sandbox where business logic procedures are executed in the context of a JSP.

Apache and JBoss [5] were selected as the target platform to test our load balancing proposal due to their open source nature and their wide usage in the corporation environment. Fig. 2 shows how these two components can be accommodated to fit the three layer model cited above.

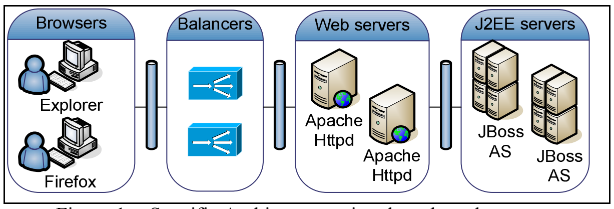


Figure 1. Specific Architecture using the selected components.

As can be seen, Fig. 2 states that as a solution to balance the load in a first stage, corporations tend to use hardware based load balancers. This way all the clients that are connected at that time to the web servers are evenly redirected to servers that will serve their requests. But, not all the requests of a client trigger the same computational needs from the back-end. These load balancers know nothing or very little about the conditions of the computer that hosts the Apache or JBoss servers, or about the thermal conditions under which the server is running. Therefore a second load balancer layer may be advisable to prevent the system to downgrade. It is possible to install load balancers in the JBoss side [6], as well as in Apache application servers [7], [8] by using a special module (namely ***mod_jk***) provided within Apache, in fact this module establishes several predefined load balancing policies to choose from which JBoss will be responsible for the computation associated to a client's request. The predefined policies are:

- Round robin: the next JBoss to be used is the one that has the oldest request. . Not advisable, see [9]
- Bytes sent: The next to be used is the one that is sending fewer bytes through the net. Useful when the network is a handicap.
- Active sessions: The selection is based on the one with the smallest number of active sessions; this is useful when the memory associated with a session is a bottle neck.

The usage of the load balancing policies into the *mod_jk* module is effective only when there is no session associated to the current request, in this case, the request can be scheduled to run in any active JBoss [10].

Currently none of the load balancing policies predefined in Apache can be adapted to serve real applications. The reasons are:

- Network traffic: this factor can be ignored due to the current network technologies, bandwidth and sharing issues provided by the communication protocols.
- Request number: only useful for homogeneous environments.
- Active sessions: 64-bits architectures and current memory technologies reduce this risk.
-

III. OUR APPROACH for LOAD BALANCING

The most used policy is the one based on the number of requests. The profile of an ideal load balancing policy that overcomes the cited issues can be found in [11] and in [12]. The load balancing strategy that is proposed in this paper considers how the application server (JBoss in this particular case) behaves with respect to the characteristics exposed in [11], in contrast to studies where load balancers are integrated into the JBoss [9], considering factors such as the number of busy threads, or the fact that this load balancer is transparent to the business logic. Our strategy is implemented in the core of the Apache web server by means of the *mod_jk* module. Results are compared with the conventional strategy based on the number of requests.

Fig. 2 shows the case of how our strategy can be used to fairly distribute requests between the JBoss servers.

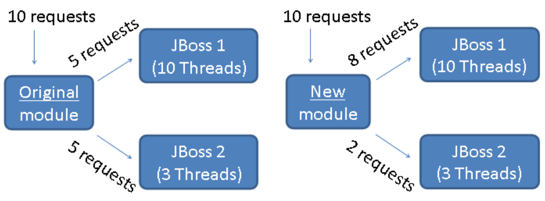


Figure 2. Sample case of the strategy developed

As can be seen in Fig. 2, if the conventional strategy is used, five requests are redirected to each one of the two active JBoss servers, of course, the balancer is not taking into consideration that each server may not be solving the same number of tasks. The original module distributes the requests equally, but JBoss2 is overloaded, it only hosts 3 idle threads and its response time is therefore high. JBoss1 is idle and therefore more request should be redirected to it, as our approach does, getting a higher average throughput.

