Hernan Espinosa Reboredo
SIGMA Gestió Universitaria,
Universitat Autonoma de Barcelona
Sant Cugat del valles, Barcelona, Spain
hespinosar@gmail.com

I. INTRODUCTION

The aim of this project is to build a high availability Web Application Firewall (WAF) [1] that will monitor and analyse client's requests data keeping in mind the protection of backend server's security.

A WAF helps to protect web applications by filtering and monitoring HTTP traffic between a web application and the Internet. By deploying a WAF in front of a web application, a shield is placed between the web application and the Internet, increasing security, performance and reliability by having client's requests pass through the WAF before reaching the servers. A WAF operates through a set of rules often called policies. These policies aim to protect against vulnerabilities in the application by monitoring and filtering out malicious traffic.

In this project, we are going to build a WAF for a high availability front end, and we are going to carry out an exhaustive analysis of the events that take place in the built architecture, using different development tools to monitor the traffic.

The main reason that has motivated this work is to improve the security of any architecture's servers, to have a better control over the events of their activity, and to reduce the time of action before an incident. Therefore, this project is appliable to any web application architecture in order to improve its security.

II. OBJECTIVES

The WAF must be able to protect web applications by filtering and monitoring HTTP traffic between a web application and the Internet. We have planned a set of objectives to achieve by the end of this work. The objectives are listed in incremental priority.

- To learn how a WAF works, and how is it used.
- To learn how the different software tools, that are used in this project, work and how are going to be for use to make the project possible.
- To design, implement and test a scalable high availability client-server web application architecture.
- To apply the data persistency and load balancing feature to the web application firewall for redundancy and performance efficiency purposes.
- To develop and add rules to the service to improve security.
- Log management to analyse all the client requests that are done in real-time and too establish rules to grant or deny the access to the backend servers.

III. METHODOLOGY AND PLANNING

This project follows a software development model based in incremental prototyping [2]. Each prototype will fulfil a set of requirements. The final prototype will accomplish the objectives from this work. Each prototype built will serve as a mechanism for the definition of requirements, since requirements can change from the initial requirements definition.

This work is divided into three main phases, Web Application Firewall version 1 (WAF_V1), Web Application Firewall version 2 (WAF_V2) and Web Application Firewall version 3 (WAF_V3), without keeping in mind the delivery of the reports and the presentation. For every phase these steps are going to be considered for the development:

The architecture analysis, to justify the proposed structure, explaining the decisions made and reasons why the structure is suitable.

The architecture design, to design the proposed structure, explaining the decisions made and reasons why the structure is suitable.

The architecture implementation, to implement the design proposed.

The architecture tests, to validate that every project phase satisfies the initial requirements.

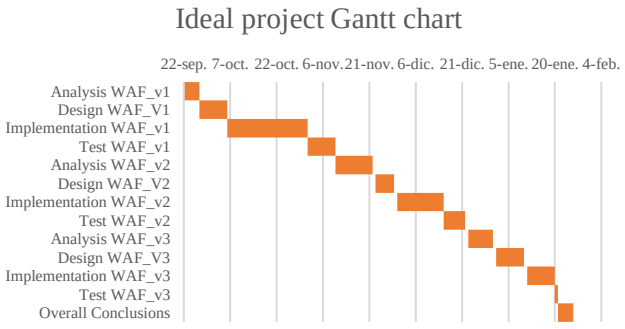

Figure 1 shows the ideal Gantt chart diagram for this work.

The methodology that will be used to follow the development of this project is the Gantt chart, which is a useful way of scheduling a project and defining the different dependencies between tasks. In the figure shown above, we can see the layout of the ideal progress from the phases of the project throughout the time. In figure 1 from the appendix there are the details from the Gantt diagram from above. In addition to

the Gantt diagram above, in Figure 2 from the appendix, there is a detailed Gantt diagram containing the project phases, their development, deliverables, project planning in weeks of work and the different activities related to a phase of the project.

This project is limited by time and budgetary resources. Since there is one developer dealing with the project, and the developer is an undergraduate engineer university student, the development is limited by the salary of the developer and the time limit of 300 hours, to finish the project.

IV. TOOLS FOR DEVELOPMENT

In this section of the paper we are going to describe the set of software that is going to be used to develop this work.
This project will be developed using a Linux operating system, Ubuntu.

