WEB SERVER: DIFFERENT IMPLEMENTATIONS AND ANALYSIS

Gianantonio Confalonieri

Andrea Curtoni

Luca Ferrari

Sergio Gentilini

Roberto Nour

Manuel Zippo

INTRODUCTION

Our project is based on different **implementations** and **performance analysis** of web servers.
In particular we implemented two different types of web servers that are **NGINX** and **APACHE** web servers and implemented with two different transport protocols: **HTTP/1.1** and **HTTP/2**. We made this because it is very interesting to observe the difference in terms of performance between these two web servers when they are running with different transport protocols. To make our analyses we use some Linux commands to test the behavior and the robustness of the web servers and the load time required to download a web page. All of our experiments are made on an HTML web page that we obtained from a template downloaded on the Internet and in which we added different web objects such as videos or images to make the page heavier. Our final objective is to make a general comparison.

WEB SERVERS

As we said, we used two different web servers: NGINX and APACHE.

TOOLS

We used Amazon Web Services, which is where we stored our web servers.
To measure the page load time we used the **developer tools** of two different browsers: **Mozilla Firefox** and **Google Chrome**. We considered the first one as our main tool because of all the functionalities that it offered, but we also considered the second one because it makes observing the concept of push promise simpler. About the commands for performance analysis, we used several Linux commands. They include:

- `ping`
- `traceroute`
- `h2load`
- `nghttp`

To test both the robustness and performance of the web servers we used:

- `ab`

To test the robustness by itself we used:

- HULK (Python program)

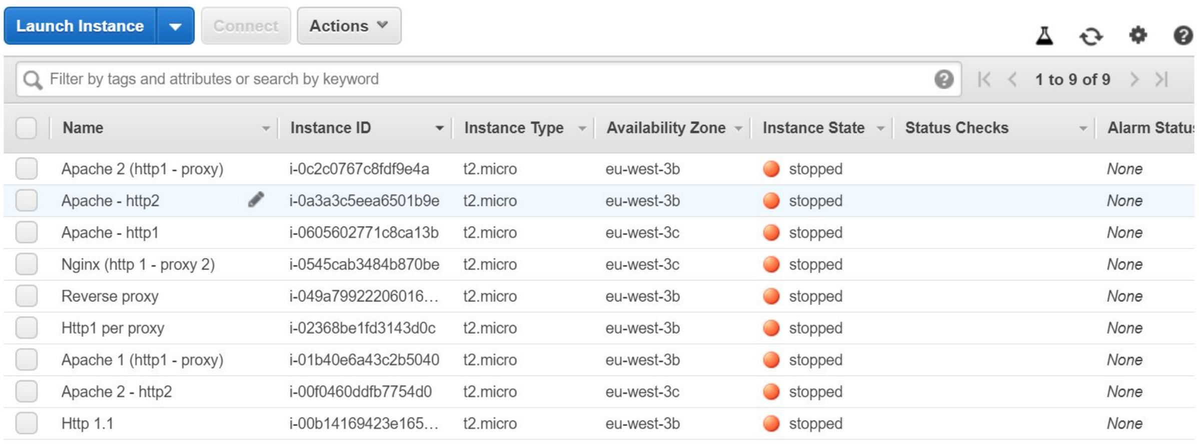
To make all these experiments we used the **Windows** and **Linux** operating systems. The latter was used to launch the command for all the experiments.

MEASUREMENT POINTS

All experiments were repeated several times and from different **measurement points** that depend on where each one of us lives, but in particular we can identify two principal locations: **Milan** and **Pavia**. The target is always Paris, where the virtual instances are stored.
It is also very important to say that each one of us has **different speed connections** depending on where we live (for example, in this case, in a metropolitan city as Milan or in a little country near Pavia), and whether we have a fiber connection or not. This is important to remember when we talk about the results, since sometimes they were very different in terms of time. We have taken this fact into account when drawing the final conclusions.

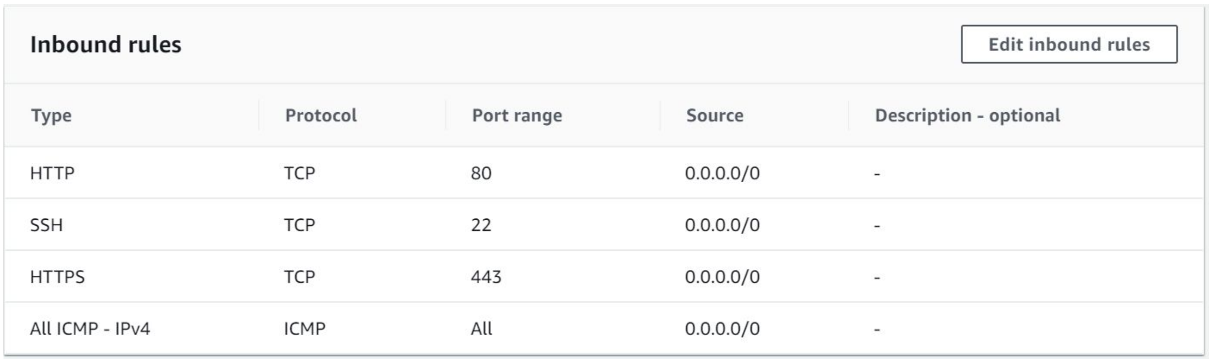
Amazon Web Services
To create our instances we used **Amazon Web Services**. It provides on-demand cloud computing platforms and APIs. One of these services is the Amazon Elastic Compute Cloud (**EC2**), which allows users to have at their disposal a virtual cluster of computers, available all the time, through the Internet, also providing scalability. We have chosen **Amazon EC2** because it provides a lot of the features that we have widely used:

- **Instances**, which are virtual computing environments. In the figure below we can see all our different instances.



Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Statu
Apache 2 (http1 - proxy)	i-0c2c0767c8fdf9e4a	t2.micro	eu-west-3b	● stopped		None
Apache - http2	i-0a3a3c5eea6501b9e	t2.micro	eu-west-3b	● stopped		None
Apache - http1	i-0605602771c8ca13b	t2.micro	eu-west-3c	● stopped		None
Nginx (http 1 - proxy 2)	i-0545cab3484b870be	t2.micro	eu-west-3c	● stopped		None
Reverse proxy	i-049a79922206016…	t2.micro	eu-west-3b	● stopped		None
Http1 per proxy	i-02368be1fd3143d0c	t2.micro	eu-west-3b	● stopped		None
Apache 1 (http1 - proxy)	i-01b40e6a43c2b5040	t2.micro	eu-west-3b	● stopped		None
Apache 2 - http2	i-00f0460ddfb7754d0	t2.micro	eu-west-3c	● stopped		None
Http 1.1	i-00b14169423e165…	t2.micro	eu-west-3b	● stopped		None

- **Images (AMIs)**, which group what we need for our servers. In particular, we did all the different implementations using Ubuntu LTS 18.04.
- **Various configurations of CPU**, **memory**, **storage**, and **networking capacity** for the instances, known as instance types. In particular, we used configurations with 1 virtual CPU, 1 GB of RAM and 8GB of SSD.
- **Key pairs**, used for authentication.
- **Regions**, physical locations in which you can store the instances. In our implementations we stored them in Paris.
- **Security groups**, in which we could organize some firewall rules (protocols, ports, source IP and more). In the figure below we can see the rules for our VMs.



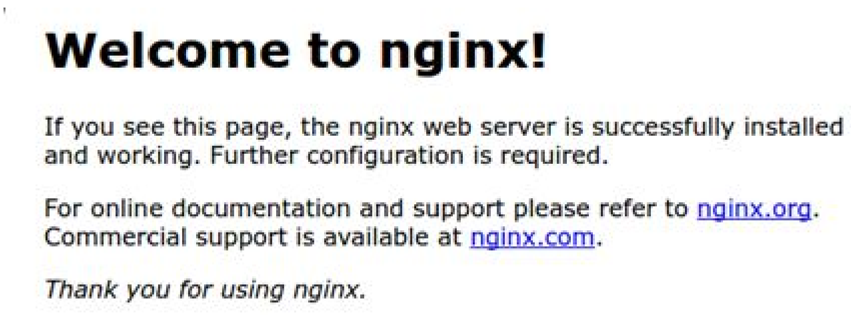
Type	Protocol	Port range	Source	Description - optional
HTTP	TCP	80	0.0.0.0/0	-
SSH	TCP	22	0.0.0.0/0	-
HTTPS	TCP	443	0.0.0.0/0	-
All ICMP - IPv4	ICMP	All	0.0.0.0/0	-

WEB SERVER IMPLEMENTATION - NGINX

Note: After creating the instance we accessed it through an SSH connection. First thing to do was to create a new user, with admin privileges. After that we also had to enable the SSH connection in the new user, with the key generated by AWS.

The installation of NGINX can be done with the command "`sudo apt install nginx`".

By only using this command the server is running! If we use the IP address of the machine we can see the default NGINX web page.



Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

The web server can be managed using simple commands which can start, stop and restart it.

Server blocks

An important NGINX feature to understand is **Server Blocks***.* It is very similar to the virtual hosting concept. In NGINX server blocks are used to encapsulate in one single server more than one domain, or in other words, we could have a physical server hosting multiple websites. All of them share the same IP address. In our case we hosted just one website in our web server. We put all the files related to our website in the directory */var/www/html*, the one used by default in NGINX.

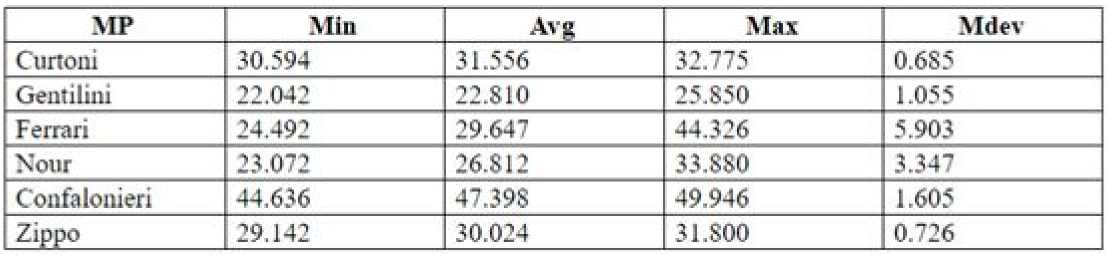
HTTP/1.1 implementation

For the first part of our analysis we used **HTTP/1.1**, with a web template downloaded from the Internet. We "upgraded" the website by adding 30 videos (6MB each), in order to obtain a heavy main page. In the image below we can see the main page's template.

HIGHWAY
WELCOME TO HIGHWAY
FREE CSS TEMPLATE by templatemo
Homepage with full-width image gallery

ping command

As first thing we tried to ping our web server. We did it to see the differences between the various measurements points, considering that the server is in Paris, looking at the **RTT**. We started using *10 packets*, each one of *200 bytes*. All the packets that we sent reached the server, and the RTTs were coherent with our connection speeds. For example, Gentilini has a good optic fiber connection and he had on average 22 ms as response time, while Confalonieri, with a slower connection, had 47 ms. No packets were lost during this test. In the table below we can see all the measurements (in *milliseconds*).