Consequently, if a strategy is able to inspect the current resource usage of any of the application servers then it could be possible to make a better partition of the work. For this purpose we have defined several metrics:

- Busy threads: any JBoss server uses a pool of threads to execute every incoming request. If there are no idle threads for an incoming request, then the server will reject it, downgrading the performance of the whole system. We set the risk threshold to 80% to enter an overload situation.
- Busy memory: as we are running in a Java environment, the garbage collector is our cornerstone. When put into action, every component is stopped to allow a clean reorganization of the memory. Reaching a memory occupation of 90% would trigger the garbage collector and this would cause a performance issue.
- Average CPU load: this value is related to the physical processor where the JBoss is being run (not the virtual machine). This value, taken as an average, can give us a snapshot of what the situation was in the past million cycles, so making a forecast for the upcoming next thousand cycles may not be a difficult task. If we consider the number of installed cores as a reference to understand this value, then a value equal to the number of cores means 100% processor usage, which is not bad. Below this point may indicate under utilization, over this point would mean overload; this is a situation that we should avoid. We may consider that an average usage of 75-80% of the computing capabilities of the processor is a good watermark for the JBoss' health.

Our strategy tries to load balance the requests maintaining a trade between resources and system's stability. So, once all the values are evaluated, the strategy decides the percentage of requests that can be redirected to the JBoss. This load balancing procedure is based on the classical procedure that uses the number of requests [10], but it was improved to consider, also, the status of the JBoss servers that are under its control. This procedure ensures that the load balancer is acting properly and preserving the following principles:

- Dynamic, the percentage of requests is modified as soon as the JBoss status changes, which is advisable [12].
- Automatic, no human interaction is needed.
- Scalable, will not introduce any overhead into the system.

- Safe, no vulnerabilities may appear as a consequence of *mod_jk*.
- Justified, system administrators must check its behaviour through logs.

Our strategy, clearly, needs two differentiated components: one for building a status report for every JBoss, another for changing the decisions in the *mod_jk* side. To implement this we have built a system with three differentiated components:

- Standard J2EE Web component [13], [14]: integrated into the JBoss [15] server to collect performance and status information such as the ratio of threads in use versus runnable threads. This information is calculated as the HTTP request designed for it, is received. This component is also designed to change these values for test purposes.
- Shell scripting components[16], [17]: it runs beside the JBoss instance and gathers all the necessary information to build an index we named JBoss availability Index, composed by values, such as ratio of occupied/free threads –http request-, percentage of used memory after last garbage collector activity and average CPU load (obtained through system calls [18]). This component builds the availability index and presents it to the Apache module. This process runs independently from the JBoss address space, a global view of the machine is therefore available. This index must be calculated in the back-end to make this information available to each Apache server that might redirect requests to it, so it is calculated only once, periodically, and made available for every Apache server. Back-ends, also usually have more computing power.
- Apache module [19],[20]: analyze the information from the other two components and creates the percentage of requests that can be redirected and processed to each JBoss, previous studies such as [21] explore this method, our approach adds the dynamic status report evaluation gathered from the JBoss application servers.

These three components are coordinated to maintain a fair workload on every JBoss server. Fig. 4 shows how these three components coordinate and work to achieve this goal.

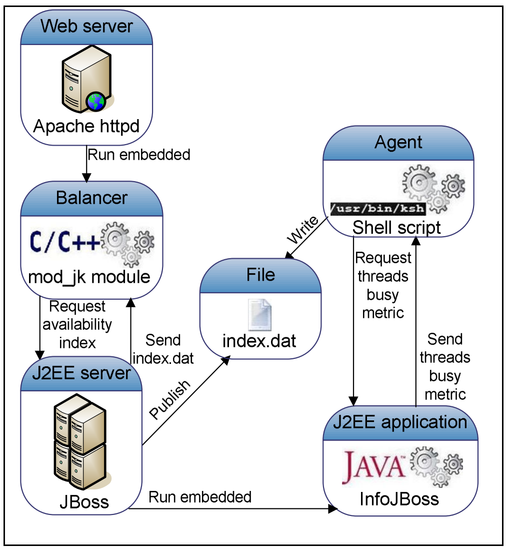


Figure 3. Global interaction between the three developed components.

The availability index is calculated periodically to define an almost real status sketch of the JBoss servers. In order to build this index it is necessary to set up three elements: metrics –cited above- , thresholds –one per metric- and penalties –values that are used to correct the availability index when the thresholds are overcome-. Table I shows metrics, thresholds and penalties to build our *availability index*.

TABLE I. HOW TO BUILD THE AVAILABILITY ÍNDEX.

