Waterford Institute *of* Technology

Cloud Computing 2

Bachelor of Science (Hons) Applied Computing

HAProxy, SSL, Varnish Lab

Ciaran Roche - 20037160

April 16, 2019

Plagiarism Declaration

Unless otherwise stated all the work contained in this report is my own. I also state that I have not submitted this work for any other course of study leading to an academic award.

Contents

1 Introduction

This report documents steps taken during lab exercises to which demonstrate a number of caching techniques. These incorporate tooling such as HAProxy and Varnish. The overall outcome will show an understanding of these techniques along with some greater insight into the tooling mentioned.
Note – It is worth noting for brevity of the report screen shots have been limited to verification of the labs, any deviations from steps outlined in lab instructions will be documented.

2 HAProxy Lab

2.1 Introduction

HAProxy is a free open source software, which provides a high availability load balancer and proxy server. HAProxy – High Availability Proxy. Many large companies, such as GitHub, Reddit etc have adopted HAProxy into their technological stack. Due to its nature, HAProxy uses a low amount of memory as it is a single-process which is event driven, thus allowing for a large number of concurrent requests.
This first lab exercise covers the installation and setting up HAProxy to load balance between 3 NodeJS HTTP servers.

2.2 Work Completed

This work was completed using Amazons EC2 instance. The AMI choosen was a Ubuntu 16.04. Once launched the instance was updated and haproxy installed along with NodeJs. With this done the node server was built. Following these steps I explored the HAProxy configuration file.

This config file was split into a number of sections, global and default. Like many technologies today, HAProxy appears to be utilising the Linux User groups. This allows for the application to be run in isolation to other applications on a system. We can see this in the global section of the config file. For the purpose of these labs we have no need to make changes here but it is worth noting that the master process is run as root.

The default section on the other hand we can see logging and timeout options. For the most part we do not need to edit these but due to running into some problems later on, my troubleshooting brought me back to this section. This will be discussed in Section 2.3.

In order to set up our load balancing between 3 HTTP listeners we need to add the following sections:

- *frontend* – where HAProxy listens to connections
- *backend* – where HAProxy sends incoming connections
- *stats* – setup HAProxy web tool for monitoring the load balancer and its nodes

Let break down the new sections, the frontend is our localnodes, in this case it is the nodejs servers, so for that this section binds to all network interfaces on port 80. The mode is also set for HTTP connections. HAProxy is able to handle lower-level TCP connections too, but to reduce ambiguity we set the mode to HTTP. Next we declare out backend.
This leads to our backend section which is added. Here we need to configure the servers to which

we will distribute traffic between. Like with the frontend we declare our mode as HTTP, next we must declare the type of load balancing algorithm used. In this case we choose roundrobin. We now add the option to add the X-Forwarded-For header. This is so our application can get the clients actual IP address and not that of our load balanced. The following options are the manual X-Forwarded-Port information. Finally we declare our servers. With that done we must restart haproxy and begin verifying our loadbalancer.

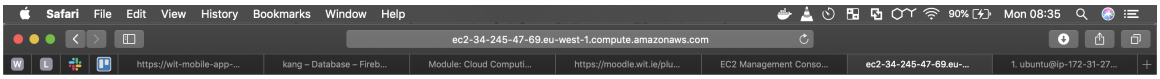
2.3 Troubleshooting

On my first run of this everything worked as expected. On my second run on which i wanted to get my screenshots for this report, on restart of HAProxy I got an error. Unfortunately I did not grab screen shots of these as I was focused on fixing the problem.
The error encountered was basically HAProxy could not start loadbalancer. On checking the HAProxy logs, very little information was given. Apart from the loadbalancer had surpassed allocated timeout. This brought me to playing around with the defaults in the config file, again the same error was thrown on restart.
I then noticed I had not set the ENABLED option within the HAProxy settings. Again this also had no affect as the same error was thrown.
After spending some time on google, I decided to tear down my instance and start again. On this time round I did not copy and paste from the exercise instructions. On restarting HAProxy no error was thrown, this leads me to believe the HAProxy file is case sensitive and on copy and pasting from the exercise sheet something had gotten mixed up. It is my only explanation for the error, but was unable to find anything on Google about it. So it could be the result of my own human error. Who knows, on my second run of it everything worked and can be seen in the following section.