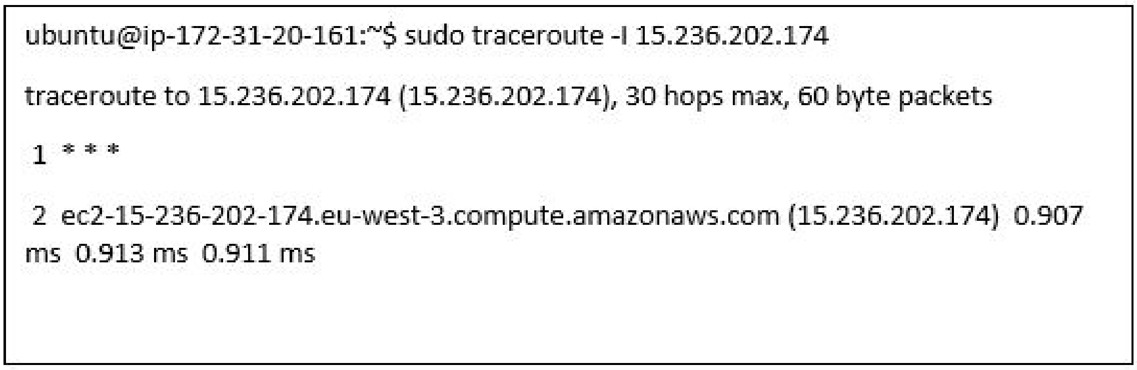


MP	Min	Avg	Max	Mdev
Curtoni	30.594	31.556	32.775	0.685
Gentilini	22.042	22.810	25.850	1.055
Ferrari	24.492	29.647	44.326	5.903
Nour	23.072	26.812	33.880	3.347
Confalonieri	44.636	47.398	49.946	1.605
Zippo	29.142	30.024	31.800	0.726

traceroute command

After we used the traceroute command. We wanted to see the **network topology**, and the various paths made by the packets. They were coherent, if the two measurement points were near, the path (and the relative number of hops) was more or less the same. We also tried to do the traceroute using a second virtual machine, instantiated in Paris. The path was coherent, there was only one hop.

Example, output from the near VM:

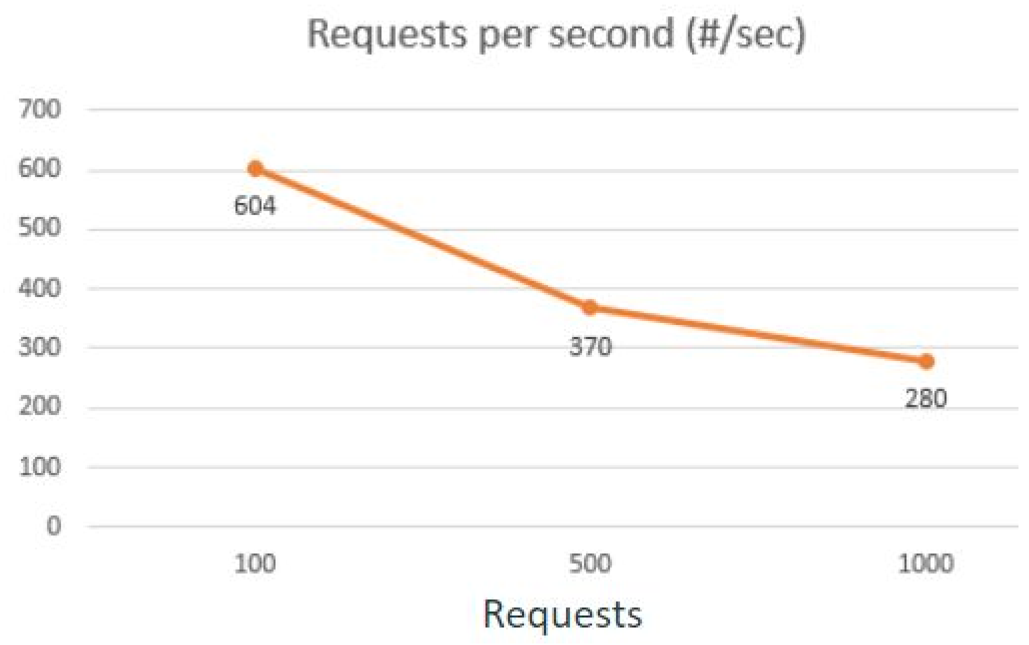


```
ubuntu@ip-172-31-20-161:~$ sudo traceroute -I 15.236.202.174

traceroute to 15.236.202.174 (15.236.202.174), 30 hops max, 60 byte packets

 1 * * *

 2  ec2-15-236-202-174.eu-west-3.compute.amazonaws.com (15.236.202.174)  0.907
ms  0.913 ms  0.911 ms
```

Apache benchmarking tool – *ab* command

We continued using the *ab* command. We modified the option "-n" (Number of requests to perform for the benchmarking session) and the "-c" (Number of multiple requests to perform at a time). In the image below we can see the results from one measurement point.

(The other results from other measurement points were so similar, in terms of the "shape" of the curve.)



As we can see from the image, increasing the number of requests, the number of requests per second goes down, because more time is needed to serve all of them.

PLT – browser developer tools

At the end we used the web developer tools. The PLT was obviously so high, in average was about 60 seconds. We can clearly see that, because we are using HTTP/1.1 the videos are download one at a time, not concurrently: this is a problem of HTTP/1.1.

