ANALISYS OF A EVENT ORIENTED WEB SERVER AS A REVERSE PROXY WITH FLEXIBLE LOAD DISTRIBUTION

Luiz P. Petroski[1], João E. Bertotti[1], Jonathan de Matos[1]

[1] Universidade Estadual de Ponta Grossa (UEPG) – Ponta Grossa, PR – Brazil

petroskilp@hotmail.com, joao.e.bertotti@hotmail.com, jonathan@uepg.br

Abstract: According to research done by CGI (Internet Management Committee in Brazil) is indeed the growth of new internet users, as well as increased demand for use of servers that are responsible for letting the pages available. This steady growth can affect many services that are unprepared, causing the servers to work in a high rate of use by making them become slow and even stop working, causing users to slowness or lack of it. This meta-paper describes a possible solution for servers that can continue to provide web pages with good quality for a high demand of simultaneous HTTP requests. The solution is based on using the NGINX web server as a reverse proxy to receive new requests and with the help of a distributor that uses the Ganglia tool for gathering information about servers to balance the load among the available servers.

Keywords: Load distribuition; Event oriented web server; reverse proxy; High demand

1. INTRODUCTION

Web server is a computer program that accepts connections mainly using the HTTP protocol and its architecture fits in the client-server category. It is responsible by accepting clients requests, mostly web browsers and provide them with HTTP responses. The responses can include data as web pages in HTML, pictures or other media [Lingan, 2007]. The internet traffic and requests are difficult to predict, because the world wide coverage, so web servers have to be operating 24-7. The systems published in the web by web servers need, in some scenario, more than one server to endure the high demand of requests. In this cases not only multiple servers are needed but also an efficient way to spread or distribute requests to this servers. Whether the request distribution is not efficient and aware of the characteristics of the servers, it may result in bad servers utilization.

The traditional web server model is based in threads, for each connection a new thread is allocated. In the past, other techniques such as process allocation (CGI) were used. Of course not all new requests receive an exclusive thread, they can be queued and after being processed by a free thread. The use of multiple threads can produce a large memory footprint and threads may stay occupied with only one client connection that can be delayed by a busy resource. To increase the number of requests processed simultaneously by a server, administrators generally increase the number of threads, increasing almost linearly the amount of operating systems resources and memory. This scenario has a connection with the well know “C10K problem” [Nginx, 2014]. Generally there is some limitations on resources that causes thread oriented servers to accept the maximum of 10K connections simultaneously. Some techniques as triggers and epoll help to solve this problem, thus increasing the number of simultaneously connections.

NGINX is an event oriented web server instead of thread based, despite it still has some threads to processes events. It was also designed to be a reverse proxy. The event based characteristics helps it to allocate less memory to handle each new connection, providing a low memory footprint. Its operation creates workers, that are threads used to receive requests like a thread oriented solution. These workers receive new connections and use the resources of the operation system to create handlers with pointers to functions that will be executed when the handler descriptors suffers a change in their state. This operation makes NGINX perform in an asynchronous way, preventing a thread to block until the processing procedure ends for each connection. A single thread can be responsible to process multiples request-response clients simultaneously, i.e receive another request while waiting the response to another client.

Great part of web content is static, consisting in text, image and video, but there is also dynamic content originated of forms filled by users in several ways, resulting in pages generated by scripts. The end user generally does not know what generated a page and if it is processed with input data. The server has to know what kind of content it will deliver to end user. This is generally specified in its configuration files and is associated with file extensions. Static content is delivered, transferred, directly to the user. Dynamic content is processed by an application, sometimes using an interface called CGI (Common Gateway Interface) to call a new processes of a script interpreter. This technology is used to make the web server language independent, making no difference in what interpreter it will call [Kabir, 2002]. NGINX web server is an efficient static content server, but is not able to handle dynamic content. Dynamic content is passed forward by NGINX to another web server, like Apache HTTP Server, able to handle the request. This scenario is called reverse proxy.