Metrics: ○ ***Mt***: ratio (per thousand) of busy / idle threads by AJP1.3 (port 8009). ○ ***Mm***: % used memory in the JVM alter “garbage collector” operation ○ ***Mc***: average load of the processor during the last minute.
Thresholds: To define the transition from a healthy state to a risky state. ○ ***Ttw***: Threshold related to threads in use for "warning" state. Ttw = 800. ○ ***Ttc***: Threshold (threads in use) for "critical" state. Ttc = 900 ○ ***Tmw***: Memory threshold for "warning" state. Tmw = 70 ○ ***Tmc***: Memory threshold for "critical" state : Tmc = 85 ○ ***Tcw***: average CPU load threshold for "warning" Tcw = 6 ○ ***Tcc***: average CPU load threshold for "critical" Tcc = 12
Penalties: ○ ***Pt***: Penalty for excessive threads in use. Pt = { 0 (Mt < Ttw); 2 (Ttw <= Mt < Ttc); 5 (Mt >= Ttc) ○ ***Pm***: Penalty for excessive memory used after a Garbage Collector operation. Pm = { 0 (Mm < Tmw); 2 (Tmw <= Mm < Tmc); 5 (Mm >= Tmc) ○ ***Pc***: Penalty for exceeding the average load Pc = { 0 (Mc < Tcw); 2 (Tcw <= Mc < Tcc); 5 (Mc >= Tcc)

We define a one digit based availability Index (0 <= I <= 9) for each running instance of JBoss application server, composed as follows

$$I = \begin{cases} 9-(Pt+Pm+Pc) & \text{if } (Pt+Pm+Pc <= 9) \\ 0 & \text{if } (Pt+Pm+Pc > 9) \end{cases}$$

Once the environment is defined and the metrics are exposed together with the method used to compose the availability index, we run tests to verify the performance and efficiency achieved, Fig. 5 shows the tested configurations used.

Although the strategy can be ported to any application which complies with a three tier architecture and is based in a transactional model, our approach was tested in the context of the cryptographic web service, that build transactions for each customer for authentication purposes. Although the operations are against a huge data matrix, they are all the same. We configured three different sets of workload suites to test it, as depicted in section IV.

IV. Results and Conclusions

To evaluate the performance of the web server and application servers, we used JMeter [22], a tool from the Apache foundation to monitor performance. Tests are launched in pairs, we launch a workload with the original *mod_jk* and after that we repeat the workload with *mod_jk* (servers are restarted to avoid warm caches and index contamination due to CPU average variations). Table II shows the resources available for the tests.

TABLE II. CONFIGURATION AND RESOURCES

JBoss configuration table		
Physical Memory	JVM Memory	Tomcat Threads
1536 MB	1280 MB	200

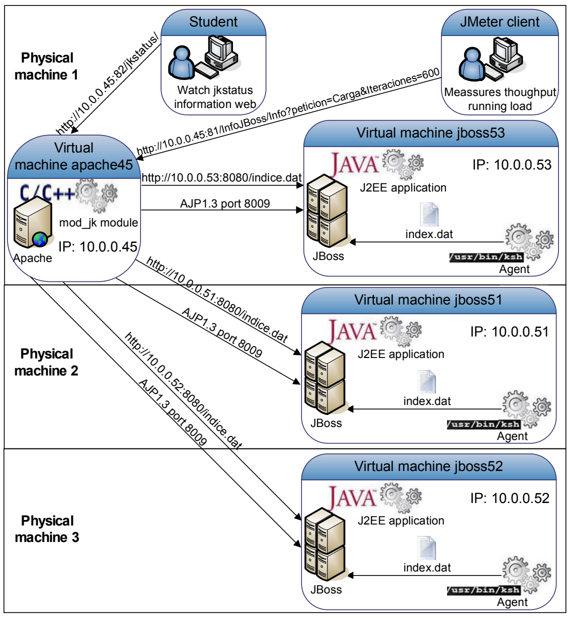


Figure 4. Configuration used to simulate a real application scheme.

Each workload has three configurations:

- Average workload, which lasts from five to ten minutes. Normal to high client concurrence (60 clients) and from 120 to 300 requests per client.
- High workload: moderate duration (ten to fifteen minutes). High client concurrence (80).
- Durable workload: which lasts from twenty to thirty-five minutes, 80 clients and 300-600 requests/client.

