Jerrymouse: a tool for a flexible and dynamic distribution of web service requests

Paulo S L de Souza, Bruno S Faiçal,
Marcos J Santana, Regina H C Santana
University of Sao Paulo - SSC/ICMC
Sao Carlos, Brazil
{pssouza, bsfaical, mjs, rcs}@icmc.usp.br

Jonathan de Matos
State University of Ponta Grossa - UEPG
Ponta Grossa, Brazil
jonathan@uepg.br

Ed Zaluska
Electronic and Computer Science
University of Southampton
Southampton, United Kingdom
ejz@ecs.soton.ac.uk

***Abstract*— This paper presents a novel architecture for distributing web service requests on clusters of servers. The architecture facilitates a transparent dynamic distribution of requests according to a range of specified policies. This enables a flexible performance in respect of different objectives, services and platforms (typically based on server workload). The architecture has been successfully demonstrated with a prototype implementation (called "Jerrymouse"). Our preliminary results with Jerrymouse indicate stable behaviour and worthwhile performance gains (compared with Apache HTTP Server). A specific policy to deliver reduced cluster electricity savings has also been successfully implemented.**

Keywords- web services, SOA; distributing requests; clusters; monitoring; Ganglia.

I. INTRODUCTION

Service Oriented Architecture (SOA) allows the design, implementation and use of services from a number of different sources in a loosely-coupled and interoperable way [1]. Web services provide a mechanism to expose services for client applications using protocols such as HTTP. Web services typically use languages such as XML and WSDL (Web Service Description Language) together with protocols such as SOAP (Simple Object Access Protocol) and REST (Representational State Transfer). SOAP and REST describe how parameters are sent to and received from services while WSDL is a language that describes how to access web services, including data types and functions [1].

Web services typically require at least two server-side sub-systems: an application server and a web server. An application server implements the delivery of the specific service requested. JBoss, IBM WebSphere, Apache Axis and Apache CXF are all examples of application servers with different levels of functionality [1][2]. Web servers process messages between clients and services using HTTP. JBoss and WebSphere implement web servers as well. JBoss derives from Apache Tomcat, with optimizations to process requests with dynamic content using JSP (Java Server Pages) and servlets [2]. Apache Tomcat was originally a servlet reference implementation and this is one of the reasons for its popularity. Frameworks can be inserted into Apache Tomcat for web services execution, allowing both publication of these services and interaction with clients and other providers using HTTP.

Web services support distributed solutions which can provide greater performance, scalability, fault tolerance and good availability for systems with a high demand. Despite these clear advantages, distributed solutions usually increase the overall system complexity when compared to centralized alternatives [3]. 'Request distribution' on clusters of web services is an example [4] which can be implemented in several different ways: by DNS, by NAT, by interaction between the Apache HTTP Server and Apache Tomcat, in the EJB layer or even in the database layer [5].

DNS distribution has the potential for a simpler implementation, but normally has problems with caching, the target server state is not known and fault tolerance is not supported. NAT distribution offers better support for fault tolerance, but it has the same difficulty of obtaining state information from the target server. When applying request distribution in the database layer, the service will redirect the request to a database cluster, considering load balancing in this level. The distribution in database servers does not avoid an eventual bottleneck in the web service nodes. MySQL Cluster [6], PG Cluster [7] and Slony [8] are some examples of this kind of distribution. The request distribution performed by interaction between the Apache HTTP Server and Apache Tomcat take into account the resources accessed by the Apache HTTP server and hence achieve a potentially optimum distribution. One possible implementation is the mod_jk module (part of the Apache HTTP Server) which supports a range of distribution policies: round-robin, weighted round-robin, busy (i.e. considering the existing server load), sections and network use. These policies present significant performance-limitations because there is

978-0-7695-4462-5/11 $26.00 © 2011 IEEE
DOI 10.1109/SCC.2011.90

no dynamic feedback about the platform current state nor flexibility to distribute the submitted requests according to service demand.

The systems supporting web services ideally require a flexible request distribution that offers a better efficiency to the diversified demand generated by different services on different distributed platforms. Transparency and portability are equally important and desirable features. Monitoring of the current server node status can be used to support an improved dynamic distribution because it is based on more accurate decisions. Some works considering monitoring in this context can be seen in [9] and [10].

Many studies have been developed at "processes scheduling" for High Performance Computing (HPC) and Distributed Systems (DS). However, workloads distributed on servers of web services are not "processes"; but "requests" to services. Thus, processes scheduling policies need to be adapted to the web services context, where its servers have distinct features as: specific middlewares to support the services, demands with services composition, objectives, protocols and a high use of remote data bases.