2.4 Verification

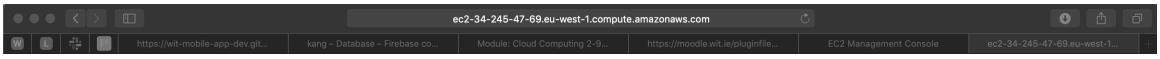
As we can see from the screen shots below, when we visited the public DNS of our ec2 instance we are directed to our node server. Subseqent refreshes show we switch port in round robin fashion, as well as seeing the X-Forwarded-For header and the IP address of the client.

Figure 1: *Port 9000*



{"host":"ec2-34-245-47-69.eu-west-1.compute.amazonaws.com","upgrade-insecure-requests":"1","accept":"text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8","user-agent":"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_3) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/12.0.3 Safari/605.1.15","accept-language":"en-ie","accept-encoding":"gzip, deflate","x-forwarded-port":"80","x-forwarded-for":"89.100.103.200"}
There's no place like 0.0.0.0:9001

Figure 2: *Port 9001*



{"host":"ec2-34-245-47-69.eu-west-1.compute.amazonaws.com","upgrade-insecure-requests":"1","accept":"text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8","user-agent":"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_3) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/12.0.3 Safari/605.1.15","accept-language":"en-ie","accept-encoding":"gzip, deflate","x-forwarded-port":"80","x-forwarded-for":"89.100.103.200"}
There's no place like 0.0.0.0:9002

Figure 3: *Port 9002*

3 HAProxy SSL Termination

3.1 Introduction

This is a follow on to the previous lab in that we are building apon our HAProxy server to implement SSL Termination. This is the practice of terminating/decrypting SSL connections at the load balancer and sending encrypted connections to the backend servers.

3.2 Work Completed

For this work we are not using our local node servers, instead we launch multiple ec2 instances. On these instances an nginx server is launched. The resulting topology can be seen in Figure 4. With these servers running it is time to reconfigure our HAProxy server.

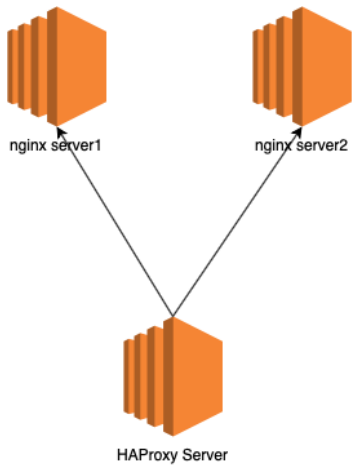


Figure 4: *SSL Termination Topology*

On the HAProxy server we must create a self-signed certificate, and a pem file. The pem file is a certificate, the key and other cert authorities concatenated into one file. With this done we now edit our HAProxy config file, here we must bind port 443 to our frontend, and point it to where our pem file is stored. This listens for all connections on port 443. Next we edit our backend nodes so that the servers are set to our nginx ec2 servers. As we are doing SSL Termination we set these to port 80 as the load balancer is handling the SSL, there is no need for our nginx servers to listen on port 443. Finally on the frontend we add a redirect so that all traffic is forced to use SSL.

3.3 Verification

Once the new configuration is handled we restart our server and visit our HAProxy public DNS. Figure 5 show the warning we get due to our cert being self signed. We can proceed from here and Figure 6 shows a successful connection to our NGINX server. As we can see our HAProxy is handling the SSL connection before forwarding the traffic. Nothing else has changed, our servers still receive the X-Forwarded-Header