A. Docker: The development environment

Docker [4] is a tool designed to make it easier to create, deploy, and run applications by using containers [5]. Containerization allow us to package up an application with all the parts it needs, such as libraries and other dependencies, and run it all out as one package. Thanks to its environment management, makes it a suitable environment for this project.

Separating the different components of our web application service into different containers will have security benefits, because if one container is compromised the others remain unaffected. Separating the different components of our web application into different containers will avoid conflicts with dependencies, therefore, gaining practicability.

B. HAProxy: Building tool

Since in this work a high availability web application will be built, we are going to use the HAProxy software to build the high availability load balancers. HAProxy is an open source, very fast and reliable solution offering high availability, load balancing, and proxying for TCP and HTTP-based applications. It is particularly suited for web application services.

C. Apache ModSecurity: Building tool

For the monitorisation of traffic load, we will use Apache ModSecurity software for analysis to be able to make conclusions. Apache ModSecurity is a toolkit that will be used for real-time web application monitoring and logging.

D. Elastic Stack: Data Management tool

The Elastic Stack [6] is a powerful search engine which will allow us to process logs generated from the architecture. Elastic Stack is a complete end-to-end log analysis solution which helps in deep searching, analysing and visualizing the log generated from different machines.

Elastic stack will let us search through the multiple logs at a single place and identify the issues spanning through multiple servers by correlating their logs within a specific time frame found in our environment.

E. Security Onion: Testing tool

Security Onion is a free and open source Linux distribution for intrusion detection, monitoring, and log management. It includes Elasticsearch, and many other security tools.
Security Onion collects many tools for forensic analysis, both networks and systems, so that we can guarantee the proper functioning of all of them and the absence of all kinds of intruders in the net.

F. Apache AB: Testing tool

Apache AB is an Open source testing tool developed by the Apache organisation used for benchmarking an HTTP web server. Apache AB will be used to measure the performance of the architecture built. Thanks to its ease of use, we will use Apache AB to test if the architecture built works as expected.

G. Apache JMeter: Testing tool

Apache JMeter is an Open Source software designed to test functional behaviour and measure performance of a web application or a variety of services. Apache JMeter will be used to test the web application firewall efficiency and performance.

H. Github: Version Control tool

A versioning tool is being used, in order to keep copies and record of the work done over the course of this project. A repository has been created in GitHub [3] where the project is updated, and version controlled.

V. PHASE 1: WAF_V1

a) Analysis

The first prototype of the project consists in building a client-server architecture composed of three nodes. The client node, which generates the requests, the intermediary node, which will be used to secure the server node, and the server node, which will handle the client requests and will generate responses. The architecture will serve an index.hml file stored in the backend server. The objective of this phase is to develop an simulate a web server architecture including the intermediary node, which will work as a load balancer in future prototypes. This phase will work as a base for future upgrades in the next phases. Different resources and processes will be carried out to suit the different objectives from the work. With this phase we want to improve backend server's security and reduce security vulnerabilities from the web application service.

To keep up with the main objectives from this work, we are going to develop data persistence in the nodes which will serve the clients requests, we are going to manage architecture networking, and we are going to centralise the logs from the service's activity in order to know what is happening during the request and response of a user request.

b) Design

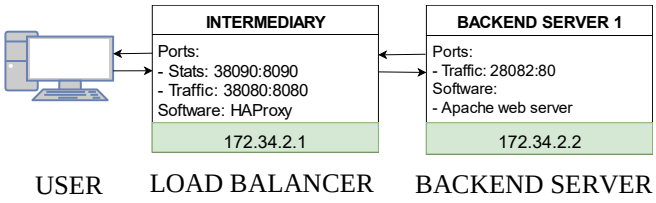

Figure 2 shows the UML component diagram from the design structure proposed for phase one of this work.