The proxy role is to contact a remote server acting like the client and get a response to the client's request. The response is handled back to the client, but can also be saved in the proxy server to answer faster a request from another client without using internet connection again to a server with static content. Proxies can even be used to limit client's access to the internet, for example, establishing bandwidth thresholds, authorization policies and access free timetable. It is worth noting that proxies can have lot of benefits when dealing with static content, because of the absence of internet latency in sequential access access to the same content by different clients. A reverse proxy acts on server side, sitting in front of web servers, providing static content and passing dynamic content forward to specialized servers [Kabir, 2002]. The advantage can be the memory cache of the proxy and keep apart the specialized static server and dynamic ones. Other benefit is the distribution of requests to multiple dynamic servers by the static one.

A reverse proxy is generally placed as the first step to a request when it arrives at the servers structure. As aforementioned, it is possible to have many servers to forward requests that cannot be handled by the reverse proxy, so there should be an “intelligent” way to decide which server will be chosen. This process is called load balancing or request distribution.

Load balancing is the ability to make lot of servers provide same services to the user to improve its experience. Some of the benefits of load balancing are: scalability, availability, manageability and security. The scalability provides ways to an application grown, distributing new requests among servers. There is a limit of connections servers can receive, the scalability is responsible for making this limit grow. The ability of a service stay most part of the time accessible to the users through the internet is the availability. Manageability is the characteristic of a service that allows it to be moved

from one server to another without users realize that. At last, security can be improved by the load balancing using mechanisms in the policies to prevent not allowed accesses, like DDOS or brute force attacks, for example. [K. Gilly et. al.]

Some web servers have native load balancing. Nginx has three different load balancing methods or also called policies. The round-robin method dispatch every new request to a different server, when all servers receives requests, next requests are sent to those servers in the same sequence. This implies in an equal number of requests to all servers. Another method available in Nginx, is the Least-Connect which distributes a new request to the server that has least number of pending requests. The IP-Hash policy uses a hash table to discover to which server a request should be send. The key of this hash table is the client's IP address, if a key with the IP address is not found, a round-robin it adopts a round-robin policy. Once a request is associated to a server when it first comes to the front-end, the decision of the destiny is stored in the hash table so that in next requests from this client, the request takes the same way. This policy is useful when applications are stateful. The default policy of Nginx is the round-robin [Nginx, 2014]. The Apache Web Server also has three different policies, they are: by requests, by traffic and by business. They use different metrics to the decision, the first one counts the requests forward to each back-end, the second one uses the traffic to each back-end and the third one sends requests to the servers with least pending or that are processing least number of requests [Apache, 2014].

[Matos, 2009] developed a distributor that allows dynamic changes in policies, with no need of stopping and reconfiguring the chosen web server. The policies used in this proposal can be created by the web server administrator, as an example, he can create policies to control energy efficient load distribution [Faiçal et al, 2011]. The policy creation is facilitated by the monitoring resources available in the distributor. The monitoring used in the distributor is also flexible and transparent to the policy creator, so he has no need to know what monitoring service the distributor is accessing, he only knows that the metrics are available. The default monitoring system is Ganglia [Massie, 2009]. Some of the metrics available to the policies are I/O capabilities and load, memories amounts, processor status, server temperature, OS counters (like context switches, interrupts). This metrics can make easier the policy creation, besides new metrics can easily be added to the system using Ganglia's resources.

The IP-hash policy of Nginx can have a problem when multiple Nginx front-ends are used. Each of the front-ends have an internal hash table and clients can reach the server farm either to one or another front-end. This scenario implies in not right distribution of stateful requests. Other types of request distribution in Nginx are even worse to stateful applications. Another flaw in IP-hash is the origin of the key being the IP address of the client. IP address are affected by proxies and NAT client's structures, implying in a series of client's leaving their networks with the same IP address. The session information existing in HTTP headers can generate a better effect in stateful request distribution.

Based on these facts this work proposes the use of Nginx web server together with the distributor proposed in [Matos, 2009]. The main idea is to use a high demand reverse proxy distributing requests in a flexible (with the possibility of creating new customizable policies) and dynamic way.

2. MATERIALS AND METHODS

The development of this work used the Nginx web server in its version 1.4.1, the Apache web server version 2.2.22. For application testing purposes it was used the

version 5.5.38-0 of MySQL database to make possible the OpenCart 2.0 installation and execution. Nginx web server was compiled with GCC version 4.7.2. Requests were executed using curl version 7.26.0. The OS used in the tests was Debian GNI/Linux 7.6.