Considering the web service context, this work proposes an original architecture for a dynamic distribution of client requests, offering flexibility, transparency and portability. Flexibility allows a weak coupling between hardware and software and is achieved by the use of several policies to distribute the services. The proposed architecture has been implemented in a prototype called Jerrymouse, which demonstrates portability using different service providers, such as Apache Tomcat and JBoss. The architecture and prototype will act as a basement for future research works in this area.

The performance of both the proposed architecture and Jerrymouse has been evaluated using experimental studies, performed on two different platforms: one homogeneous and one heterogeneous. These experimental studies make use of four different distributing policies, three of them similar to the policies already existing in the mod_jk module (part of the Apache HTTP server). This approach allows the overhead introduced by Jerrymouse to be measured when it is implementing a distribution algorithm similar to the Apache HTTP Server. The fourth distribution policy implemented allows explicit control of the number of server nodes currently powered-up according to the overall system load, hence reducing costs when possible by saving energy.

The results presented in this paper show that the proposed architecture does not introduce a significant execution overload and, at the same time, offers significant performance gains and improves the service availability when compared to Apache HTTP Server.

This paper is composed of 5 sections. Section II describes the proposed architecture in a top-down approach. Section III describes the Jerrymouse prototype. Section IV discusses the main results obtained from the experimental studies with Jerrymouse. Section V presents the conclusions.

II. ARCHITECTURE

Fig. 1 presents the proposed architecture together with a message using SOAP. The SOAP message is received by a network border element (step 1), implementing a NAT or DNS distribution logic. After it has arrived, the request is redirected to a web service provider acting as a front-end, which will then either redirect the request to a target node or service the request locally. In this example, the provider is located in node 2. The decision to redirect is taken by a novel distributor element in the provider (step 2), using a policy to decide which distribution to apply. This policy uses data from monitors running in every service node in the overall distributed platform. The distributor selects an appropriate service node and responds to the provider (step 3) with the required destination. The provider forwards the message to the destination node (step 4) in order to execute the service. The target node, node 1 in this case, receives the request using its framework (step 5). The response from the service reaches the front-end first (step 6) and then the provider forwards it (step 7) to the client. There are alternative solutions, using connections-migration, which allow the service response to be forwarded directly to the client [11]. However, these solutions require changes either to the system core or to the TCP protocol for both servers and clients. This means that this alternative is unattractive because transparency is usually required on the client side and modifications inside the system core will make it difficult to maintain independence between software and hardware.

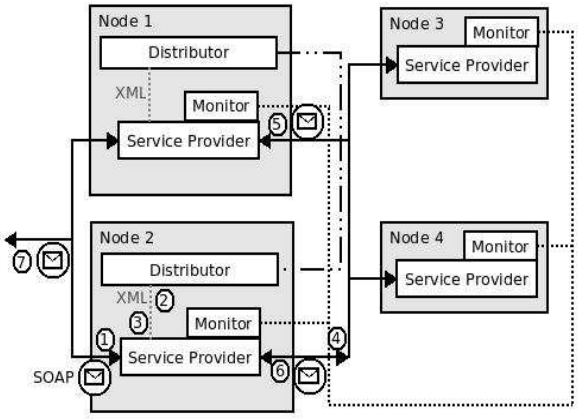

Figure 1. Novel architecture for distributing requests to web services providers.

The proposed architecture does not mandate a rigid (and inflexible) location for the software and hardware components. The front-end can respond to a service request directly (if allowed by the policy and the current load distribution). More than one front-end can co-exist if there is sufficient client demand. When this occurs a logical link is required between the front-ends in order to provide consistency for overall system. Nodes 3 and 4 do not have a distributor because they are back-end servers and thus not able to receive requests directly from a client.

A filter is inserted into the service provider enabling it to communicate with the Distributor (see section 3). Fig. 2 gives the internal components of the Distributor, which are

described below. In Fig. 2 we can also see the components responsible to connect the Distributor to Tomcat, Ganglia and other Distributor.

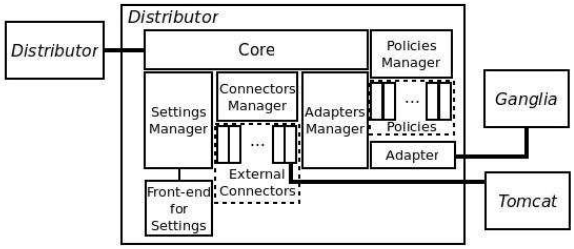

Figure 2. Distributor internal components.