In Figure 1 above, we can observe the layout of the three nodes which compose WAF_v1. Both Intermediary and Backend Server 1 nodes are built on Docker containers and a private network has been designed to host the architecture. The Backend Server 1 node consists of an Apache web server which serves an index.html file that will be requested by the user client.
In this phase of the project, the Intermediary node of the architecture is built to hide the Backend Server node's IP from the Internet, making it more secure. Every time a user requests the index.html file from the backend server, instead of connecting to the server endpoint directly, the user will request the file using the intermediary's endpoint, which works as a proxy. The intermediary node will redirect the client's request to the server as well as it will redirect the server's response to the user.
Because of the need of transparency and privacy of the backend server, we have opted to use the HAProxy software in the intermediary node, as its software can be configurable to work as expected.
Since we want to be able to run the Docker containers always with the same configuration, data persistence has been developed.
Since we want to monitorise the activity from our architecture, logging has been configurated so we can track requests and see if everything works as expected.

c) Implementation

Since we are using containerization to develop our work, data doesn't persist when a container no longer exists, and it can be difficult to get the data out of the container if another process needs it. To manage that problem, we are using Host-based Persistence among containers, in order to store files in the host machine, even after the container stops. By using host-based persistence [7], data is persisted outside of the container, which means it will be available when a container is removed. In WAF_v1, data persistence is important since we need our architecture's containers to centrally store logs to the same directory, making it easier to process the logs, so we can test the architecture's performance.

For log management, we are going to use the software Rsyslog. Rsyslog is a powerful, secure and high-performance log processing system which accepts data from different types of source and outputs it into multiple formats. Redirecting all the logs from rsyslog to the standard out device makes logs play nice with docker default logging.

In this configuration, the Intermediary node uses the software HAProxy to load balance traffic across the architecture. Concerning HAProxy event logging, the software, doesn't log to stdout by default. To solve that problem we built a configuration which takes the logs generated by the software and sends them to a specific local directory both in the container, and the local machine, where the containers are executed.

Since the design is scalable, in a future situation we may have the situation where each backend server is connected to different databases to serve client requests or we may have the situation where we have multiple load balancers that are connected to each other in a certain configuration. Data is generally replicated to enhance reliability or to improve performance. One major problem is to keep replicas consistent. Data Base servers as well as the load balancers must be configured to perform Master-Master replication as load balancing involves both reading and writing to all backends.

When configurating the network for the architecture environment in this phase of the project, a private network has been used for security reasons, since sensitive data can travel through the private network. If a public network configuration is used, the sensitive information would face security vulnerabilities, because of the information not being encrypted. The HAProxy intermediary node, is a basic load balancing node which listens on a specific IP address and port, then forwards the incoming traffic to a specified server.

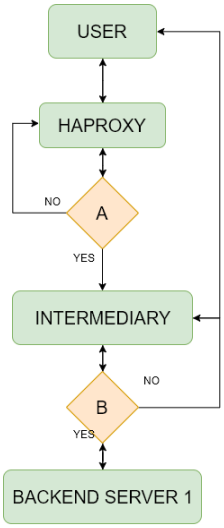

Figure 3 shows the UML flow diagram from WAF_v1 of this work.

In the figure, there are represented in incremental order of priority, from letter A to B, the different possibilities from which the route of the request can take through the architecture, in flow of execution.
In this use case, when the architecture is set up, the intermediary is the load balancer node, and the backend server 1 is the backend server which serves the index.html file.

By typing to any web browser, the following URL address, 172.43.1.1:28080/index.html, composed by the endpoint from the intermediary node, the index.html file stored in the Apache backend server node is requested by the User.
Possibility A takes place: If the intermediary node is available, the request will be handled by the node in question. The HAProxy software will handle the request and will forward it to the backend server. If the intermediary node is not available, the HAProxy software will send to the USER an HTTP 503 error response advertising the unavailability of the service. Then possibility B takes place, if the backend server is available, it will deal with the response, which will be delivered to the client by the intermediary node. If the backend server is not available, the intermediary node will send the User an HTTP 503 error response advertising the unavailability of the service. Note that in any of the events, the Backend server IP has been exposed to the Internet, the IP that has been exposed is the one from the intermediary node.

Thanks to data persistency, and the software Rsyslog, the logs from the http request will be stored in a directory from the host's computer and we are able to trace the client request and response.

d) Test

In this part we are going to perform an implementation test to the architecture to verify that the prototype follows the initial requirements.
Since the main objective of this phase is to build a base client-server architecture, functional testing is going to be performed for quality assurance. Black-box testing will be performed, which bases its test cases on the specifications of the software component under test. Functions are going to be tested by feeding them input and examining the output. In this phase, stress tests are not going to be performed.