Fig. 6 graphically shows the evolution of each availability index for durable workload test, using three JBoss servers.

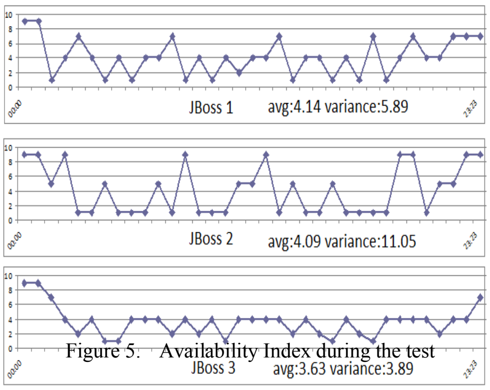


Figure 5. Availability Index during the test

If the evolution of the application servers is analyzed, using the data plotted in Fig. 6, it can be said that JBoss1 was the best performing server (best average and low variance). JBoss2 had a similar index as JBoss1 but its high variance indicates that it suffered very different periods of load. JBoss3 experienced the worst I index but the best variance and therefore the best stability.

A. *Metrics and their values gathered during the tests*

This subsection will show the metrics gathered for each test. Results show that our method performs better than the original method. We have collected values for performance, throughput, average time and maximum time for solving a request.

Table III shows the configurations used in the system for each type of workload defined for the tests. Table IV and table V, contain the results obtained from the tests launched using two JBoss servers and three JBoss servers respectively.

TABLE III. CONFIGURATION PARAMETERS USED FOR THE TESTS

Configuration parameters for the tests			
Average workload			
Servers	*Number of concurrent clients*	*Requests per Client*	*Total requests*
2	60	120	7200
3	60	300	18000
High workload			
Servers	*Number of concurrent clients*	*Requests per Client*	*Total requests*
2	80	120	9600
3	80	300	24000
Durable workload			
Servers	*Number of concurrent clients*	*Requests per Client*	*Total requests*
2	80	300	24000
3	80	600	48000

TABLE IV. METRICS OBTAINED FOR TWO JBOSS SERVERS.

Average workload					
Load Bal. Strategy	***Performance (sec)***	***Throughput (requests/sec)***	***Average time/request***	***Max. T./ request***	***90% TimeLine***
Original	402	17,9	3163	26062	9992
New	378	19,2	2951	18968	8362
High workload					
Load Bal. Strategy	***Performance (sec)***	***Throughput (requests/sec)***	***Average time/request***	***Max. T./ request***	***90% TimeLine***
Original	550	17,5	4366	26661	12682
New	500	19,2	3912	30774	10890
Durable workload					
Load Bal. Strategy	***Performance (sec)***	***Throughput (requests/sec)***	***Average time/request***	***Max. T./ request***	***90% TimeLine***
Original	1368	17,5	4462	30334	16939
New	1150	20,9	3712	26042	10043

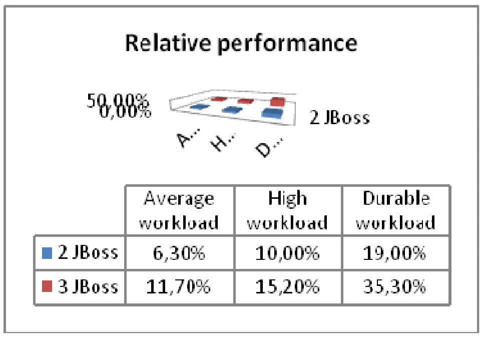
The performance measurement is related to the number of seconds in solving (from start to end) a request. Throughput is related to the number of requests solved per second; this metric is useful in tracking the server's behaviour when submitted to load peaks. The average time per request is that related to the average of the times for all the requests during a workload test; from each workload we picked the request that lasted longest and kept this time as the maximum time (useful for tuning purposes and defining timeout intervals). We also show in 90% TimeLine column, the maximum time reached when we consider the 90% of the values with less dispersion (ignoring values whose dispersion is high).

TABLE V. METRICS OBTAINED FROM THREE JBOSS SERVERS.