A. The Distributor

The Distributor components are currently the Core, the Setting Manager, the Adaptors Manager for Monitoring, the Connector Manager and the Policies Manager. In addition, there are also Adapters for Monitoring, Connectors and the Requests-Distribution Policies. These components are dynamic and can be added, removed or replaced at run-time. This approach increases the flexibility of the architecture, since they can be created and compiled as separate libraries. Adapters for monitoring support the communication between the Distributor and the Monitors. The Connectors support communication between the Distributor and the target web service provider. The Policies define the high-level system requirements and effectively decide how to redirect the requests.

B. The Core

The Core is the main Distributor element (the first one to be executed), responsible for managing all other components and acting as a bridge between connectors, policies and monitors. The Core loads the settings at the start of Distributor execution and then loads the Managers that stay running concurrently.

C. The Setting Manager

The Setting manager unifies all information about the architecture, so that every Distributor component can access it. Its initial functionality is to load the system settings (described in an XML file) and then validate them using XML Schema or DTD. The parsing of the settings information creates a binary representation which is used by the components. When the initial setting is complete, the Setting Manager remains active and returns execution back to the Core. New settings can be established at runtime through a direct interaction from the system administrator or from another distributor. This Distributor dynamic reconfiguration does not require execution to be suspended. The Settings Manager can also change the status of Connectors, Adapters and Policies between active and inactive and in addition these components can also be deleted and inserted. This process requires that the configuration´s data are always consistent because the Distributor remains active and is continually receiving service requests for distribution. The Settings Manager also propagates updates to other Distributors and can register their actions in a log file. The granularity of the stored information can be configured in the settings file.

D. The Connectors Manager

Connectors ensure that the Distributor is loosely coupled to the web service providers and removes from the Core the necessity to store the interaction details required to communicate with different providers. All Connectors use a standard interface with the Core and are responsible for managing any differences. The Connectors Manager reads the connectors settings already established by the Setting Manager and executes them. The Connectors Manager maintains a reference to all active connectors and when it receives a new instruction from the Setting Manager it notifies the connector as appropriate (e.g. activation, deactivation (but remain in memory), inclusion or removal).

E. The Policies Manager

The provision of multiple policies provides maximum flexibility for the Distributor. The design concept is to make available a number of different policies able to redirect service requests. Using the same mechanism implemented for the Connectors Manager, this manager also receives notifications from the Settings Manager. When the Policy Manager starts, it uses the data present in the Distributor to load the required policies. Policies, unlike connectors, do not remain active during Distributor execution. They are executed only when a decision is requested by the Distributor, saving processor clock cycles and memory space. The decision to avoid continuous execution was taken because probably there will be several policies running concurrently to support different service requests. Policies can be enabled, disabled, added and removed using this Manager.

F. The Adaptor Manager for Monitoring

The Adapter Manager loads the adapters that are connected to all of the monitors present in the system. It can enable, disable, add and remove adapters from the Distributor. Besides this management role, it also serves as the bridge between politics and adapters. The adapters are started by this Manager and remain active until a closing notification is received. The policies are able to load indices internally from a specific function that uses the Adapters Manager to query the active adapters (all adapters are queried until information is returned). The internal structure of adapters is designed to avoid bottlenecks in the expected return of information. Another concern is the complexity of the adaptors, which has been minimized because they are all queried and this therefore imposes a limit on the scalability of the architecture at present.

III. JERRYMOUSE

The architecture described above has been successfully implemented in a prototype called Jerrymouse. The Jerrymouse implementation has required the development of both distributor and filter elements associated with the web

services provider. The implementation was based on C, Perl and programming scripts in GNU/LINUX. Fig. 3 shows the relationship of Jerrymouse with the Apache Tomcat provider.

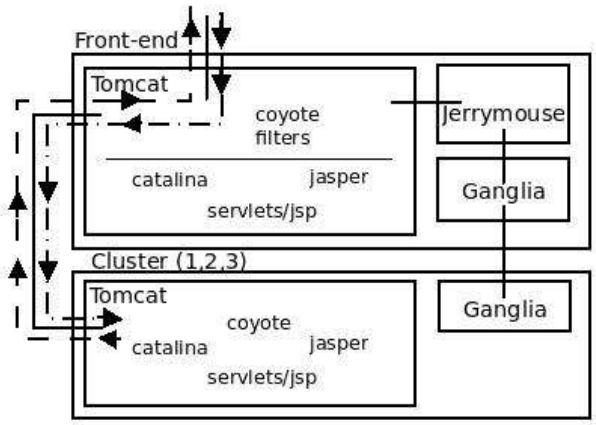

Figure 3. Interactions between Apache Tomcat, Jerrymouse and Ganglia.