For the test, we are going to simulate a test case where a user performs a single HTTP request to the index.html file. We are going to use Apache Benchmarking. It is designed to give us an impression of how the current Apache installation performs. The aim of this test is not to test how many requests per second our Apache installation can serve, but to ensure the identification of functions that the software is supposed to do.

By opening a new terminal window in the virtual machine environment and typing the following command line: `ab -n 1 http://172.34.2.1:8080/index.html` we are performing a functional test which the index.html file is requested one time. The test will finish when the ApacheAB software receives a response from the server. The tool gives us information about the request, such as the server hostname, IP, port and documents requested which can be found in detail in figure 2 from the appendix.

Since the Intermediary node is the main load balancer from the architecture, by checking the log from the request, we can assure that the request has gone through the path expected. The log details from the query are in Figure 3 from the appendix.

By analysing the log file from the HAProxy software from the intermediary node and the results from the ApacheAB test, the request follows the path designed and the architecture performs as expected, passing the test performed.

VI. PHASE 2: WAF_V2

a) Analysis

In this phase of the project, phase 1 has been upgraded by developing high availability load balancing and data redundancy to the architecture, by adding two nodes in the front-end of the architecture. The main objective of this phase of the project is to set up a simple IP failover between two servers, which will be the load balancers. We want to develop and understand how high availability load balancing works, how different configurations can have an impact to the architecture's performance and usability, and to improve backend server's security.
In order to carry out the objective, there has been an improvement regarding the architectural configuration from the WAF_v1's set-up, by the addition of more nodes either on the frontend side of the architecture and the backend, and the improvement in the software configuration which will suit the architecture's needs.
Performance tests have been performed, as well as the comparison between WAF_v1 and WAF_v2 from the project, in order to analyse the importance of a high availability load balancing set up. Different resources and processes have been carried out to suit the different objectives from the work.

b) Design

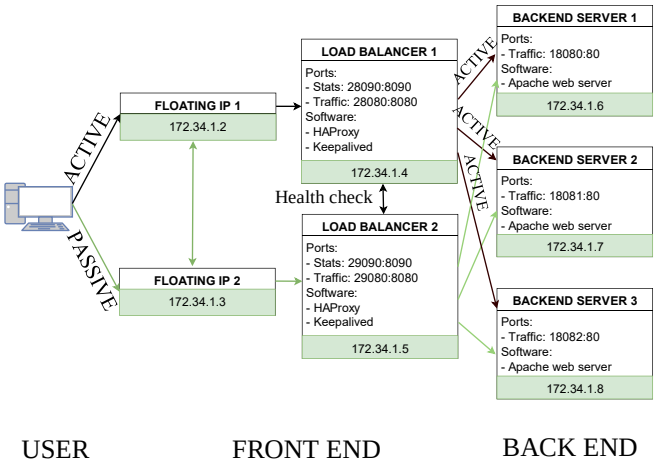

Figure 4 shows the UML component diagram from the design structure proposed for phase two of this work.

In Figure 2 above, we can see the layout of the nodes which compose the architecture's prototype. Every node in this configuration is built on Docker containers and a private network has been designed to host the architecture. Backend server one, two and three consist of an Apache web server which serves an index.php file, which contains the path of the request throughout every node. In this version of the WAF

we have opted to use the index.php file since php has multiple native methods, from its language, which makes it easier to get the IP and Hostname from a certain node from the architecture and display it in the html file. The file is the element which will be requested by the user client. In this phase of the project, the load balancers are built to provide robustness to the architecture.

Every time a user requests the index.php file from the backend servers, instead of connecting to the server endpoint directly, the user will request the file using the intermediary's endpoint, a virtual IP, which works as a proxy which will redirect the request to the load balancers. Considering high availability, one of the load balancers will be an active node, which will redirect the request to the backend servers and will redirect the response to the User. The other load balancer will be a passive node. If the main server fails or its unavailable, the passive node, or the one that is available, takes the Master role and can redirect and balance client request throughout the architecture, making the design more secure and reliable.