HTTP/2 implementation

As we already said, Nginx is a powerful web server that offers a lot of capabilities in terms of configuration.
Since Nginx supports the HTTP/2 protocol, we decided to implement it to assess the server performance with that configuration.

SSL certificate

Even though HTTP/2 does not require encryption, since the most popular browsers stated that in the future they will only support HTTP/2 on HTTPS connections, and since we wanted to make the web experience as close to reality as possible, before implementing HTTP/2 we need to secure the server with HTTPS.
To do that we need a **SSL certificate**.
We could have used a self-signed certificate, but for the same reasons stated above we decided to get our certificate signed by a third party authority which was in this case **Let's Encrypt**.
To get the actual certificate we used the Certbot command line plugin that provides a simple interface to do that and automatically communicates with Let's Encrypt.
After launching the **Certbot** command all we need to do is:

1. Specify the server name (**virtual host**) on Nginx
2. Specify the domain name of the website
3. Done!

To get a proper certificate we purchased a domain for the website (***ediproject.online***) and we pointed it to the server IP.
After making sure that the server was using the HTTPS protocol and the domain was correctly pointing to the website IP, we configured Nginx to use HTTP/2.

Configuration

The configuration is quite simple and it's basically a matter of setting the correct **port** to listen for HTTP/2 traffic and setting the **ciphers** used by Nginx in the configuration file.

```
server {
        root /var/www/ediproject.online/html;
        index index.html index.htm index.nginx-debian.html;
        server_name ediproject.online www.ediproject.online;
        location / {
                try_files $uri $uri/ =404;
        }
    expires $expires;
    listen [::]:443 ssl http2 ipv6only=on;
    listen 443 ssl http2;
    ssl_certificate /etc/letsencrypt/live/ediproject.online/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/ediproject.online/privkey.pem;
    ssl_ciphers
EECDH+CHACHA20:EECDH+AES128:RSA+AES128:EECDH+AES256:RSA+AES256:EECDH+3DES:RSA+3DES
:!MD5;
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem;
}
```

Once the HTTP/2 server was up and running we did some performance testing using the tools showed before to see whether the HTTP/2 was actually performing better than HTTP/1.

NGINX: HTTP/1.1 and HTTP/2 Comparison

Website Performance and Analysis

First, a heavy webpage with 30 short videos was used as website, and the behaviour of the two NGINX servers was compared. In the case of HTTP/1.1 the videos must be downloaded **one at a time** and, while one is processed, the others are left pending and can't be watched by the user. In the end, it takes about 60 seconds to download the whole webpage with all its objects.
In the case of HTTP/2, however, the videos can be downloaded in **parallel** and the website takes less time to load. This happens because of certain HTTP/2 mechanisms, such as the possibility of using streams to transfer multiple objects at the same time.
The conclusion is that, for this website, HTTP/2 is the most user-friendly alternative, as it's faster than HTTP/1.1 and lets the user watch the videos even while they're being downloaded.

Performance Comparison

On the same website, ab was used to measure the performance on the NGINX web server in both the HTTP/1.1 and HTTP/2 cases. Several experiments were attempted, each with different numbers of concurrent requests sent to the server, and the behaviours were very similar.

HTTP/1.1	*HTTP/2*
Requests per second: 218.89 [#/sec] (mean)	Requests per second: 369.75 [#/sec] (mean)
Time per request: 4568.470 [ms] (mean	Time per request: 2704.542 [ms] (mean)
Time per request: 4.568 [ms]	Time per request: 2.705 [ms]
Transfer rate: 2628.20 [Kbytes/sec] received	Transfer rate: 4457.92 [Kbytes/sec] received

In this case, 1000 concurrent requests were sent with ab, and the result is that HTTP/2 can satisfy a larger number of requests in a smaller amount of time.

PLT Analysis

Another experiment was attempted: a lighter webpage with 100 small images was used, and then the **page load time** was measured with different browsers and from different measurement points.

	HTTP/1.1	*HTTP/2*
MP1	2.20s	1.22s
MP2	2.59s	2.03s
MP3	2.02s	1.71s
MP4	3.23s	2.27s
MP5	3.40s	2.99s
MP6	6.35s	5.17s

The times are averages over multiple measurements, for each measurement point. In all cases, HTTP/2 was **quicker** and the improvement is relevant.
An interesting observation is that the webpage used for this experiment causes a noticeable overhead because of the large number of images and required RTTs: this means that HTTP/2 can handle requests of this kind with ease, thanks to mechanisms such as streams.

Conclusion

Under these circumstances, **HTTP/2 is the preferred configuration** for an NGINX web server implementation.

HTTP/2 - SERVER PUSH

As we already mentioned, HTTP/2 comes with a lot of features to optimize how content is delivered by the server to the clients.
One of them is the **server push**, which provides a way for the server to send responses before requests are even made.
Multiple responses are triggered by a single request and in this way, it is possible to save RTTs.

Configuration

To implement it in Nginx we need to change the **http2_push** parameter in the site configuration file and specify the files that will be pushed when a certain resource is requested.

```
server {
    listen 443 ssl http2;
    ssl_certificate ssl/certificate.pem;
    ssl_certificate_key ssl/key.pem;
    root /var/www/html;

    # whenever a client requests demo.html, also push
    # /style.css, /image1.jpg and /image2.jpg
    location = /demo.html {
        http2_push /style.css;
        http2_push /image1.jpg;
        http2_push /image2.jpg;
    }
}
```

To make sure that the server was actually pushing the files we used two different tools: **Chrome web developer tools** and the **nghttp** command. In the Chrome web dev tools pushed objects are tagged with the term **PUSH** in the initiator tab, so we can easily verify the server push.