The filter is a design pattern of the servlets specification that implements a sequence of steps to be executed between two elements. In Apache Tomcat, for example, filters are running in the HTTP server called Coyote, the Catalina servlet container and Jasper (which is responsible for executing servlets). The parameters and the content of the HTTP request are accessible by these filters. The Jerrymouse implementation requires the inclusion of a filter in Apache Tomcat to intercept HTTP requests and then send a request to Jerrymouse asking which node should be used to run the service. Jerrymouse determines the policy to be used, applies this policy to select the service node to forward the request to and then returns this back to the filter. The filter in the provider recreates the HTTP request and sends it to the required node. The HTTP message is received by the Coyote component in the target node, which invokes the Catalina and Jasper modules for service execution. When the service is complete, the front-end (where the filter was invoked) receives the response and returns the result to the client. In this example, the front-end provider has used just the HTTP Server, because the filter inhibits the usual request-path to the Catalina and Jasper modules. In the event that there is any failure in Jerrymouse or in the communication with other remote providers, the filter can forward the message directly to the local Catalina and Jasper modules to provide a local fall-back execution of the service. This can also be an advantage in times of low client demand and hence low overall workload. Installing the filter inside Apache Tomcat is not intrusive, since there is no necessity to change any source code. For installation, the filter class has to be enabled in the Apache Tomcat class loader and then the settings file can be changed for this class to run as a filter.

Fig. 3 also shows the relationship between Jerrymouse and the Ganglia monitoring tool. The choice of Ganglia is because of its scalability and existing use on large platforms [12]. It operates in a hierarchical way and supports cluster federation, offering monitoring either periodically or after state changes. The Ganglia monitor runs on all cluster nodes that need to be monitored and the monitoring information can be obtained directly via TCP/IP. New metrics can be inserted into Ganglia, making it a highly-flexible monitoring tool and thus contributing to the overall flexibility of Jerrymouse as well.

Jerrymouse connects with Ganglia via an adapter using a TCP/IP connection and stores the collected load indices from nodes in a hash table. When a policy requires data from the indices, the search for the desired value can be undertaken in O(1). The memory storage complexity is proportional to the amount of indices and nodes. The communication between the Ganglia monitors and Jerrymouse is non-blocking and therefore does not interrupt Jerrymouse execution. Managers, adapters and connectors are all executed using threads hence all of the adapters remain active collecting data from monitors asynchronously. All communication between Jerrymouse and adapters is performed using shared memory. When a connection arrives, the connector calls the appropriate function inside the Core which performs a query on a hash table containing web services and policies before invoking the appropriate policy.

Connectors, adapters, and policies are all loaded using dynamic libraries which ensures that elements can be loaded and unloaded without interrupting Jerrymouse execution.

The policies are the most important elements in Jerrymouse, because they can specify different distributing behaviours for each individual web service. Policies can nevertheless have simple implementations and can make use of data persistence.

IV. RESULTS

The performance of the overall proposed new architecture and the Jerrymouse prototype implementation were compared to the existing solutions provided by the standard Apache HTTP Server, which uses the mod_jk module to provide connections to Apache Tomcat. The objective is firstly to demonstrate that Jerrymouse does not generate significant overhead and thus to reduce the request performance as experienced by the client application.

The experiments described here have considered two example web services. The first example service models financial transaction authorization using credit cards, using a service based on EJB with requirements for data validation and data persistence. The second example service supports the recognition of characters and images and was implemented using the framework provided by Axis2 and Apache Tomcat.

Each service was requested from 3 concurrent clients, each one generating 1, 3, 5, 7 and 10 threads. Each thread requested 100 times sequentially the same service. In this way, considering 10 threads from 3 clients, for example, there were 30 clients performing a total of 3000 requests.

Fig. 4 illustrates the platform structure used in the experiments. Mod_jk (fig. 4a) runs in the Apache HTTP Server, redirecting messages to the nodes using Apache Tomcat. Jerrymouse (fig. 4b) uses Apache Tomcat directly in the front-end to receive requests to the web services.

Jerrymouse is also connected directly to the monitors that gather the workload indices from each node.