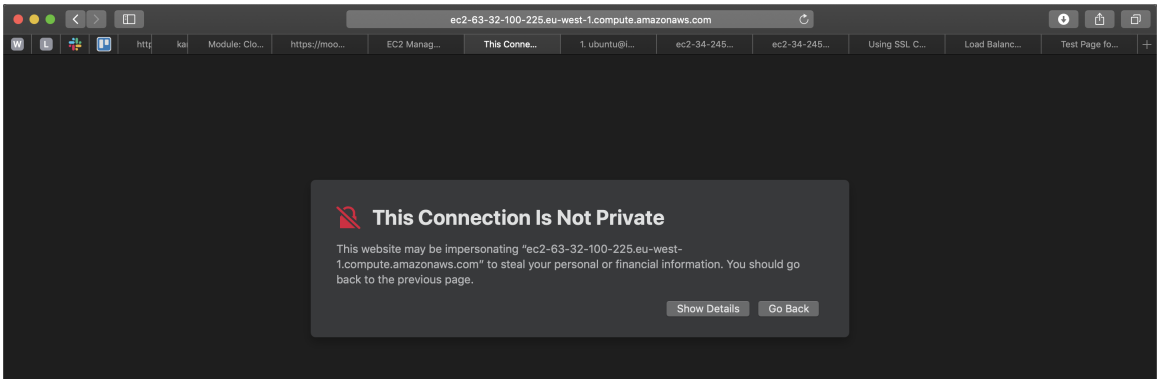


Figure 5: *Connection Warning*

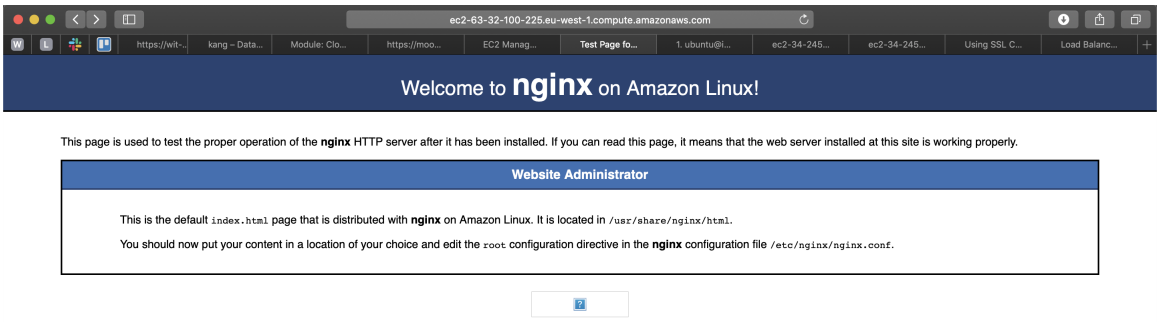


Figure 6: *Proceed to NGINX Server*

3.4 Conclusion

I feel this demonstrates some of the real power and value behind HAProxy, through minimal configuration we can add SSL to our application. While still benefiting from the X-Forwarded-Header. An other option would be to use SSL Pass Through, this is where we pass the encrypted traffic through the loadbalancer and let the nginx server handle it. As the traffic is encrypted we lose the X-Forwarded-Header. This requires a lot more configuration on both the NginX servers and

HAProxy server. From losing the X-Forwarded Header and the extra configuration I fail to see the benefit of this approach.
Overall HAProxy provides an easy to use fault tolerable solution to load balancing an application. Does it add value to existing solutions provided by AWS? I am unsure, but looking at its usage and adoption by large companies I am sure handling load balancing in this manner has its benefits.

4 Varnish Cache

4.1 Introduction

This lab looks at exploring Varnish Cache. It is so called a reverse caching proxy as it is a piece of software that you put in front of your web servers to reduce the loading times of the application by caching the servers output. It does this by mimicking the behavior of the server that sits behind it.

4.2 Steps Taken

So following the instructions laid out in the lab, I tried to build apon the infrastructure I already had running from Section 3. Unfortunately I was unable to get the lab to run as expected, to save time on troubleshooting, I felt it best to start over. In doing so I created the topology seen in Figure 7.

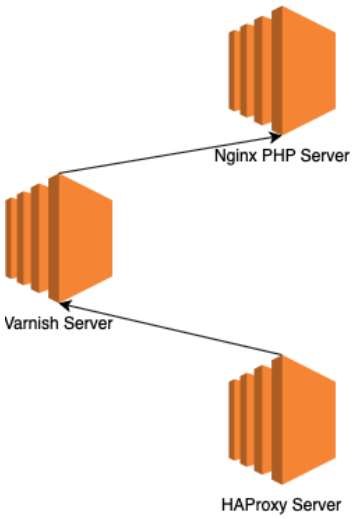


Figure 7: *Varnish Topology*

The topology consisted of 3 ec2 instances, utilizing Ubuntu 16.04 AMI's. One server acted as my HAProxy Server, another being my Varnish Server and finally an Nginx PHP Server. If we look at Figure 8 we can see my HAProxy configuration. This is the same configuration outlined in the lab handout. With the backend servers and proxy addresses changed to match that of my Varnish Server.

```
global
        daemon

defaults
        mode http

frontend http-in
        bind *:80
        default_backend servers

backend servers
        server server1 52.18.29.28:6081

frontend http-in-proxy
        bind *:81
        default_backend servers-proxy

backend servers-proxy
        server server1-proxy 52.18.29.28:6083 send-proxy-v2

listen stats
        bind *:1936
        mode http
        log global
        stats enable
        stats uri /
        stats hide-version
        stats auth someuser:password
```