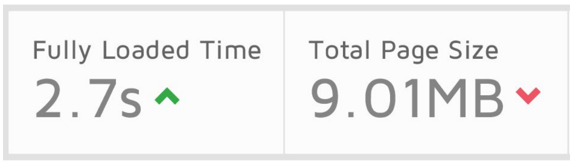
Experiments

As we did before, we carried out different experiments to evaluate the server performance and verify that the HTTP/2 push configuration was actually better than the simple HTTP/2 configuration.

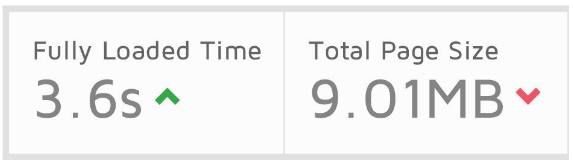
1. The first experiment was conducted on the **30-videos-website** presented before and comparing the PLT of the simple HTTP/2 configuration and the server push configuration we couldn't notice any significant difference.
   That was probably due to the fact that the videos download time was so big that any relevant improvement in terms of PLT was hidden by that.
2. To solve this problem, we modified the 30-videos-website by replacing the videos with **100 small images** and then re-did the measurements with both the non-push and the push configuration, making so that the server was **pushing all the images** when the **index.html** file was requested.
   Unexpectedly we found out that the non-push configuration was performing better than the push configuration.
   After thinking, discussing and researching about this issue, we concluded that pushing too many objects is **counter-productive** in terms of performance improvement because by doing this, the TCP connection underneath can get congested by the traffic generated with the pushed assets, resulting in higher PLTs.

3. As a final experiment, to verify our assumptions, we reduced the number of pushed images to 10 and that reduced the PLT by a solid **16%**.

Sample measurements taken with GTmetrix page speed analyzer.
(on the left the Push configuration, on the right the non-Push configuration)



Fully Loaded Time	Total Page Size
2.7s	9.01MB



Fully Loaded Time	Total Page Size
3.6s	9.01MB

These experiments taught us that HTTP/2 server push is a powerful feature, but to get the most out of it careful thinking about the number of objects to push and configuration fine tuning configuration are needed.

NGINX: Attacks and Robustness Tests

After measuring the performance of the web servers, a different kind of test was attempted: the robustness of the web server was checked with attacks performed through two tools. In particular, our robustness tests consisted in denial of service attacks issued through the ab command and the HULK Python program.

DDoS-like Attack through ab

With ab, a denial of service attack was delivered by attempting **1000 concurrent requests** from seven different measurement points, for a total of 7000 requests. Even though the server took more time than in the previous cases to handle all the requests (which is expected because of their large number), in the end they are all satisfied correctly. No request is lost and there are no major issues: NGINX is robust to this kind of attack.

HULK: HTTP Unbearable Load King

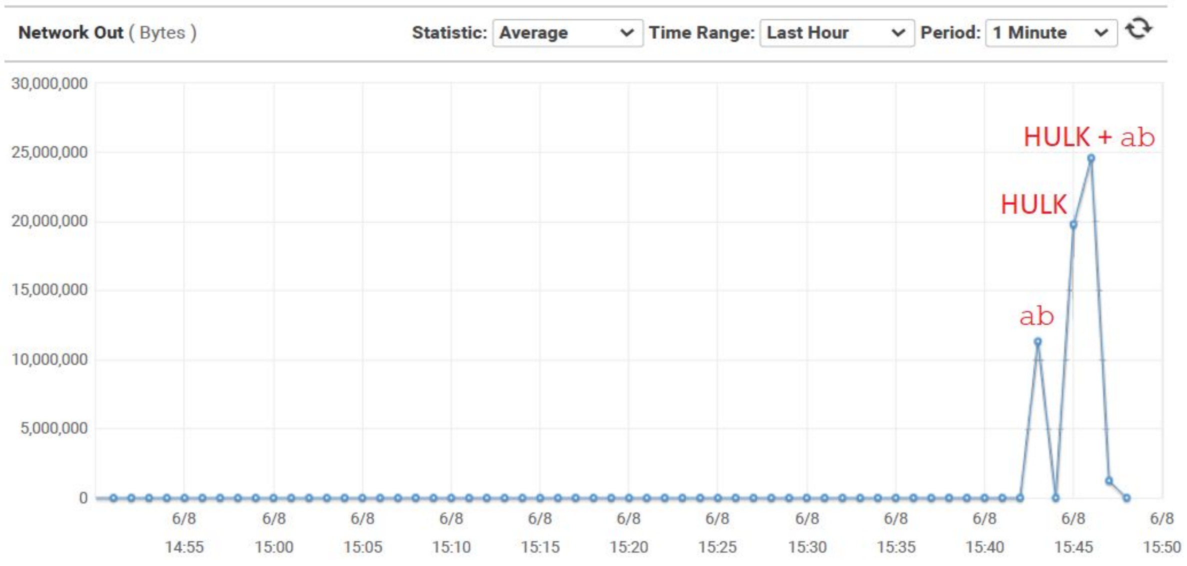
For the second DoS attack, the HULK program was used: it's a Python script whose peculiarity is being able to attack a web server by generating a unique pattern with every connection, making it hard to detect and avoiding prevention systems.
HULK takes advantage of five techniques:

- *Obfuscation of source client*, which consists in indicating a different `user-agent` each time a new connection is performed. The user agent is chosen from a list at random, so that it can't be easily detected.
- *Reference forgery*, which means that the `referer` header line is changed with every connection, again to make the attack harder to detect.
- *Stickiness*, which consists in keeping the connection active by indicating a `keep-alive` value that is chosen at random with every connection.
- *No-cache*, which means that HULK requests the whole web page with every connection and avoids cached content.
- *Unique transformation of URL*, which means that URLs specified in forged header lines are randomly generated by the program to evade pattern-detection systems.