Average workload					
Load Bal. Strategy	***Performance (sec)***	***Throughput (requests/sec)***	***Average time/request***	***Max. T./ request***	***90% TimeLine***
Original	622	29	1929	21216	7191
New	557	32,3	1736	18873	5907
High workload					
Load Bal. Strategy	***Performance (sec)***	***Throughput (requests/sec)***	***Average time /request***	***Max. T./ request***	***90% TimeLine***
Original	909	26,4	2905	27575	10519
New	789	30,5	2515	29840	8546
Durable workload					
Load Bal. Strategy	***Performance (sec)***	***Throughput (requests/sec)***	***Average time /request***	***Max. T./ request***	***90% TimeLine***
Original	1898	25,3	3042	101277	10266
New	1403	34,2	2067	62224	6173

We can see in Table VI and Fig. 7, using the strategy we propose, remarkable improvements in performance. The tests show improvements of 35% for durable workloads. Throughput also follows the same trend, improving when the workload is incremented to high workload, we can see that we even obtain improvements when the workload is upgraded to a durable workload (maintaining the same number of concurrent clients), this is accomplished by doubling the number of requests per client. The improvements are more remarkable the longer the test is being run. The longer the workload, the lower overhead we reported by the proposed method.

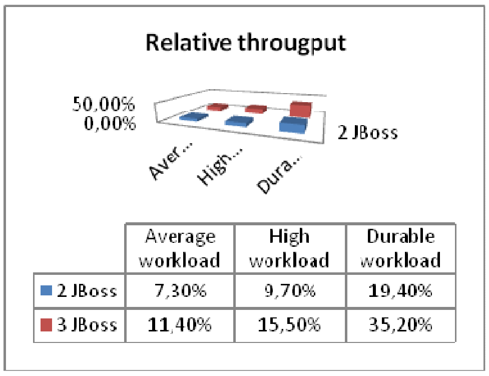
The relative improvement obtained gathering all metrics obtained, is analyzed in the upcoming datasets. The performance is related to the number of seconds employed in having a request fulfilled. Low performance, therefore, means that for a certain unit of work, the system resources were underused, so they will be available to do more work.



	Average workload	High workload	Durable workload
🟦 2 JBoss	6,30%	10,00%	19,00%
🟥 3 JBoss	11,70%	15,20%	35,30%

Figure 6. Relative performance improvement

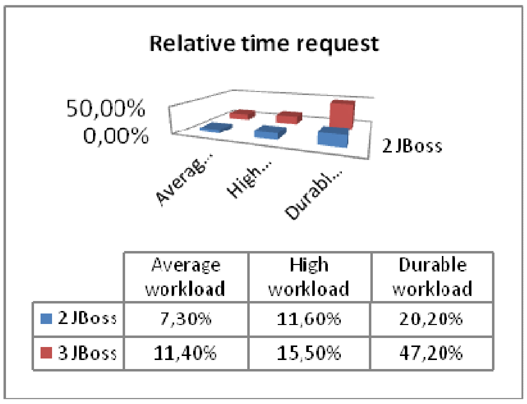
As it can be seen in figure 6, a 35% of improvement was obtained with the module designed according to the load balancing strategy proposed, during the durable workload for 3JBoss servers. regarding throughput, fig. 7, it is related to the average number of requests solved per second. A higher throughput means a better utilization of hardware, and the ability to server more work at the same expense. This metric can reflect how the system behaves when submitted to peaks of load.



	Average workload	High workload	Durable workload
🟦 2 JBoss	7,30%	9,70%	19,40%
🟥 3 JBoss	11,40%	15,50%	35,20%

Figure 7. Relative througput improvement (requests/sec)

As we can see, there is a very evident throughput improvement of a 35%, like performance improvement, with the new module for the same scenario: durable workload and 3JBoss. This improvement is higher when we the load is increased, moreover this happen also when the time of the test is increased with the same number of concurrent clients (doubling the number of request of each client). Consequently, we get a better throughput when the system is running more time. This can be justified because the overhead is lower due to the load addressed based in the new index. The JBoss that receive more loads has to generate a higher number of resources like threads and memory to process them and are useful for new requests without more overhead. When referring to Time by Requests, see Fig. 8, in seconds, it is related to the average computed running time of all **URL** requests send in the test. Less time by requests means that a better feeling is transmit to the user because a less time is waiting for the answer and less time the resources are captured. It exists less possibility of blockings avoiding bottle necks and in consequence a great scalability.