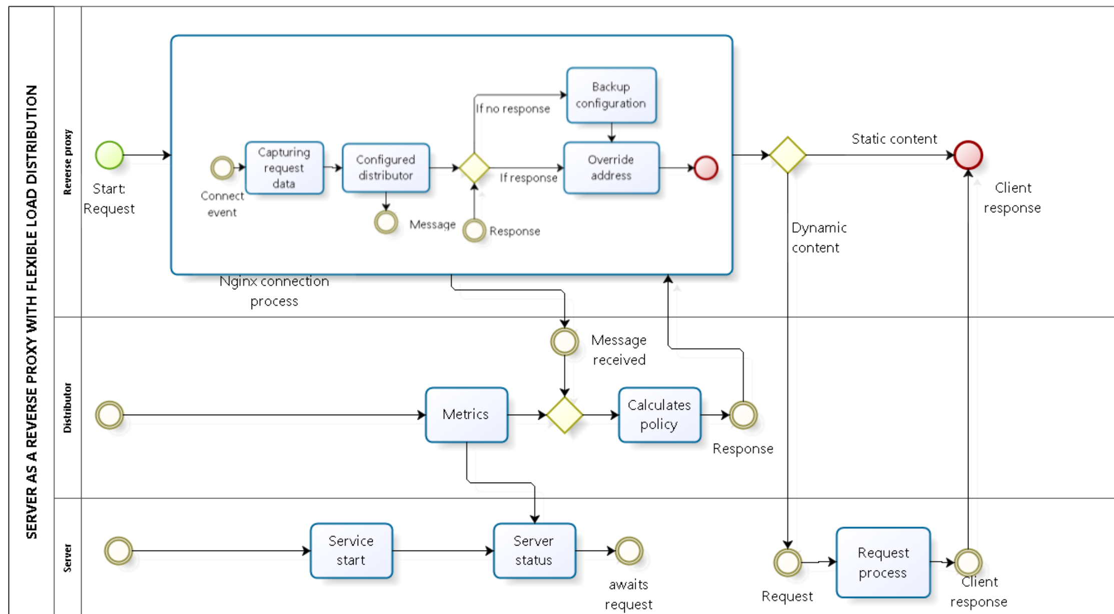

Figure 1: METHODS

Observing Figure 1, the main modification in the whole request processing is the connection processing in Nginx, which implements the request distribution.

There is a structure in Nginx configuration responsible to set the forwarding request procedure to other servers. This structure was used to set what kind of content the server is providing (static or dynamic) in a request, assuming to it the static content and forwarding dynamic ones. Some PHP pages of OpenCart were the tested dynamic content for the web server structure. The dynamic content, was processed by the Apache web server and its extensions.

The Nginx configuration file allows to set in a static way which servers will receive a forwarded request, but in the free version, this configuration can not be changed without a configuration reload. Using this same configuration, the Nginx code was changed to read the destination set in the configuration file and when a new request comes it is not forwarded to specified host, but this address is contacted to discover the forwarding destiny to the request. The address contacted is from the distributor, that answers back an address to which the request is finally forwarded.

Nginx source code version 1.4.1 was reverse engineered to reveal where the changes were needed. As an event oriented program, there were some difficulties in determine the sequence of functions execution. Some new functions were created and some existing ones were changed. After changes, the code was compiled using GCC.

When a new request arrives to Nginx, a communication is established with the distributor. The address of the distributor is the one provided in configuration file. The major change on Nginx was in the function ngx_http_upstream_connect, where the connect function received a changed parameter with the distributors answer. No bigger changes were required, because the event oriented concept. The message sent to the distributor carries the name of the policy which the distributor should execute, the client IP address and the HTTP header of the request. The message is delivered using UDP,

but distributor can easily support other protocols, even ones like ZeroMQ [Hintjens, 2013]. Once the distributor receives the message, it executes a policy and discover the destiny address to the request. The policy result is delivered to Nginx that overwrites the address field in the ngx_http_upstream_t structure. From this point on, the Nginx work continues as if there are no changes in the code.