DoS Attack through HULK

HULK was used to perform an attack towards the NGINX web server. The attack increases the loading times of the web page, but it remains reachable even under all the stress. Despite being subjected to a sophisticated attack, NGINX is **still available**, so it's safe to say that this implementation makes for a robust and reliable web server.

Output Graph



This graph shows the responses that the web server was able to send back when using the two tools. The first peak shows the impact of ab, while the second is caused by HULK and, five minutes later, by HULK and ab together. It reveals that HULK is much more powerful than ab, but, even then, the web server resists the DoS attack.

APACHE - INTRODUCTION AND WHY TO USE IT

Then we implemented other web services, on which we installed **Apache HTTP Server**, a free and open source web server software, developed and maintained by an open community of developers.

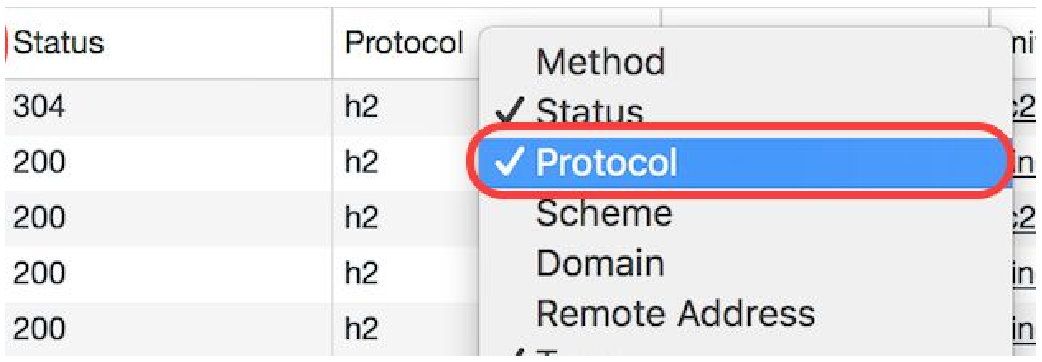
Even if NGINX performances are often better than Apache's, the latter has some characteristics which are preferred, for example:

- *Apache* supports **ALL Unix like** systems and **fully** supports Microsoft Windows, while *NGINX* support **almost all** Unix Like OS and **partially** supports Windows;
- *Apache* customization is **more flexible** since it adopted dynamic modules since longer time;

So, in many professional environments it is performed a combination of the two softwares: *NGINX* is used as a **reverse proxy**, running on the front-end; *Apache* is used as a **web server**, running on the back-end. In fact, we implemented also this type of infrastructure.

Apache HTTP/1.1 – HTTP/2

Implementation
That's not much different from the one we have seen for NGINX.
Brief recap: after the installation of the web server software, **HTTP/1.1** is already set as default protocol; then, to adopt **HTTP/2**, we needed to update some rules on the Ubuntu firewall and we loaded two modules required (ssl and HTTP2 modules); finally, we activated them and restarted the server software, then HTTP2 is running (we assessed it from the browser developer tools, by looking at the fields of column 'protocol'; and also by using the linux command curl, which carry also the information about the protocol used).



Status	Protocol
304	h2
200	h2
200	h2
200	h2
200	h2

Ping measurements

Even on these implementations we performed the ping command on 10 packets of size 200B, then of size 2000B, obtaining the following results:

MP	Mean RTT (200B) HTTP/1.1	Mean RTT (200B) HTTP/2	Mean RTT (2000B) HTTP/1.1	Mean RTT (2000B) HTTP/2
Sergio	23.030ms	22.819ms	28.192ms	27.573ms
Andrea	33.247ms	33.099ms	34.837ms	32.964ms
Luca	35.470ms	27.797ms	30.079ms	30.017ms
Gianantonio	46.237ms	46.242ms	65.673ms	65.789ms
Manuel	31.249ms	31.038ms	34.645ms	32.571ms
Roberto	29.422ms	28.096ms	31.247ms	25.232ms

We didn't get some really unexpected results: with pings using small sized packets we are measuring the RTT, which is **not very different** from the previous implementations, since the physical machines used are in the same location (Paris) and use the same network (we are still using machines offered by AWS); when increasing the size of the packets to 2000B, , we can see that there aren't relevant differences between the two HTTP protocols, since the ping command **operates over ICMP**, which is a lower protocol with respect to HTTP (so the version doesn't influence measurements). In conclusion, **things go as expected**, from what we know about the theory.

Concurrent requests analysis

Then we performed again the command 'ab', varying the number of concurrent requests in the same way we did before, using 100, 500 and 1000 concurrent requests.
Even this time, they're handled **without many problems** by both HTTP/1.1 and HTTP/2: the number of requests per seconds (rps) processed keeps being quite good, and the number of requests failed kept staying at 0.
However, when we tried to stress the service sending 1000 requests all together, from the different MPs, the two protocols had a **problematic behavior**, in different weights according to the protocol: by the web server using HTTP/1.1 a lot of requests failed (about 60%), from all the MPs, and when trying to get to the web page by the browser we got a response time very high (about 20 seconds); by the server using HTTP/2 instead, a low percentage of requests had failed (about 10%) in all MPs and we didn't get relevant delays when requesting the page.