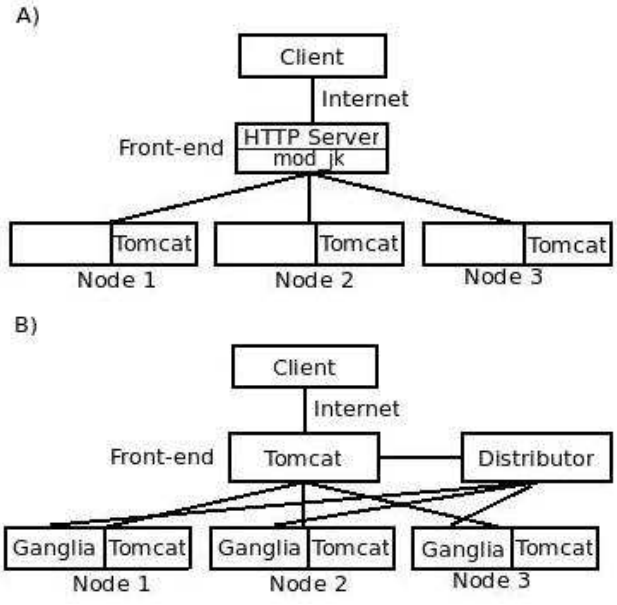

Figure 4. Experimental platforms used for the experiments (a) homogeneous and (b) heterogeneous

The experiments were performed using two different platforms: one homogeneous and the other heterogeneous (in respect of constituent nodes). The tests conducted in the homogeneous platform used six nodes: thee of these being clients, one front-end and two web service servers. The heterogeneous platform used eight nodes: three clients, one front-end and four servers. The clients and front-end systems used Intel Core 2 Quad processors in both scenarios (heterogeneous and homogeneous). The heterogeneous platform used servers on processors ranging from 800 MHz up to 1.8 GHz and RAM memory from 256 MBytes up to 512 MBytes. These same nodes in the homogeneous platform have a clock of 3.4 GHz and 2 GBytes of memory. For all experiments a 100Mbps network was used.

Three different pairs of policies were compared in the experiments. The first comparison considered the round-robin policy (as used in mod_jk) with a round-robin variant, especially developed for Jerrymouse. The goal is to identify possible overloads in Jerrymouse when it uses similar distribution policies to mod_jk on homogeneous platforms. The second comparison considered weighted round-robin policies on both mod_jk and Jerrymouse, allowing the behaviour of Jerrymouse on heterogeneous platforms to be investigated. The third comparison considered the server-occupation policy (busy) used in mod_jk with a Jerrymouse policy based on memory and processor usage, using an exponential moving average in order to reduce peaks of performance.

The policy flexibility developed for Jerrymouse has been analyzed with the use of a GreenPolicy aiming to reduce the energy consumed by the overall distributed platform. Liu et al. [13] and Bertini et al. [14] present prior research in this area and other previous paper have also discussed this approach [15][16][17].

The GreenPolicy developed in this paper is designed to activate nodes “on demand” using workload monitoring. The node activation is based on the WOL (Wake-on-Lan) resource, thus reducing the electricity consumption of the nodes. A server turned on all day long for one year can present consumption of 523.8Kwh/year, according to EU Energy Star [18]. Considering that this server can be required just 6h/day, when there is high demand, the GreenPolicy could be used to save 392.85KWh/year, just for one server. This economy is meaningful in large data centers and could be applied orthogonally to the clients and requested services.

Due to limited space, only the main results will be highlighted in this paper although more extensive results are now available, with similar results to the results presented here. All graphs shown in Figures 7 up to 10 consider the web service implementing financial transaction authorisations.

Fig. 5 shows the results for the homogeneous platform and policies round-robin. It is possible to observe that the Jerrymouse performance was superior to the mod_jk results at all load levels. This confirms that the proposed Jerrymouse structure does not increase the processing overhead significantly (which could potentially invalidate the advantages of the dynamical flexibility provided by the architecture).

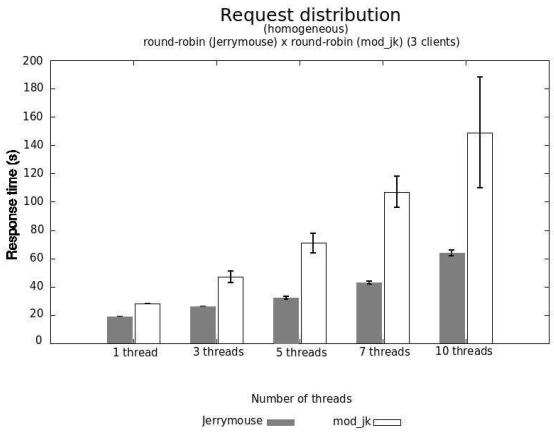

Figure 5. Results for the homogeneous platform and policies round-robin executing the financial transaction authorization service.

Fig. 6 shows the results for the homogeneous platform, but now using the mod_jk busy policy and Jerrymouse policy based on memory and processor usage. It is possible to observe that both Jerrymouse and mod_jk present a statistically similar performance, determined by hypothesis tests and by a high confidence interval from mod_jk. This result is probably due to the load index used by the Jerrymouse policy, which does not match the demand generated by the service.