The tests used an application to simulate commonly found in the real world. Commonly static contents are images, texts, css, javascript files and dynamic content are scripts the requires server side processing, sessions and database queries. The chosen test application was the open source Opencart, which implements an e-commerce application with product listing, navigation menu, buying items, user sessions. Some of the data come from a MySQL database. These characteristics are present in much of web sites.

The test environment used an Apache web server [Apache, 2014], installed for local access using 8080 port acting like the back-end. This Apache web server provided the Opencart application. The Nginx web server was used as a reverse proxy using 80 port acting like a front-end.

The cURL software was used to create successive requests and measure the response time for the requests.

```
1   #!/bin/bash
2   CURL="/usr/bin/curl"
3   URL="$1"
4   NUM="$2"
5   echo "Host: $URL Quantity: $NUM"
6   echo "Time Total "
7   COUNTER=0
8   while [ $COUNTER -lt $NUM ]; do
9       echo '$CURL -o /dev/null -s -w %{time_total} $URL'
10      let COUNTER=COUNTER+1
11  done
```

Figure 2: Script for measure the response time

Three tests scenarios were created. The first one tested the response time for the Apache web server directly, without a reverse proxy. The second scenario used Nginx as a reverse proxy with its original code. The third one was similar to the second, with the difference of the modified code using the distributor to select which back-end to use for each request.

The distributor in the tests were replaced by a simulator, which uses UDP as the distributor. This decision was made to analyze only the modification in Nginx code, not the policies and metrics. This work does not contemplate the distributor optimization or policies development. The bigger difference presented in the configuration of this work is the network communication. Part of the communication time will be ignored, the network traffic time, as it would take much more effect in the tests scenarios if the Apache web server as back-end had to be accessed in a remote server as the distributor communication.

Event oriented web server as a reverse proxy

Nginx is a front-end server, receiving all the connections and forwards them to other servers.

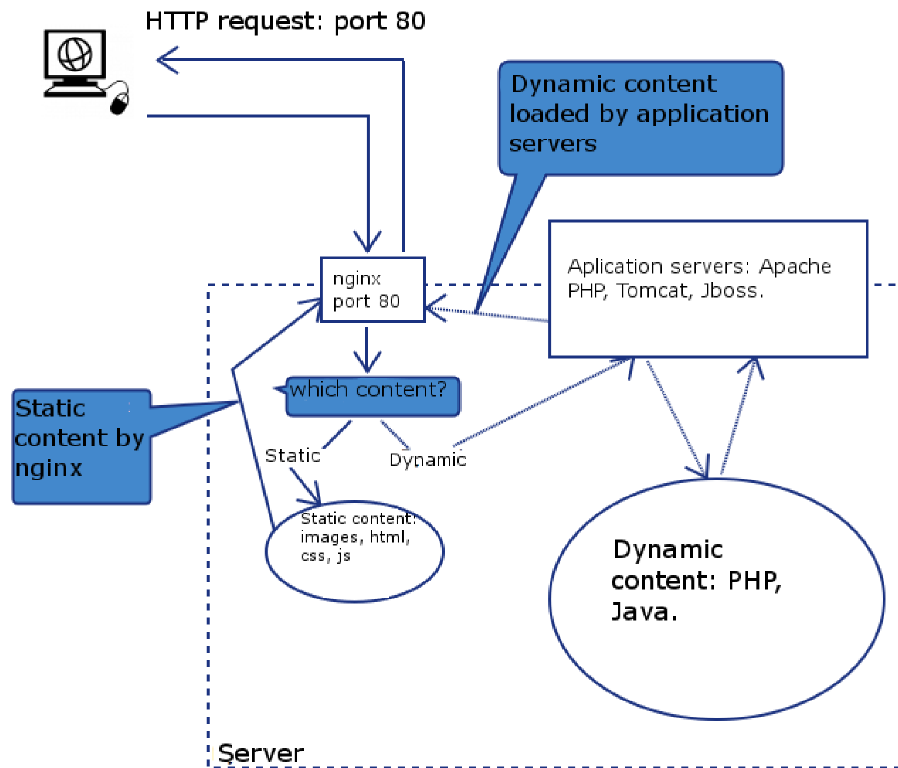

Figure 3: Nginx server

The standard Nginx configuration can be seen in the Figure 3, where Nginx listen to port 80 and receive all HTTP connections. It filters incoming requests by their content, whether the content is static, Nginx itself has the ability to handle the requests with a performance greater than Apache, for example, because of the low memory footprint. If the content is dynamic, the request is forwarded to a specialized back-end server. The client does not contact directly the back-end server, it continues to contact the front-end.