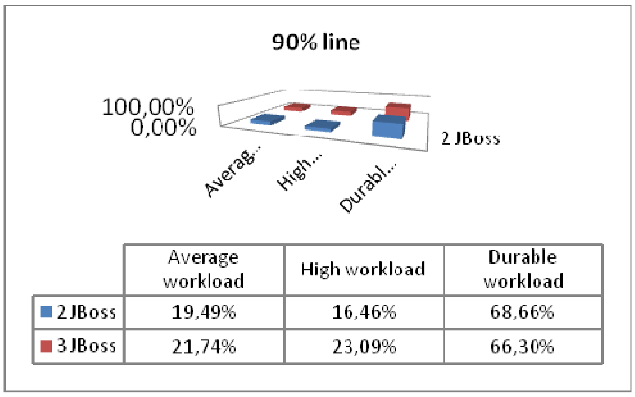


	Average workload	High workload	Durable workload
🟦 2 JBoss	7,30%	11,60%	20,20%
🟥 3 JBoss	11,40%	15,50%	47,20%

Figure 8. Relative time/request improvement

Using this metric the gains when using our strategy are more than evident, this metric shows the best relative improvement. Requests are 47,2% faster using the new model, for the case of having 3 JBoss and a durable load.

90% Time Line is related to those samples whose response is in the 90% of the timings, the "rare" cases where rejected, therefore this metric offers the point of view of reliability.



	Average workload	High workload	Durable workload
🟦 2 JBoss	19,49%	16,46%	68,66%
🟥 3 JBoss	21,74%	23,09%	66,30%

Figure 9. Improvements achieved inside of 90% line

Results shows that the new Load Balancing strategy proposed in this work has meant a higher improvement in all tests. Next table summarizes the relative improvement values get for the metrics in the three different situations.

TABLE VI. RELATIVE IMPROVEMENTS ACHIEVED FOR EACH TEST.

Relative Improvement						
JBoss	***Workload***	***Performance***	***Throughput (requests/sec)***	***Time / request***	***90% TimeLine***
2 JBoss	Low workload	6,3%	7,3%	7,2%	19,49%
	High workload	10,0%	9,7%	11,6%	16,46%
	Durable workload	19,0%	19,4%	20,2%	68,66%
3 JBoss	Low workload	11,7%	11,4%	11,1%	21,74%
	High workload	15,2%	15,5%	15,5%	23,09%
	Durable workload	35,28%	35,18%	47,17%	66,30%

The average time per request is the value that experiments the best improvements when the new method is used. As it can be seen, values can achieve improvements of up to 47.2% when using three JBoss servers and a durable workload, and improvements of 66.3% if we consider the 90% of the values, with less dispersion.

B. *Conclusions*

It has been shown that the proposed load balancing strategy and its integration into application servers, comes down to performance benefits for the applications (notable performance and throughput improvements). In addition, servers can be equipped with a tool to dynamically adapt the requests that must be redirected to each of the JBoss servers available. A direct consequence of using the strategy proposed is that aspects like system stability, resource usage and availability, global system performance and user experience were directly improved.

ACKNOWLEDGMENT

This work has been funded by grants from the Spanish Ministry of Science and Innovation (TIN2008-01117), and Junta de Andalucía (P08-TIC-3518), in part financed by the European Regional Development Fund (ERDF).

REFERENCES

[1] G. Huang, W. Wang, T. Liu, H. Mei, “Simulation-based analysis of middleware service impact on system reliability: Experiment on Java application server”, Journal of Systems and Software, Volume 84, Issue 7, ,Pages 1160-1170, ISSN 0164-1212, July 2011.

[2] J. Guitart, D. Carrera, V. Beltran, J. Torres, E. Ayguade, “Designing an overload control strategy for secure e-commerce applications”, Computer Networks, Volume 51, Issue 15, 24, Pages 4492-4510, October 2007.

[3] K. Birman, R. van Renesse, W. Vogels, "Adding high availability and autonomic behavior to Web services," Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on , vol., no., pp. 17- 26, 23-28 May 2004

[4] A. Guruge, Java and Web Services, Web Services, Digital Press, Burlington, 2004, Pages 227-270, ISBN 978-1-55-558282-1, DOI: 10.1016/B978-155558282-1/50008-7.