Fig. 7 shows the results for the heterogeneous platform with a round-robin policy. Jerrymouse provided superior

performance for all cases, with the exception of the test with 10 threads. This occurred because of the behaviour of requests by mod_jk when it was under high demand - it did not execute the service, instead returning an "overloaded" message to the client. Jerrymouse in contrast served all requests, even when a high delay would be necessary. This scenario demonstrates that Jerrymouse has provided a higher availability than mod_jk, although of course the policy can be adjusted to meet different requirements and discards requests in a similar fashion to mod_jk if required. Such a policy might be important to guarantee a particular quality of service requirement.

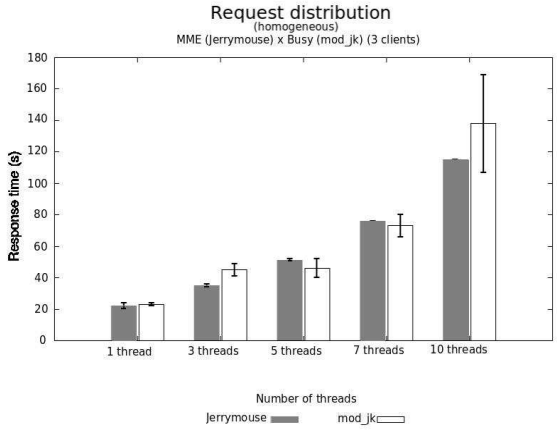

Figure 6. Results for the homogeneous platform and ***mod_jk busy*** policy vs. ***Jerrymouse policy based on memory and processor usage*** executing the financial transaction authorization service.

Fig. 8 shows the results for the heterogeneous platform, but now using the mod_jk busy policy and a Jerrymouse policy based on memory and processor usage. In this experiment, Jerrymouse demonstrated a performance statistically equivalent to mod_jk. This result is similar to the result obtained for the homogeneous platform shown in Fig. 6.

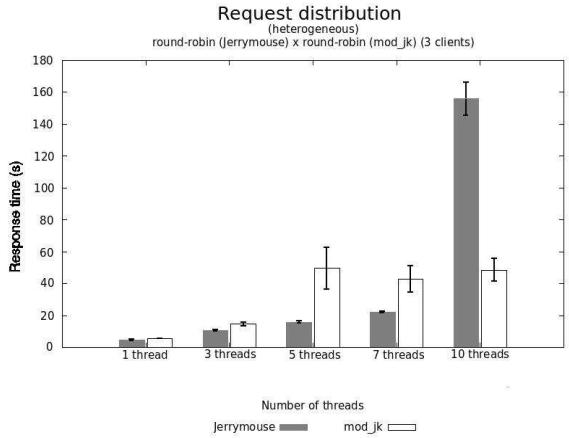

Figure 7. Results for the heterogeneous platform and policies ***round-robin*** executing the financial transaction authorization service.

Fig. 9 shows the results for the heterogeneous platform and weighted round-robin policies for Jerrymouse and mod_jk. In this experiment Jerrymouse also produced a higher performance when compared to mod_jk.

The results obtained for the pattern recognition service show that Jerrymouse produces similar performance gains to those reported above for the financial transaction authorization service.

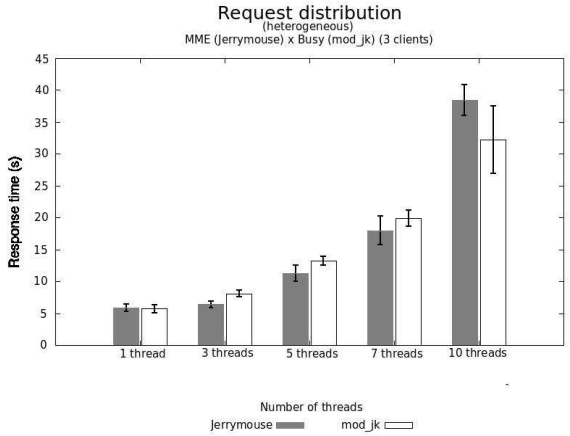

Figure 8. Results for the heterogeneous platform and ***mod_jk*** busy policy vs. ***Jerrymouse policy based on memory and processor usage***, executing the financial transaction authorization service.

The experiments conducted with the GreenPolicy were performed on a homogeneous platform with two servers, with one of them receiving all requests and the other one normally turned off. The second server was activated when the load index of the first server reached a defined threshold. The scenario used was the financial transaction authorization service with 1, 3, 7, 15 and 31 concurrent threads.