The configuration of Nginx is based on a file generally called nginx.conf, Figure 4 shows an example of the configuration file.

```
1:  user    www www; ## Default: nobody
2:  worker_processes 5; ## Default: 1
3:  error_log logs/error.log;
4:  pid     logs/nginx.pid;
5:  worker_rlimit_nofile 8192;
6:  events {
7:   worker_connections 4096; ## Default: 1024
8:  }
9:  http {
10:   include  conf/mime.types;
11:   include  /etc/nginx/proxy.conf;
12:   include  /etc/nginx/fastcgi.conf;
13:   index  index.html index.htm index.php;
14:   default_type application/octet-stream;
15:   log_format  main '$remote_addr - $remote_user [$time_local] $status '
16:    '"$request" $body_bytes_sent "$http_referer" '
17:    '"$http_user_agent" "$http_x_forwarded_for"';
18:   access_log  logs/access.log main;
19:   sendfile   on;
20:   tcp_nopush  on;
21:   server_names_hash_bucket_size 128; # this seems to be required for some vhosts
22:   server { # php/fastcgi
23:    listen    80;
24:    server_name domain1.com www.domain1.com;
25:    access_log  logs/domain1.access.log main;
26:    root     html;
27:    location ~ \.php$ {
28:     fastcgi_pass  127.0.0.1:1025;
29:    }
30:   }
31:   server { # simple reverse-proxy
32:    listen    80;
33:    server_name domain2.com www.domain2.com;
34:    access_log  logs/domain2.access.log main;
35:    # serve static files
36:    location ~ ^/(images|javascript|js|css|flash|media|static)/ {
37:     root  /var/www/virtual/big.server.com/htdocs;
38:     expires 30d;
39:    }
40:    # pass requests for dynamic content to rails/turbogears/zope, et al
41:    location / {
42:     proxy_pass   http://127.0.0.1:8080;
43:    }
44:   }
45:  }
```

Figure 4: Nginx configuration

Figure 4 shows how the configuration to answer HTTP requests works. In the line 31, there is a server configuration, where it is specified the server that handles requests in a given port and address (lines 32 and 33). The line 36 shows the location configuration where filters (based on regular expressions) are defined. The location specifies the directory where data can be found or other action needed. One of the actions can be the proxy_pass (line 42) that tells Nginx to behave like a reverse proxy. This example configuration shows that Nginx will forward all requests to a dynamic capable back-end. As the line 36 for static content is placed before the configuration for dynamic content, the first one make Nginx answer directly any content with the extensions that indicates they are static, so the last one is less specific than the first.

