Cloud Computing: Exploring HAProxy and Varnish Cache

Dimitra Zuccarelli, February 2019

1 HAProxy

HAProxy is a (reverse) proxy and load balancer for both HTTP and TCP, which provides two essential features:

- It listens for incoming connections on what is known as a `frontend`. In a typical HTTP reverse-proxy configuration with HAProxy, the frontend represents the address of the load balancing server and the port(s) that connections are accepted on (typically port `80`). Once any processing is performed on these packets, they are forwarded to what is known as a `backend`. A backend defines a cluster of backend servers and the load balancing algorithm used to distribute packets among these servers. A commonly used algorithm is the `round robin` algorithm, which, as the name suggests, distributes requests in turn to each backend server. Other slightly more sophisticated algorithms might examine the server that has the least number of connections or the lowest resource usage, and pass the connection on to that server.
- HAProxy also performs health checks by continuously monitoring the state of the backend servers, and only sending traffic to them if they are valid, to make sure no connections are lost. By not sending requests to servers which have failed the health check, the user will never be aware of any issues in the backend (unless all servers go down, which is unlikely). This ensures high availability, reliability and fault tolerance.

1.1 Lab exercise - Part I

Following the steps provided in the lab sheet, HAProxy was installed on an EC2 instance running Ubuntu 16.04. Next, Node.js was installed and three node servers were started on three different ports (`9000, 9001, 9002`). These servers act as the backend servers that HAProxy will load balance incoming requests across.

The two main sections of the configuration file used to setup HAProxy are defined below. The entire detailed explanation of the configuration file is provided in the lab sheet.

1. HAProxy is listening on port `80` for incoming requests to the application. Once a request is received, it will forward it on to one of the servers in the backend called `nodes`. The nodes correspond to the three node servers started on localhost:

   ```
   server web01 127.0.0.1:9000 check
   server web02 127.0.0.1:9001 check
   server web03 127.0.0.1:9002 check
   ```

   The server which will handle the request is decided based on the `round robin` algorithm.

2. The `option forwardfor` is set to add the `X-Forwarded-For` header to the packet, in order to identify the IP address of the requesting client. Without this, the application would see every incoming request as originating from the load balancer's IP address.

{"host":"63.35.215.211","upgrade-insecure-requests":"1","user-agent":"Mozilla/5.0
(Macintosh; Intel Mac OS X 10_14_2) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/72.0.3626.119 Safari/537.36","dnt":"1",
"accept":"text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8"
"accept-encoding":"gzip, deflate",
"accept-language":"en-GB,en-US;q=0.9,en;q=0.8", "x-forwarded-port":"80",
"x-forwarded-for":"80.233.45.248"}
There's no place like 0.0.0.0:9002

{"host":"63.35.215.211","cache-control":"max-age=0","upgrade-insecure-requests":"1",
"user-agent":"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_2) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/72.0.3626.119 Safari/537.36","dnt":"1",
"accept":"text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8"
"accept-encoding":"gzip, deflate",
"accept-language":"en-GB,en-US;q=0.9,en;q=0.8", "x-forwarded-port":"80",
"x-forwarded-for":"80.233.45.248"}
There's no place like 0.0.0.0:9001

{"host":"63.35.215.211","cache-control":"max-age=0","upgrade-insecure-requests":"1",
"user-agent":"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_2) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/72.0.3626.119
Safari/537.36","dnt":"1","accept":"text/html,application/xhtml+xml,application/xml;q=0.9,image/we
"accept-encoding":"gzip, deflate",
"accept-language":"en-GB,en-US;q=0.9,en;q=0.8", "x-forwarded-port":"80",
"x-forwarded-for":"80.233.45.248"}
There's no place like 0.0.0.0:9000

As expected, killing one of the Node servers does not affect the client in any way, because HAProxy simply skips over the problematic server.