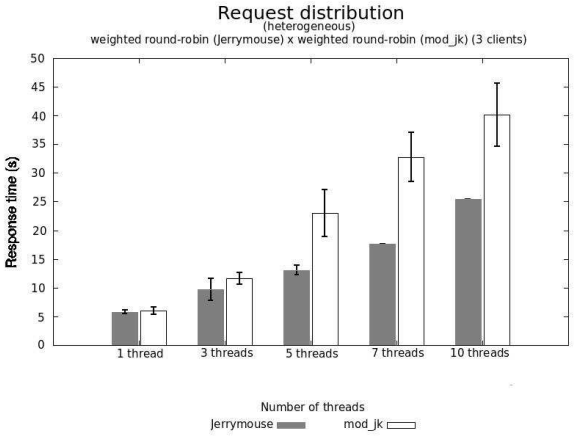

Figure 9. Results for the heterogeneous platform and policies ***weighted round-robin*** for ***Jerrymouse*** and ***mod_jk***, when executing the financial transaction authorization service.

Fig. 10 shows the client response time increasing as the load increases. The loads are identified with labels in the graph, which shows a response time peak close to 5700 requests. This represents the moment that the second server

was activated and started to serve requests. This peak occurs owing to the loading of the second provider and decreases shortly as soon this server starts to respond. In this experiment was saved 36.31% of energy.

Fig. 11 compares the same service execution, but now showing the behaviour with one server and two servers being activated on demand. It is possible to observe the difference in response time when using one or two nodes for the increased load. The transparency and flexibility of the architecture proposed in this paper simplify the use of features such as WOL.

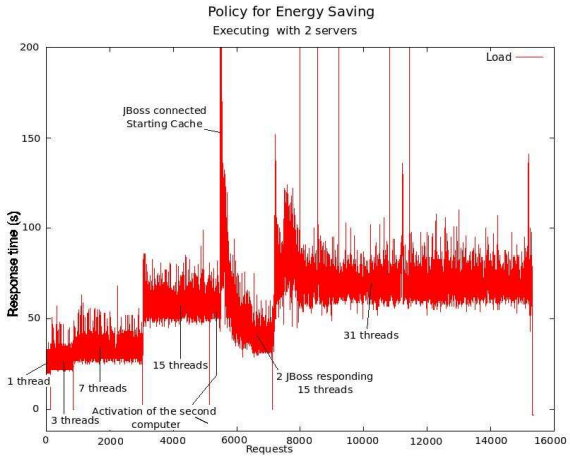

Figure 10. Results from GreenPolicy using two servers and executing the financial transaction authorization service.

V. CONCLUSIONS

This paper proposes a novel software architecture for the distribution of requests inside web service clusters. This architecture is flexible, dynamic and transparent both to the end-user and also web service developers. The architecture has been implemented in a prototype called Jerrymouse and has had its performance evaluated by experimental tests. The performance of Jerrymouse was evaluated by comparison with the distribution of requests made by the Apache HTTP Server mod_jk module for Apache Tomcat providers in remote nodes.

The results obtained with round-robin and weighted round-robin policies on both homogeneous and heterogeneous platforms have demonstrated that Jerrymouse delivers a lower overhead for distributing requests when compared to the existing mechanisms built into Apache HTTP Server.

The results for Jerrymouse policies gathering load indices from servers introduce an overhead because of the performance monitoring, but nevertheless provide a similar overall performance to that obtained directly from the Apache HTTP Server. Another relevant point is the requirement to determine the service demand, in order to design the best policies to support that service. In fact, the studies carried out in this initial work did not have the specific objective of analyzing the performance of possible distribution policies, but rather to compare the behaviour of both architecture and Jerrymouse when using default policies, already well-known and currently available in Apache HTTP Server.

The GreenPolicy developed for energy-saving also demonstrates that it is possible to offer the user a straightforward system which will reduce energy costs and optimize computing resources usage.

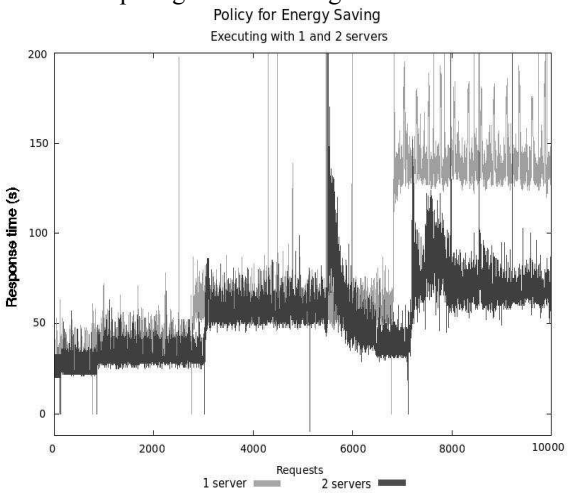

Figure 11. Results from Policy for Energy Saving using one and two servers when executing the financial transaction authorization service.