Because of the need of high availability as well as keeping transparency and privacy of the backend servers we have opted to use KeepAlived software alongside HAProxy software to run the load balancers, as its software has been configured to work as expected. The backend of the architecture consists of two Apache web servers, which will respond the index.php file that will be requested by the User, to test the front-end is working as expected.

c) Implementation

In this version of the project we have set up a two-node front end load balancer in an active/passive configuration with HAProxy and Keepalived, it is important to note that this configuration is a concrete use case, this configuration can be scaled by adding more load balancer nodes, or reduced by removing load balancers, satisfying a use case's needs. The load balancers sit between the user and two backend Apache web servers that hold the same content. The backend servers from this configuration reassembles a company's service backend. Not only does the load balancers distribute the requests to the three backend Apache servers, but it also checks the health of them to check their availability. If one of them is down, all requests will automatically be redirected to the remaining backend servers. In addition to that, the two load balancers monitor each other using the software Keepalived, and if the master fails, the slave becomes the master, which means the users will not notice any disruption of the service.

Keepalived is the software that manages the configuration of the Virtual Floating IP (VIP) from the architecture frontend. Thanks to Keepalived, we can scale our architecture by adding more load balancer nodes at any time in the future if it's needed, without disrupting the service.

Keepalived uses VIP's to perform load balancing and failover tasks on the active and passive routers, while HAProxy performs load balancing and high-availability services to TCP and HTTP applications. All nodes running Keepalived use the Virtual Redundancy Routing Protocol (VRRP) [8]. The active node sends VRRP advertisements at periodic intervals; if the backup nodes fail to receive these advertisements, a new active node is elected.

Keepalived performs failover on layer 4, upon which TCP conducts connection-based data transmissions. When a real server fails to reply to simple timeout TCP connection, Keepalived detects that the server has failed and removes it from the server pool.

HAProxy offers load balanced services on layer 7, HTTP and TCP-based services, such as Internet-connected services and web-based applications. Depending on the load balancer scheduling algorithm chosen, HAProxy can process several events on thousands of connections across a pool of multiple real servers acting as one virtual server.
The scheduler determines the volume of connections and either assigns them equally in non-weighted schedules or given higher connection volume to servers that can handle higher capacity in weighted algorithms.

HAProxy lets configure the load balancer to load balance traffic to the backend servers in four different algorithms. A static Round Robin, where each backendserver is used in turns per their weights. Unlike the Round Robin algorithm, changing server weight during execution is not an option. When a server goes up, it is immediately introduced into the farm once the full map is recomputed and the load balancing algorithm does the scheduling.
The second algorithm is Least Connections, where in this algorithm, the server with the least number of connections receives the traffic load. This algorithm is dynamic, like Static Round Robin.
The third algorithm is Source. In this algorithm, the source IP is hashed and divided by the total weight of running backend servers. If no server goes down or up, the same client IP will reach the same server, every time. The situations will change every time the hash result changes.
The fourth algorithm is URI. In this algorithm the URI from the client request is hashed and divided by the total weight of running servers. In this scenario, the same URI will be redirected to the same server if the total server weight never changes.
The fourth and final algorithm is URL parameter. In this algorithm, a URL parameter is specified in the configuration file. If the parameter that's found is followed by an equal sign and value to the URL parameter, the value is hashed and divided by the total weight of running servers.

Depending on the situation and the use case of the architecture, an algorithm or another will be used. Using a Least Connections algorithm is useful when very long sessions are expected, such as SQL, LDAP, etc., whilst using a Round Robin algorithm is useful when we want to assign equitable weight to every server from a net of servers.

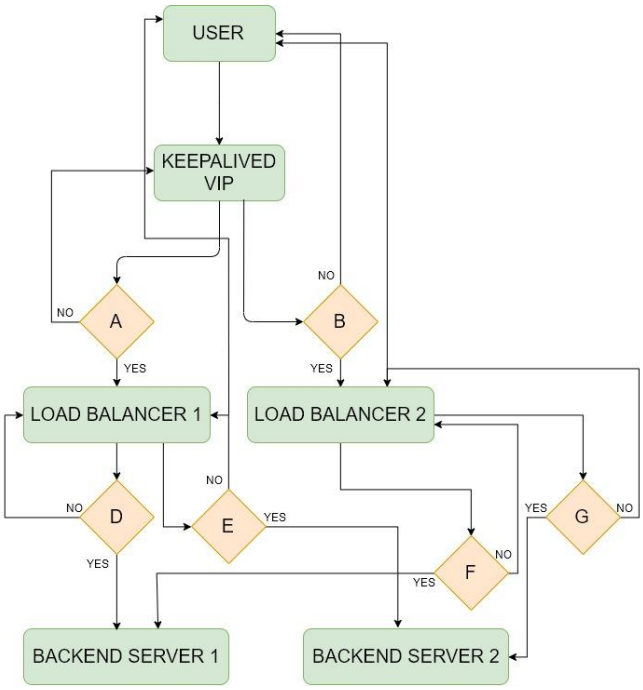

Figure 5 shows the UML Flow diagram from WAF_v2.