DoS attack

Finally, we performed the DoS attack using HULK, and in both cases the web site was **no more reachable**: we got an ERR_CONNECTION_TIMED_OUT message. By the way, the server using HTTP/2 tolerated for more time this attack, but at a certain point it broke down too. So even in this case we could assess that HTTP/2 can handle bigger loads of data exchanges.

NGINX vs APACHE

Then, we made a **comparison** between some performances of the web server running on Apache and on NGINX (hosting the same web site).

PLT comparison

We measured the PLT using the **developer tools** of Mozilla and Chrome: we requested the page about 10 times (each time cleaning the cache and the cookies) for each MP, and averaged the results; let's take a look at the **comparison** between Apache and NGINX using HTTP/2 (Mozilla results):

MP	NGINX - mean PLT	APACHE - mean PLT
Manuel	1.20s	3.37s
Andrea	2.03s	2.28s
Luca	1.70s	4.23s
Sergio	2.27s	1.58s
Gianantonio	2.98s	4.30s
Roberto	3.17s	3.98s

As we expected, **NGINX performs better** than Apache; also, the times obtained are more stable on the different MPs (we could conclude the same with the machines using HTTP/1.1).

Concurrent requests comparison

About the handling of the multiple concurrent requests, in the case of *NGINX* we always got **all requests served** (using both HTTP/1.1 and HTTP/2) and we had almost **no delays** in the response of the web page; while, with *Apache*, we saw that some requests **failed** and the web page was accessible with **consistent delays**.

DoS attack robustness comparison

Finally, we assessed that *NGINX* is **more robust** to the DoS attack.
When performing the HULK command, even when using HTTP/1.1, the server was able to handle the huge amount of traffic and the webpage was always accessible; while with the Apache servers we saw that, sooner or later, we got an *ERR_CONNECTION_TIMED_OUT* message, which persisted even when the attack stopped (so the service was unavailable for a relevant amount of time).

So, we have been able to confirm that, from the point of view of PLT and handling of the requests, **NGINX has better performances**.

Load Balancer Implementation

Reverse Proxy and Load Balancer - Nginx Implementation

Last idea we decided to implement was an infrastructure based on a reverse proxy.

On the web (in particular on nginx official website), we had the possibility to discover that there is a difference between a reverse proxy and a similar infrastructure called load balancer. In detail:

- A **reverse proxy** accepts a request from a client and forwards it to a server that is able to fully satisfy it. The same server will send the response to the client as if it had been generated by the proxy server itself.
- A **load balancer,** on the other hand, distributes incoming client requests among a group of servers that return the response to the appropriate client.

We decided to adopt **nginx,** previously used by us as a web server but widely used as a reverse proxy and a load balancer too. In particular we decided to implement it in a load balancer configuration.

Setup of Infrastructure

First necessary step was to setup the necessary infrastructure.
In particular we decided to use three different virtual machines running Ubuntu Server 18.04:

- Two machines used as web servers, on which we installed Apache HTTP Server.
- One machine used as a load balancer, on which we installed nginx.

The two web servers machines had to contain the same version of the same website and, in order to compare performances to the previous experiments, we uploaded the website already used for previous experiments. The website under consideration consists in one hundred images integrated.

On the load balancer machine we had to setup the appropriate configuration to distribute the traffic through the two web servers. At this purpose it was necessary to modify the file *`/etc/nginx/sites-enabled/default`*.

```
location ediproject {
        server 15.236.158.115;
        server 15.188.207.167;
}

server {
    listen 80;
    server_name ediproject.online;

    location / {
      proxy_http_version 1.1;
        proxy_pass http://ediproject;
        proxy_set_header Host $host
        proxy_set_header X-Forwarded_Host $server_name;
    }
}
```

In the first *location ediproject* block we specified the ip addresses of the two web server machines with the parameters *server*.
Inside *server* block we specified the information for nginx machine, to make it works as a load balancer. In particular, with *port 80* HTTP/1.1 has been selected as transfer protocol between clients and the load balancer. *server_name* specifies the name associated to nginx machine, also to use it in headers.
Inside the new *location* block we specified the information necessary for the load balancer configuration. *proxy_pass* specifies the addresses which receives traffic; in this case they correspond to the ip addresses specified under the *location ediproject* voice, defined at the beginning. *proxy_http_version 1.1* sets HTTP/1.1 as HTTP version for proxying. Other parameters has been set to specify specific headers to use in the communication between the load balancer and the web servers.

Testing the Infrastructure

Before carrying out the experiments, it has been necessary to test whether the infrastructure was working properly. To do this we accessed the ip address of the nginx machine from a web browser and we observed the content of the file *access.log* on all the machines.
It was possible to observe that the file present to the load balancer contained the requests for all the content of the webpage. All these requests were distribute in the *access.log* file on the two different web servers which divided the traffic between them
In this way we had the possibility to observe the correct operation of the infrastructure

Here below are reported results related to the test executed accessing via nginx the default websites present on the apache web servers.