The Jerrymouse prototype uses existing software solutions, such as monitoring and component libraries. Its design provides scalability, performance and the flexibility to change components without suspending execution.

Our preliminary results show that the Jerrymouse has stable behaviour and is able to support future research works in this area.

Current research is being directed to investigate the relationship between load indices and overall web service performance. In addition, new distributing policies will be devised for Jerrymouse using load indices especially designed to reduce the demand imposed by clients on servers.

ACKNOWLEDGMENT

This work has been supported by FAPESP, a Brazilian funding agency, under processes: 07/57971-7, 08/00553-1 and 09/06670-2. The authors thank its financial support.

REFERENCES

[1] Liu, Dong; Deters, Ralph . Management of service-oriented systems . Service Oriented Computer Applications. Springer-Verlag London. (2008) , v.2 p.51-64.

[2] Red Hat. Jboss.org – community driven: JBoss Web. Available in: <http://www.jboss.org/jbossweb>. Last access: 14/Feb/2011

[3] Huhns, M.N.; Singh, M.P.; "Service-oriented computing: key concepts and principles,"IEEE Internet Computing, vol.9, no.1, pp. 75- 81, Jan-Feb 2005 doi: 10.1109/MIC.2005.21

[4] Mei-Ling Chiang, Chun-Hung Wu, Yi-Jiun Liao, Yu-Fen Chen, New Content-aware Request Distribution Policies in Web Clusters Providing Multiple Services, Proceeding, In.: SAC '09 Proceedings of the 2009 ACM Symposium on Applied Computing, 2009, pp 79-83.

[5] Brittain, Jason; Darwin, Ian F. Tomcat: The Definitive Guide. Sebastopol: O'Reilly. p. 336, 2003.

[6] Mysql AB. Mysql cluster. Available in: <http://www.mysql.com/products/database/cluster/>. Last access: 25/Feb/2011.

[7] Pgcluster top page. Available in: <http://pgcluster.projects.postgresql.org/>. Last access: 14/Jan/2011.

[8] Slony-lpg. Available in: <http://slony.info/>. Last access: 25/Feb/2011.

[9] Barbon, F.; Traverso, P.; Pistore, M.; and Trainotti M. Run-Time Monitoring of Instances and Classes of Web Service Compositions. In Proceedings of the IEEE International Conference on Web Services (ICWS '06). IEEE Computer Society, Washington, DC, USA, 63-71, 2006. DOI=10.1109/ICWS.2006.113

[10] Pallickara, S.L.; Plale, B.; Jensen, S.; Sun, Y.; , "Monitoring access to stateful resources in grid environments," IEEE International Conference on Services Computing, vol.1, no., pp. 343- 346 vol.1, 2005 doi: 10.1109/SCC.2005.68

[11] Sultan, F.; Srinivasan, K.; Iyer, D.; Iftode, L. Migratory TCP: connection migration for service continuity in the Internet. Proceedings 22nd International Conference on Distributed Computing Systems, 2002. p. 469-470.

[12] Massie, Matthew L.; Chun, Brent N.; Culler, David E. The ganglia distributed monitoring system: design, implementation, and experience. Parallel Computing, North-holland, v. 30, n. 7, p.817-840, 2004.

[13] Bin Liu, Jian Yang, Yu Zhao, Online prediction-based dynamic cluster configuration for energy conservation, In.: Proc of 2nd Int. Conference on Advanced Computer Control (ICACC), vol.4, pp.247-251, 2010.

[14] Luciano Bertini, Julius C.B. Leite, Daniel Mossé. Power and performance control of soft real-time web server clusters, Information Processing Letters 110 (2010) p.767–773.

[15] Elnozahy, E. N.; Kistler, Michael; Rajamony, Ramakrishnan. Energy-Efficient Server Clusters. Power-Aware Computer Systems. LNCS. Springer: Berlin, 2003. vol. 2325/2003. p. 179-197.

[16] Horvath, Tibor; Skadron, Kevin. Multi-mode energy management for multi-tier server clusters. Proceedings of the 17th international conference on Parallel architectures and compilation techniques. Toronto, 2008. p. 270-279.

[17] Lefurgy, C.; Rajamani, K.; Rawson, F.; Felter, W.; Kistler, M.; Keller, T. W. Energy management for commercial servers. Computer. IEEE Computer Society: Los Alamitos, 2003. vol. 36, ed. 12, p. 39-48.

[18] EU Energy Star. European Community Energy Star Programme for energy efficient office equipment. Available in: <http://www.eu-energystar.org/en/index.html>. Last acess: 25/Feb/2011.