Figure 8: *HAProxy Configuration*

If we note Figure 9 we can see in my Varnish Configuration I have changed the host to match that of my Nginx server. In the lab handout, along with the details from the Varnish Blog post found at Appendix A, there was discrepancies between the Nginx server on port 80 and 8080, due to running the Nginx server on a different ec2 instance I set my backend default to port 80.

```
# Default backend definition. Set this to point to your content server.
backend default {
    .host = "52.17.185.56";
    .port = "80";
}
```

Figure 9: *Varnish Backend Configuration*

Figure 10 shows the custom vcl set in the Varnish Configuration, these are to be wrote to the header of the request which was to be consumed by the Nginx server.

```
sub vcl_recv {
    # Happens before we check if we have this in cache already.
    #
    # Typically you clean up the request here, removing cookies you don't need,
    # rewriting the request, etc.
        set req.http.x-clientip = client.ip;
        set req.http.x-serverip = server.ip;
        set req.http.x-localip = local.ip;
        set req.http.x-remoteip = remote.ip;
        return(pass);
}
```

Figure 10: *Varnish Backend Custom VCL Configuration*

I took me quite some time to configure the Nginx server to render PHP, the final configuration file I have can be seen in Figure 11. With this configuration file I can reach my PHP page which can be seen in Figure 12. As we can see the page renders but as we are directly accessing varnish is not setting the IP, thus the addresses are not rendering.

```
server {
    listen 80 default_server;
    listen [::]:80 default_server ipv6only=on;

    root   /var/www/html/;
    index  index.html index.php;

    location / {
        try_files $uri $uri/ =404;
    }

    location ~ \.php$ {
        try_files $uri =404;
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        fastcgi_pass unix:/run/php/php7.0-fpm.sock;
        fastcgi_index index.php;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        include fastcgi_params;
    }
}
```

Figure 11: *Nginx configuration*

Figure 13 shows the response from the varnish server, if we request varnish directly or via HAProxy the response is the same. Time was spent insuring the instances had the correct security groups,

Figure 12: *PHP Page via Nginx DNS*

I also revised the ports and tried different combinations, before finally admitting defeat to getting the lab to run.

Figure 13: *Varnish Response*

4.3 Conclusion

Unfortunately the end result which I did not achieve should of shown the HAProxy handling incoming connections, passing the connection to Varnish which handles the caching and finally the Nginx server which serves the page when not available from the cache. The Page served should of shown the IP addresses of that of the Client, HAProxy and the Varnish Server. While HAProxy and Varnish are abstracted from the user, the end user only sees the content of the Nginx server. Overall Varnish seems pretty cool, being easily interoperable with HAProxy in that all you have to do is point your HAProxy at Varnish. All the heavy lifting is done for you. Varnish itself has quite a small about

of configuration needed. That said I found it tricky to debug and troubleshoot. Unfortunately as with a lot of technologies, when it comes to documentation it is often written by someone with a high understanding of the underlying technology, which I feel leads to valuable information being left out, so when you are only trying that technology for the first time when something goes wrong the documentation is of little help.

5 Summary

I have two take aways from this lab exercise, first, the appreciation for the work that goes in to increase the speed and performance of web applications. The tooling and services needed to enhance user experience, by abstracting all this heavy lifting away from the user to provide a better experience. Secondly how complex these systems can become. In this case of a simple PHP application, as I was unable to render the application as expected I have now introduced 2 additional layers to my application that I need to debug. At the start I was unable to render the PhP page as is, this took time to figure out my initial problems where rendering PHP on my Nginx server, finally over coming that problem, I had to look where next in the chain I was running into the problem of fetching the php page. In this case i was unable to fix the problem, which leads me to think of how complicated it must be at a larger scale when problems do occur.

Appendices

A Varnish Blog Post

http://feryn.eu/blog/varnish-4-1-haproxy-get-the-real-ip-by-leveraging-proxy-protocol-support/