Nginx - Load Balancer

```
87.14.3.203 - - [09/Jun/2020:08:19:23 +0000] “GET / HTTP/1.1” 200 3138 “-” “Mozilla/5.0 (Windows
NT 10.0; Win64; x64; rv:77.0) Gecko/20100101 Firefox/77.0”
87.14.3.203 - - [09/Jun/2020:08:19:23 +0000] “GET /icons/ubuntu/logo.png HTTP/1.1” 200 3338
“http://15.236.92.210/” “Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:77.0) Gecko/20100101
Firefox/77.0”
87.14.3.203 - - [09/Jun/2020:08:19:23 +0000] “GET /favicon.ico HTTP/1.1” 404 244 “-” “Mozilla/5.0
(Windows NT 10.0; Win64; x64; rv:77.0) Gecko/20100101 Firefox/77.0”
```

Apache - Web Server 1

```
15.236.92.210 - - [09/Jun/2020:08:19:23 +0000] “GET / HTTP/1.1” 200 3440 “-” “Mozilla/5.0
(Windows NT 10.0; Win64; x64; rv:77.0) Gecko/20100101 Firefox/77.0”
15.236.92.210 - - [09/Jun/2020:08:19:23 +0000] “GET /favicon.ico HTTP/1.1” 404 455 “-”
“Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:77.0) Gecko/20100101 Firefox/77.0”
```

Apache - Web Server 2

```
15.236.92.210 - - [09/Jun/2020:08:19:23 +0000] “GET /icons/ubuntu/logo.png HTTP/1.1” 200 3587
“http://15.236.92.210/” “Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:77.0) Gecko/20100101
Firefox/77.0”
```

Experiments - Page Load Times Computations

After testing the operation of the infrastructure, we made some experiments in order to compare the performances of the load balancer we implemented with the results previously obtained with the single web servers.

First experiments we performed was to compare the page load time with the results previously discussed, considering the protocol HTTP/1.1.
We executed the measurements five times, cleaning the browser cache at each repetition. Then we reported the average of the results in the table below.

	HTTP 1.1 (LOAD BALANCER)	HTTP 1.1 (NGINX)	HTTP 1.1 (APACHE)
Andrea	1.57 s	2.59 s	1.91 s
Gianantonio	2.47 s	3.40 s	3.55 s
Luca	2.45 s	2 s	3.41 s
Manuel	3.06 s	2.2 s	3.36 s
Roberto	1.81 s	6.35 s	2.83 s
Sergio	2.61 s	3.23 s	2.87 s

What we observed is that page load times are perfectly comparable with the previous measurements. This behaviour was expected by us, since the improvements given by a load balancer are evident when it is necessary to satisfy an high number of requests coming from different clients.

Experiments - ApacheBench

Second set of experiments has been carried out using the Apache Bench Linux tool.

At first the experiments were performed by a single user setting 100, 500 and 1000 packets and number of concurrent requests by the command line.
Also in this case we obtained results very similar to the case analyze with only the web servers. This is due to the behaviour of the load balancer already discussed.

Next step was to repeat the previous experiment, sending 1000 packets and number of concurrent requests by our six devices simultaneously.
In this case we observed interesting results due to the capacity of the load balancer to better satisfy a large number of requests: number of failed requests decreased significatively respect to all the experiments did using only the web servers.
While using HTTP/1.1 before, number of packet loss was about 60%, with the load balancer we implemented, number of packet losses was under the 10%.
This is due to the load balancer property of balancing the incoming traffic between the available server machines.

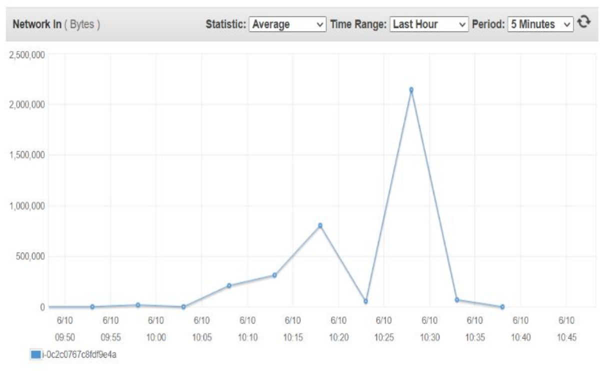
Experiments - Stress Test with Hulk Python Script

Last experiment was to use the HULK Python script in order to consider verify reliability of the service with a simulation of a DDoS attack. We observed two interesting facts:

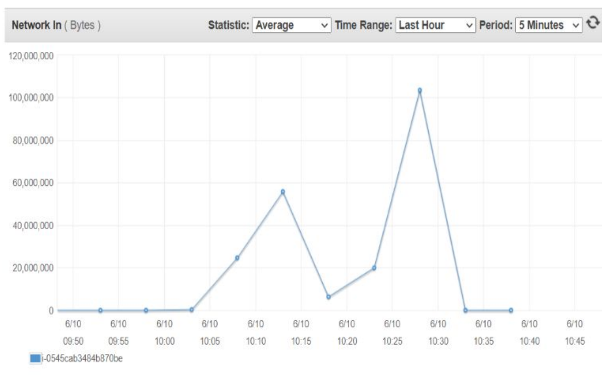
1. Differently from the test made with the previous infrastructure, websites did not go down and PLT did not changes.
2. The incoming traffic was high on the load balancer and lower on the two web servers, on which it was equally distributed.

This infrastructure was able to resist to the attack and to serve all the requests, balancing the incoming traffic between the available servers.

In the plots taken from AWS monitoring tools reported below, there is the possibility to observe the discussed behaviour. The lowest peaks are related to the experiments previously did with *ab*, while the highest are related to the last stress test executed with HULK Python script.


Network In (Bytes) Statistic: Average Time Range: Last Hour Period: 5 Minutes

Apache - Web Server


Network In (Bytes) Statistic: Average Time Range: Last Hour Period: 5 Minutes

Nginx - Load Balancer

Conclusions

Developing this project we had the possibility to analyze the implementation related to the main web protocols studying during the course and we had also the possibility to explore a part of the most widely used instruments for this purpose, in particular *apache, nginx* and some tools for testing.

We experimentally observed that the infrastructure which guarantee best performances is nginx as a web server, using HTTP/2 protocol and implementing server push technology.
We have also the possibility to experiment that for and higher reliability and in order to satisfy an high number of incoming requests, a load balancer is a very good solution.