Due to the complexity of WAF_v2 architecture, in Figure 2.2 we have arranged the representation of a UML flow diagram reassembling an example use case from a user requesting the index.php file from the backend servers.

In the figure, there are represented in incremental order of priority, from letter A to G, the different possibilities from which the route of the request can take through the architecture's use case, in flow of execution.

In this example, when the architecture is set up, the LoadBalancer 1 is the Master node, and the LoadBalancer 2 is the Backup node. The algorithm that the HAProxy is using to balance the traffic to the backend servers, in the current in the current configuration, is a RounRobin algorithm.

The User types to any web browser the following URL address, 172.43.1.2:8080/index.php, which is the VIP endpoint. Through the VIP we are accessing through the virtual endpoint to the Master node. Thanks to the Keepalived software, the software in the master node sends periodic advertisements to the other load balancer, to check its status of operation. Possibility A takes place: If Load balancer 1(LB1) is available, the request will be handled by the node in question. If LB1 is not available, possibility B takes place. If Load balancer 2 (LB2) is available, which is the backup node, the VRRP instance determines the running status of the active node. Since the active node fails to advertise after a previously configured interval, Keepalived initiates failover and its status will change to master, as for LB1, it will change its status to inactive. In LB2, the HAProxy software will handle the request and will forward it to the backend servers, performing layer 7 traffic load balancing. If LB2 is also not available, an HTTP 503 error response is given to the user, communicating the unavailability of the service.

Concerning the backend servers, possibility D goes in pair with possibility F, but changing the backendserver node. The same happens with possibility E and G.
Possibility E and D: If the backendserver is available, the request will be handled from the load balancer to the backendserver node. If the backend server is unavailable, since replication is implemented for both backend servers, then possibility E and G take place: If the other backendserver is available, the request will be handled by the node available, if either of the backend servers are available the User will be informed with a HTTP 503 error response, generated by the load balancer, advertising the unavailability of the service.

For testing purposes, to follow the trace that the request has taken, the index.php file shows the request's route taken, by showing the IP and Hostname where the request paquet has passed.

As well as with the index.php file, by checking the load balancer logs, we can overview that the traffic from the client request and response goes through all phases anticipated, and we can corroborate that the implementation of the design works.

Thanks to the upgrades to WAF_v1, the internal implementation of our architecture is not exposed to the public internet. By having multiple nodes for both the front end and backend, front-end high availability has been developed, making the design more robust, fault tolerant and reliable to overtake large amounts of client requests.

a) Test

In this part we are going to perform an implementation test to the architecture to verify that the prototype follows the requirements.
Since the main objective of this phase is to perform high availability load balancing with the client requests, functional testing is going to be performed for quality assurance. A type of black-box testing that bases its test cases on the specifications of the software component under test is also going to be performed. In this test, functions are going to be tested by feeding them input and examining the output. In the next progress report, stress tests are going to be performed in order to test the design's capabilities.

For the tests, we are going to simulate a test case where multiple users perform queries requiring the index.php file. We are going to use Apache Benchmarking, the same tool used for testing WAF_v1.

The aim of this test is not to test how many requests per second our Apache installation can serve, but to ensure the identification of functions that the software is supposed to do. The testing and analysis of how many requests per second our Apache installation is capable of serving under heavy load, the point of failure of our architecture, the average response time of the architecture under different loads and the maximum number of requests per second which the architecture can handle will be conducted in the final report from this work.

The test is performed with the Keepalived configuration with LB1 as the master node and LB2 as the backup node. The HAProxy node's software load balancing algorithm is set to Round Robin. The actual test is performed with a default configuration, meaning that there is not rate limiting or SSL termination.
Since the backend from the architecture is composed by three backend servers, and the algorithm used to distribute the load is RoundRobin, we are going to perform 3 queries, so we can assure that all backend servers are requested the index.php file.