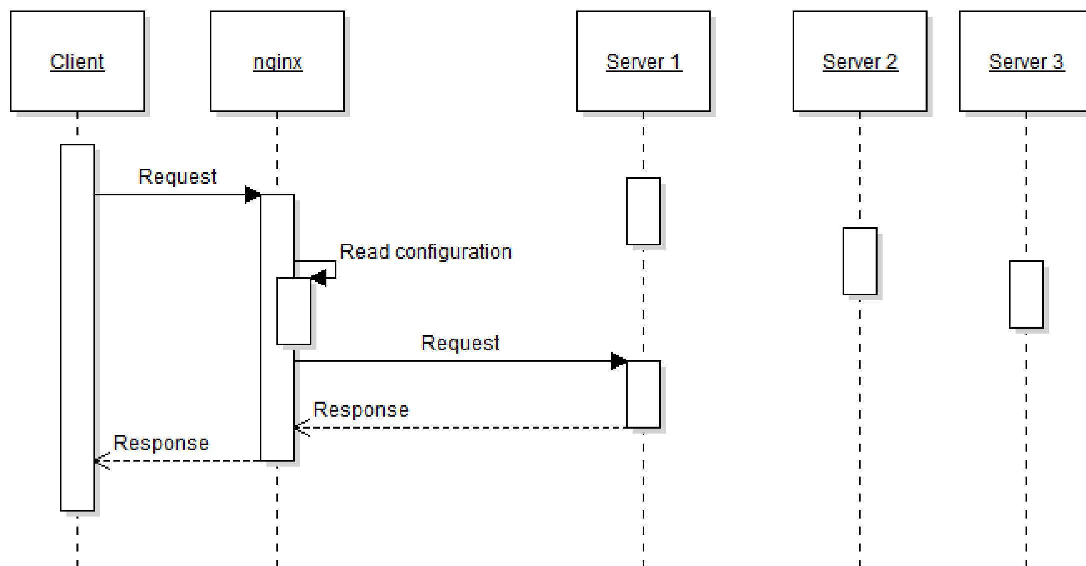

Figure 5: Original Nginx

Figure 5 indicates the mechanism of the original Nginx, where once a request come to the server, Nginx look at the configurations to find which back-end server will respond to the request. The next step is to forward the request and wait to the answer. Once it has the answer, it returns the last one back to the client.

Flexible load distribution

The first modification in Nginx was in the configuration file, where it continues to respond static requests directly and the dynamic requests are sent to another server.

The structure remains the same to the modificated Nginx, using the server to the proxy_pass. The difference is the upstream session where the back-end server is specified. The modified code looks for a string with the word "distribuidor.[policy]". In this case, the "[policy]" word will be part of the query to the distributor to inform the last one what policy it should execute to return the answer to Nginx. This word can be changed to whatever is the name of the policies existing in the distributor. In the upstream session containing the "distribuidor" word there is also a server reserved word that on the original version indicates the back-end server which will receive the request. In the modified version the configuration remains the same, but it indicates the IP address and the port of the distributor.

```
upstream distribuidor.roundrobin {
        server 127.0.0.1:10010;

}
```

Figure 6: Upstream

Figure 6 illustrates a configuration to the new version where the distributor can be contacted in the address 127.0.0.1 (localhost) and the port 10010. The reserved word "distribuidor" in the upstream session activates the modification, otherwise Nginx will behave as usual, forwarding requests to this upstream directly skipping the distributor query. In the example the policy that will handle the decision in the distributor will be the roundrobin. This word is not a reserved word and has to be the same in the distributor configuration. In the distributor, the policy name is used to find the function pointer in a hash table to execute the policy. This upstream configuration will be used in

the filter session that tells Nginx what to do with a content. Figure 5 exemplifies this configuration.

```
location / {
    proxy_pass http://distribuidor.roundrobin;
}
```

Figure 7: Foward request

In the configuration example shown in Figure 7, all requests to all types of content will be sent to a back-end server specified in this case as “distribuidor.reoundrobin” the same defined previously in upstream configuration in Figure 6. To forward only dynamic content to the back-end indicated by the distributor, the content must be filtered by the regex closer to the “location” word or it is necessary to filter static content previously in the configuration file, this is a location session right before this example location. This last described situation can be seen in Figure 4.

When a request is filtered to an upstream indicating the distributor, the last one will not receive the request by itself, Nginx will query the distributor by which back-end is the destination of the request. This query carries the address and port of the client responsible by the request, the policy name came from the configuration file and the HTTP header. The last one has much information about the request like session name, destiny location or even a x-forwarded-for entry.

Nginx code was modified to obtain right before the connection event data like the client's address and port, policy name and the HTTP header. This data was easily found in request and configuration structures, thanks to its organization. The logic in the code change was to use the existing address and port in the upstream structure to create a UDP communication with the distributor and with its response replace the upstream address and let Nginx work normally from this point on.

The protocol chosen for the communication between Nginx and the distributor was the UDP for its lightness and to prevent lots of connection being created and terminated inside this small part of code. Connections could be more efficiently used changing some global structures of Nginx. Other communications types could be used like ZeroMQ. The distributor receives the previously specified data and returns with a character string containing the address and port of the back-end. If the distributor fails to return the string, Nginx will retry two more times to contact it and in the case of total failure the backup back-end will receive the forwarded request. This backup back-end exists in configuration file and can be freely configured by the administrator. In failure scenario it is used by the modified version as the original.