nodes

	Queue			Session rate			Sessions						Bytes		Denied		Errors		Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
web01	0	0	-	0	1		0	1	-	23	23	25s	7 387	11 690		0		0	0	0	0	1m44s UP	L7OK/200 in 0ms	1	Y	-	4	2	12s	-
web02	0	0	-	0	2		0	1	-	27	24	25s	7 714	11 950		0		1	0	3	0	14m UP	L7OK/200 in 0ms	1	Y	-	3	1	6s	-
web03	0	0	-	0	1		0	1	-	18	18	3m24s	5 652	8 960		0		0	0	0	0	1m47s DOWN	L4CON in 0ms	1	Y	-	4	2	1m53s	-
Backend	0	0		0	3		0	1	200	65	65	25s	20 753	32 600	0	0		1	0	3	0	14m UP		2	2	0		1	5s	

Figure 1: 'nodes' backend

2 Lab exercise - Part II

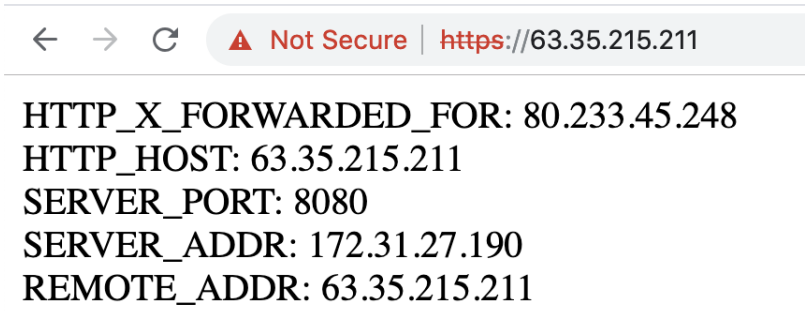
In the next part of the lab, HAProxy is configured to use HTTPS for secure communication. There are two strategies for achieving this:

- **SSL Termination**. With SSL termination, certificates are installed on the load balancer, which decrypts packets and sends the unencrypted connections to the backend servers. This strategy allows `X-Forwarded-For` headers to be added to the packets, in order to determine the IP address of the originating client. This also means that certificates only need to be managed in one location, instead of at every server in the backend cluster.
- **SSL Passthrough**. With SSL Passthrough, the encrypted packets are sent directly to the backend servers. This strategy might be used when a third party provider is being used for the load balancer, and the network link between the load balancer and the backend servers is not trusted.

In both cases, HTTPS connections were successfully established following the steps outlined in the lab sheet, so these steps are not repeated here. A simple explanation of the outcome of both examples is given below. The following PHP script was used to display the various IP addresses and port numbers requested:

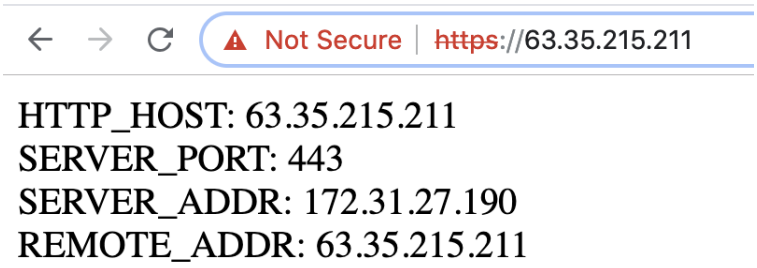
```
<?php
    $vals = array("REMOTE_ADDR", "SERVER_ADDR", "HTTP_HOST", "HTTP_X_FORWARDED_FOR",
        "SERVER_PORT");

    foreach ($_SERVER as $header => $value) {
        if (in_array($header, $vals)) {
            echo "$header: $value <br />\n";
        }
    }
?>
```

2.1 SSL Termination



- **X-Forwarded-For**: The IP address of the requesting client. In this case, the address is my public IP address.
- **Http Host**: The IP address of the EC2 HAProxy instance.
- **Server Port**: The port that Nginx is running on. In this case my Nginx instance is configured to listen on port `8080`.
- **Server Address**: The IP address of the Nginx server.
- **Remote Address**: Technically the client IP address, but because the load balancer acts as a middle man between the nginx server and the client browser, this variable returns the IP address of the HAProxy instance. That is why setting the `X-Forwarded-For` header is necessary.