By opening a new terminal window in the virtual machine environment and typing the following command line: `ab -n 3 http://172.34.1.2:8080/index.php` we are performing a functional test where the index.php file is requested three times. The test will finish when the ApacheAB software receives a response from the servers.

The detailed results can be found in Figure 5 in the appendix. By checking the log from the requests, in the master node, LB1, we can see that the requests have followed the path expected. The detailed log information can be found in Figure 6 from the appendix.

Every time a request is made, if the LB1 stays as master node, the request will follow the path from the client IP, to the backendserver 3 IP all the way through the LB1. If the load balancer's status changes, and the second load balancer is available, the path from the queries will change and the request will be handled by the LB2. If any load balancer is added to the architecture, it's status would be backup, meaning that in case of failover, the nodes would be able to take the task of a master node.

As we can see, from the results from the test, the request conducted is simple, therefore, the weight in processing power is very little. By analysing the log file from the HAProxy software from the intermediary node and the results from the ApacheAB test, the request follows the path designed and the architecture performs as expected, passing the test performed.

VII. Conclusions

The increase of global connectivity throughout the Internet to satisfy the quality of service from different companies' services, is becoming a necessity as the companies grow and offer more services. In order to satisfy the needs of a business growth, scalability is a must to meet market demands. In most cases, system scalability issues appear as performance problems caused by capacity limitation of servers and networks.
By building a high availability web application front end architecture, the architecture can be easily be vertical and horizontal scaled by adding more nodes and resources to the design, hiding communication, distribution and replication latencies as well as providing reliability and performance.
When designing the architecture, we haven't covered active-active replication in the backend servers, which is a must for data consistency, and we haven't covered network reliability and homogeneity which is necessary to keep the scalability consistent. What we did cover is network security and the WAF's configuration in order to satisfy the need of high availability.

With this in mind, scalability is one of the most important goals to achieve when building a high WAF, since a lot of requests are going hit the architecture.
With WAF_v1, we have understood how we can deploy a web service on Docker, and how we can take advantage of an intermediary node to mask the internal implementation of the web service's backend, increasing security. The downside of using WAF_v1 for a high availability front-end is that in certain demands, the architecture will rapidly become a bottle neck to the web service and will probably fail.

WAF_v2 is the start of a high availability configuration, composed by two load balancers and three backend servers, as the use case for the analysis of this report. When referring to a start of the architecture, we refer that every companies service is different, with its different needs and goals. Depending on the use case, there must be a previous evaluation of what configuration would be more suitable taking in consideration the service's needs. After evaluating the use case, the idea is to configure WAF_v2 architecture to satisfy the needs of the use case service, by adding more nodes to the front end of the architecture, or by adding more resources to the load balancers, always to increase performance and high availability. Another advantage of using WAF_v2 architecture is that depending on the use case the architecture is built, the load balancers can be configured to prevent denial of service attacks by limiting the rate of HTTP requests to the architecture, the load balancers can be configured to support SSL termination, bot detection and IP masking and different load balancing algorithms can be configurated to load balance the requests. The configuration can be found in the haproxy.cfg file which is the master configuration file for the configuration of the load balancers.

The actual state of evolution from the project is that the initial definition of objectives and requirements, and the analysis of the building tools to develop the project have been accomplished. A running scalable client-server web application architecture with front-end high availability, data persistence, network design, traffic load balancing and traffic analysis have been accomplished.

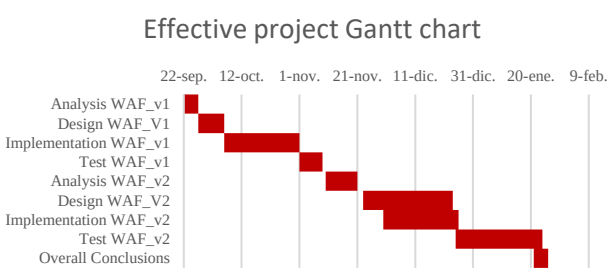

Figure 5 shows the Gantt chart diagram from the current state of the project from this work.