When the back-end information returns from the distributor, Nginx overwrites the upstream address and forwards the request. Thus, the request follows to the better back-end elected by the policy chosen, that can be the least CPU busy, least I/O busy and so on. The stateful requests are handled by the policy in the distributor, because it has access to incoming information like session ID or client's address.

The working mechanisms of modified Nginx can be seen in Figure 8.

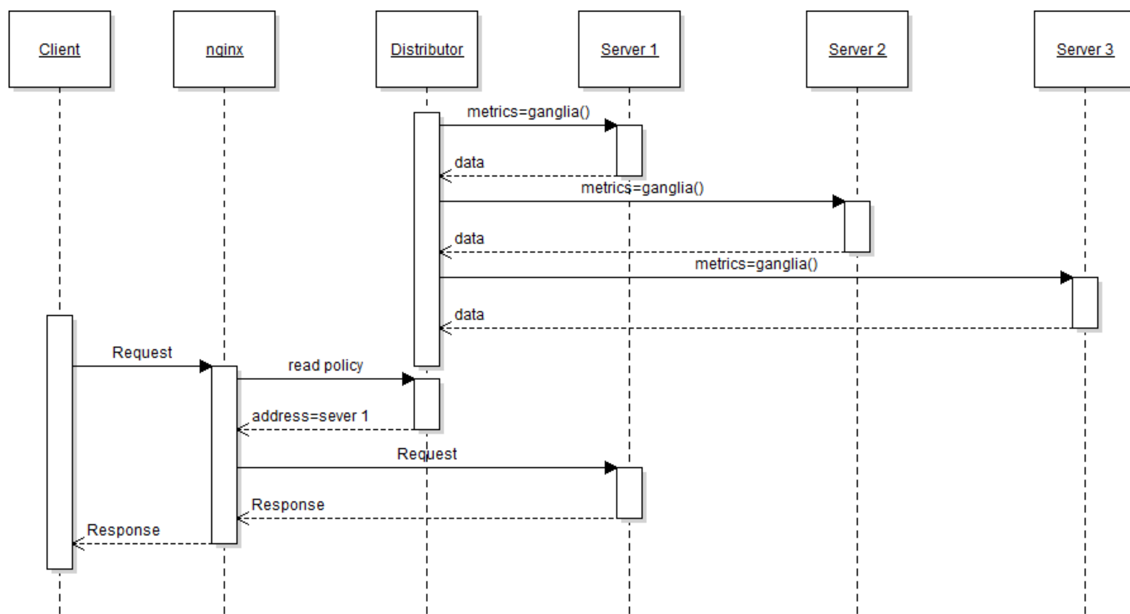

Figure 8: Modified Nginx

Figure 8 also illustrates that the distributor obtains metrics from back-ends. This metrics are constantly updated and stored in a hash table due to the multithread feature of the distributor. When a query to the distributor comes, it will not wait by metrics from the back-ends for the policy evaluation. The policy consults a hash table looking for metrics, this causes the policy to behave as non-blocking in relation to the metrics.

4. RESULTS

Nginx shown a good behavior during the tests, allowing to accomplish all the proposed objectives with a low interference in the response time when used as a reverse proxy. Table 1 lists the response time for 3 test sequences. Each test scenario was executed three thousand times for statistics purposes. The response times presented in Table 1 are the average of the tests and the standard deviation is also presented. Three tests scenarios are the ones described in the methodology. In all the scenarios, requests are processed by the same application server. The difference is the way they follow until reach to it, directly the application server in the first, through original Nginx in the second and through modified Nginx in the third.