2.2 SSL Passthrough



- **X-Forwarded-For**: These headers are no longer included, given that in order to pass a secure connection to a backend server without encrypting it, `tcp` mode needs to be used over `http` mode. The `option forwardfor` and other header options are removed, since they cannot be used in tcp mode, and headers cannot be injected into encrypted packets.
- **Http Host**: The IP address of the EC2 HAProxy instance.
- **Server Port**: The port that Nginx is running on. Compared to the SSL Termination exercise, this port is now `443` because the connection to the Nginx instance is secure.
- **Server Address**: The IP address of the Nginx server.
- **Remote Address**: Technically the client IP address, but because the load balancer acts as a middle man between the nginx server and the client browser, this variable returns the IP address of the HAProxy instance. Unfortunately because of the SSL passthrough implementation, there is no simple way of knowing the IP address of the initial client.

3 Caching with Varnish

Varnish is an application accelerator which speeds up delivery of a web page by a factor of 300-1000x. Varnish can be used with the current HAProxy and Nginx set up to cache the web page for faster delivery to the end user.

The `proxy` protocol is a protocol developed by HAProxy to allow chaining of multiple proxies and reverse proxies without losing the original client's information. In this part of the lab, both options - using the proxy protocol and not using it - are explored.

The architecture setup is shown in the following diagram:

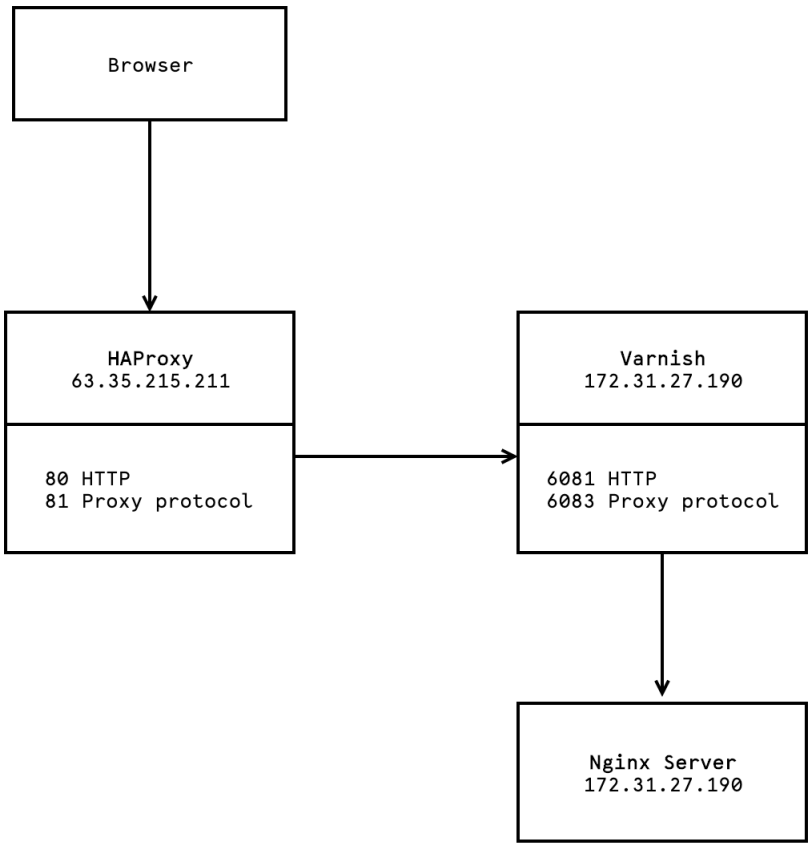
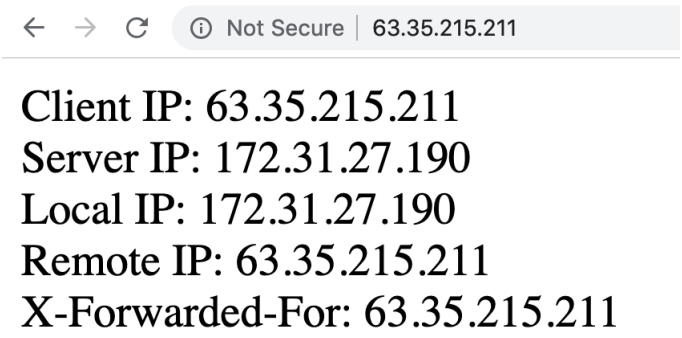