[5] H. Xiaotao, D. Chaozhi, “Design of high-available architecture for distributed application based on J2EE and its analysis[JA]. Huazhong Keji Daxue Xuebao (Ziran Kexue Ban)/Journal of Huazhong University of Science and Technology (Natural Science Edition).2005, 44-47.

[6] Y. Liu, L. Wang, S. Li, "Research on self-adaptive load balancing in EJB clustering system," Intelligent System and Knowledge Engineering, 2008. ISKE 2008. 3rd International Conference on , vol.1, no., pp.1388-1392, 17-19 Nov. 2008

[7] T. Bourke, T. Server Load Balancing. O'Reilly & Associates Press. 2001.

[8] V. Viswanathan. Load Balancing Web Applications. OnJava.com. 2001. http://onjava.com/pub/a/onjava/2001/09/26/load.html

[9] G. Lodi, F. Panzieri,, D. Rossi, E. Turrini, "Experimental Evaluation of a QoS-aware Application Server," Network Computing and Applications, Fourth IEEE International Symposium on , vol., no., pp.259-262, 27-29 July 2005

[10] Apache Software Foundation. The Apache Tomcat Connector - Reference Guide. 2010. http://tomcat.apache.org/connectors-doc/reference/workers.html

[11] [11] K. Gilly de la Sierra-Llamazares. Tesis: An adaptive admission control and load balancing algorithm for a QoS-aware Web system. Universitat de les Illes Balears. 2009. http://www.tesisenxarxa.net/TDX-1211109-113725/index_cs.html

[12] H. Elmeleegy, N. Adly, and M. Nagi. "Adaptive Cache-Driven Request Distribution in Clustered EJB Systems". Proceedings of the Tenth International Conference on parallel and Distributed Systems (ICPADS'04), 179-186, 2004,

[13] R. Johnson,. Expert One-on-One J2EE Design and Development. Wrox Press. 2003.

[14] R. B'Far, Mobile Computing Principles: Designing and Developing Mobile Applications with UML and XML. Cambridge University Press. 2005.

[15] N. Richards, S. Griffith. JBoss: A Developer's Notebook. O'Reilly, 2005.

[16] M. Garrels, Bash Guide for Beginners. 173 págs. 2008.
http://tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf

[17] M. Cooper. Advanced Bash-Scripting Guide: An in-depth exploration of the art of shell scripting, 2010. http://tldp.org/LDP/abs/abs-guide.pdf

[18] L. Wirzenius, J. Oja, S. Stafford, and A. Weeks. The Linux System Administrators' Guide. 2005. http://tldp.org/LDP/sag/sag.pdf

[19] N. Kew, The Apache Modules Book: Application Development with Apache. Prentice Hall. 592 págs, 2007.

[20] Apache Software Foundation. 2010. The Apache Tomcat Connector. Quick Start HowTo. http://tomcat.apache.org/connectors-doc/generic_howto/quick.html

[21] K. Suryanarayanan, K.J Christensen, "Performance evaluation of new methods of automatic redirection for load balancing of Apache servers distributed in the Internet," Local Computer Networks, 2000. LCN 2000. Proceedings. 25th Annual IEEE Conference on , vol., no., pp.644-651, 2000.

[22] Q. Wu, Y. Wang, "Performance Testing and Optimization of J2EE-Based Web Applications," Education Technology and Computer Science (ETCS), 2010 Second International Workshop on , vol.2, no., pp.681-683, 6-7 March 2010.

[23] El Hadad, J.; Manouvrier, M.; Rukoz, M.; , "TQoS: Transactional and QoS-Aware Selection Algorithm for Automatic Web Service Composition," *Services Computing, IEEE Transactions on* , vol.3, no.1, pp.73-85, Jan.-March 2010
doi: 10.1109/TSC.2010.5
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5432150&isnumber=5440876

[24] Picker, R.C.: The yin and yang of copyright and technology. Commun. ACM **55**(1) (2012) 30{32

[25] J.A. Alvarez-Bermejo J.A. Lopez-Ramos. Tracking Traitors in Web Services via Blind Signatures. http://hdl.handle.net/10835/1520