	Directly application server (s)	Original Nginx (s)	Modified Nginx (s)
Average	0,0785	0,0802	0,0826
Standard deviation	0,0077	0,0106	0,0075

Table 1: Reponse time

5. DISCUSSION

In the analysis of the tests, it is possible to conclude that there is a small difference between the reponse times of first, second and third scenario. The application used has a big response time comparing to the overhead caused by the web server by itself. In other possible scenarios, there would be the network latency and transmission time, that was ignored in this tests because they are based on local interface only. The network times

would cause a bigger difference between the first and the second scenario. The difference between the second and third scenario would be hidden by the other bigger difference. This difference could be greater if the distributor was placed in another host, needing network communication. Even in a network scenario, the response time for the end user would be dominated by the external network access, between the end user and the web server (WAN and end user networks tends to be slowly than the LAN of servers). Another time that would make the differences smaller are the application itself, processing the scripts and consulting databases.

Acesso Direto	Nginx original	Nginx modificado
-	+2%	+5%

Table 2: Comparison between the scenarios

The differences can be better shown in a percentage comparison. Table 2 shows this differences. The difference of 1% in original Nginx scenario shows it causes low overhead as reverse proxy. The 3% difference in modified Nginx shows the connection between Nginx and distributor causes an overhead, but can produce performance improvements using more efficient load distribution, sending requests to more suitable back-ends and complying session aware applications

6. CONCLUSION

The web server optimization for HTTP requests is a promising field, because there is a growing need in process faster an increasing amount of requests as the web applications have shown a great importance in last years. In this context, some solutions to improve performance of server allocation for requests were analyzed.

Considering the obtained results, the impact of modifications were verified. Comparing the losses caused by the modifications, they can be acceptable with the advantage of the flexible and dynamic load balancing.

The implementation of Nginx as reverse proxy overcomes the performance loss given the benefits of centralized decision, efficient static content providing and the policy development for load balancing.

One point not treated in this work was the distributor structure, that can be not well suited for greater loads. This was not analyzed and can make Nginx indirectly have difficulties in processing high simultaneous load. As future works high amount of simultaneous request should be tested and maybe multithreaded implementation of distributor connectors could be proposed.

Taking into account the benefits described in this work, the modifications shown an improvement in the flexibility of load balancing using enabling customized policies development and more conscious load distribution.

REFERENCES

Abliz, M. (2011) "**Internet Denial of Service Attacks and Defense Mechanisms**", Univesity of Pittsburgh.

APACHE (2014). "**Apache Module mod_proxy_balancer**", http://httpd.apache.org/docs/ 2.2/mod/mod_proxy_balancer.html.

CURL (2014). "cURL How to use?", http://curl.haxx.se/docs/manpage.html.

Gilly, K., Juiz, C., Puigjaner R. (2010). "**An up-to-date survey in web load balancing**", Springer Science + Business Media, LLC 2010.

Faiçal, B., Souza, P., Santana, M., Santana, R, Matos, J. (2011) "**Jerrymouse: A Tool for a Flexible and Dynamic Distribution of Web Service Requests**", 2011 IEEE Conference on Services Computing.

Hintjens, Pieter. "**ZeroMQ: Messaging for Many Applications**". O'Reilly Media, Inc., 2013.

Kabir, Mohammed J. (2002). "**Apache Server 2 Bible**". New York, Hungry Minds.

Kegel, Dan (2014). "**The C10k problem**", http://www.kegel.com/c10k.html, February.

Kerrisk, Michael (2014). "**Linux Programmer's Manual**", http://man7.org/linux/man-pages/man7/epoll.7.html, July.

Knuth, D. E. "**The Art of Computer Programming: Fundamental Algorithms**". 3rd Edition. Addison-Wesley Professional, 1997. Vol. 1.

Lingan, James B. (2007) "**Web server**", http://whatis.techtarget.com/definition/Web-server, October.

Massie, M., Chun, B., Culler, D. (2004) "**The ganglia distributed monitoring system: design, implementation, and experience**".

Matos, Jonathan (2009) "**Distribuição de carga flexível e dinâmica para provedores de web services**", USP, São Carlos.

NGINX (2014). "**Beginner's Guide**", http://nginx.org/en/docs/beginners_guide.html.

OPENCART (2014). "**OpenCart Documentation**", http://docs.opencart.com.

Silberschatz, A., Galvin P. B., Gagne G. Operating System Concepts. 9th Edition. John Wiley & Sons, Inc, 2012. 944 p.