Figure 2: Architecture

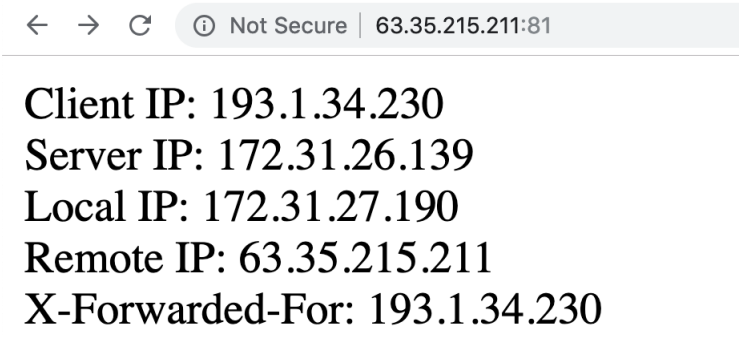
- Varnish can act as a load balancer itself, but does not support SSL communication. For advanced load balancing and HTTPS a reverse proxy like HAProxy should be used in front of Varnish.
- In this setup, HAProxy is configured to support regular HTTP on port `80` and the Proxy protocol on port `81`. When requests come in on port `80`, HAProxy forwards these to port `6081` on the Varnish server. Likewise, when requests come in on port `81`, HAProxy redirects these to port `6083` (which supports the proxy protocol) on the Varnish server.
- Once the request is sent to Varnish (unencrypted), Varnish will mimic the response of the web server, or if no hit is found, redirect the request to the web server itself. In this setup, Varnish Cache and Nginx are running on the same EC2 instance.

3.1 No proxy protocol



- **Client IP**: The IP address of the EC2 HAProxy instance. Ideally we would expect this to be the IP address of the user's browser.
- **Server IP**: The IP address of the server which hosts the Nginx and varnish servers.
- **Local IP**: The IP address of the server where the varnish server is located.
- **Remote IP**: The IP address of the server/device just before the Varnish server (in this case the EC2 HAProxy instance)
- **X-Forwarded-For**: The IP address of the HAProxy instance, not of the client's browser, which isn't very useful. If for example the web page needed to be served from an edge location closer to the end user (as we saw in the CloudFront AWS talk), only having the IP address of the remote client (in this case HAProxy) means that the user's actual location isn't known. Using the Proxy protocol would solve this issue, since it is able to retain the information of the original user.

3.2 Proxy protocol



- **Client IP**: My public IP address. Using the proxy protocol means this information is retained, and no longer shows the address of the HAProxy instance, which wasn't useful information to log.
- **Server IP**: The private IP address of the HAProxy server.
- **Local IP**: The private IP address of the Varnish server.
- **Remote IP**: The public IP address of the server/device just before the Varnish server (the EC2 HAProxy instance)
- **X-Forwarded-For**: The client IP address. Again, this field now shows the correct value that would be expected.

4 Conclusions

While the configuration may not always be easy to manage, the advantages of using reverse proxies and caching mechanisms like HAProxy and Varnish are clear. Using HAProxy as a load balancer with SSL enabled allows the developer to set up a secure, highly available, reliable, fault tolerant infrastructure with relatively low setup. Including Varnish Cache allows speed improvements of up to 1000x times, and along with the proxy protocol, means no information about the client is lost, regardless of how many layers exist between the end user and the web server.