Every project development encounters some issues that can delay the execution of a project. In the case of this project, since there is a time limitation of 300 hours, and the development of WAF_V2 has lasted more time than expected, the project has seen changes which will modify the initial requirements. In comparison from the ideal project Gantt chart, the development of WAF_V2 has taken more time because of some changes and understanding that had to be done to the design of the architecture. Considering the comparison between the ideal project development and the effective and current development from the project, the changes that have occurred are listed below. In figure 7 from the appendix, there is detailed the different specifications from the Gantt chart above. It is important to note that the tasks listed that won't be accomplished due to time restrictions and the limitation time, will be done in future work.

- The inclusion of Apache ModSecurity as a building tool for the monitorisation of traffic load,
- The inclusion of SecurityOnion which can be used for real-time web application monitoring and logging
- The development of WAF_V3, consisting of the same architecture as WAF_V2 but with the integration of the Elastic stack to the project and the development of a Kibana module to monitor logs and statistics.

The final architecture's development is completed, and testing is in progress. In parallel with the development and analysis of the software, the documentation and justification of achievements are in progress. Number wise, the project is at 85% from its completion. In the next progress report, exhaustive project tests and their analysis to the current state of the project will be conducted in order to give the last conclusions and complete this work.

REFERENCES

[1] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[2] «Prototyping Model in Software Engineering: Methodology, Process, Approach». Reviewed 16th of September of 2019. https://www.guru99.com/software-engineering-prototyping-model.html.

[3] Repository of Hernan's Espinosa Work. https://github.com/HernanEspinosa/Web_Application_Firewall.git. Last Acces: October 2, 2019.

[4] Hat, R. (2019). *¿Qué es Docker?*. [online] Redhat.com. Available at: https://www.redhat.com/es/topics/containers/what-is-docker [Accessed 22 Sept. 2019].

[5] "What is a Container? | Docker", *Docker*, 2019. [Online]. Available: https://www.docker.com/resources/what-container. [Accessed: 15 Sept- 2019]

[6] «Productos de Elastic: Búsqueda, analíticas, logging y seguridad | Elastic». Reviewed 10th of September of 2019. https://www.elastic.co/es/products/.

[7] Docker Documentation. «Manage Data in Docker», Reviewed the 12th of October 2019. Available: https://docs.docker.com/storage/.

[8] "What is a VRRP? | Cisco". 2019 [Online].Available at: https://www.cisco.com/c/es_mx/support/docs/security/vpn-3000-series-concentrators/7210-vrrp.pdf. Reveiwed the 7th of November of 2019..

APPENDIX

Figure 1:

The table below recruits all the information concerning the initiation and finalization of the different phases of the project. The information detailed corresponds to the Gantt diagram located in the methodology and planning section of the report.

TASKS	RESPONSIBLE	START DATE	END DATE	Duration	STATUS
Analysis WAF_V1	Hernan Espinosa	22-Sep-19	27-Sep-19	5 days	**Not Started**
Design WAF_v1	Hernan Espinosa	27-Sep-19	06-Oct-19	9 days	**Not Started**
Implementation WAF_V1	Hernan Espinosa	06-Oct-19	01-Nov-19	26 days	**Not Started**
Test WAF_V1	Hernan Espinosa	02-Nov-19	10-Nov-19	8 days	**Not Started**
Analysis WAF_V2	Hernan Espinosa	11-Nov-19	22-Nov-19	11 days	**Not Started**
Design WAF_v2	Hernan Espinosa	23-Nov-19	29-Nov-19	6 days	**Not Started**
Implementation WAF_V2	Hernan Espinosa	30-Nov-19	15-Dec-19	15 days	**Not Started**
Test WAF_V2	Hernan Espinosa	15-Dec-19	22-Dec-19	7 days	**Not Started**
Analysis WAF_V3	Hernan Espinosa	23-Dec-19	31-Dec-19	8 days	**Not Started**
Design WAF_V3	Hernan Espinosa	01-Jan-20	10-Jan-20	9 days	**Not Started**
Implementation WAF_V3	Hernan Espinosa	11-Jan-20	20-Jan-20	9 days	**Not Started**
Test WAF_V3	Hernan Espinosa	20-Jan-20	21-Jan-20	1 days	**Not Started**
Overall Conclusions	Hernan Espinosa	21-Jan-20	26-Jan-20	5 days	**Not Started**
LAUNCH		26-Jan-20	27-Jan-20	1 days	

Figure 2:

The Gantt chart below recruits the details of progress deliverables, progress documentation and development from the project.

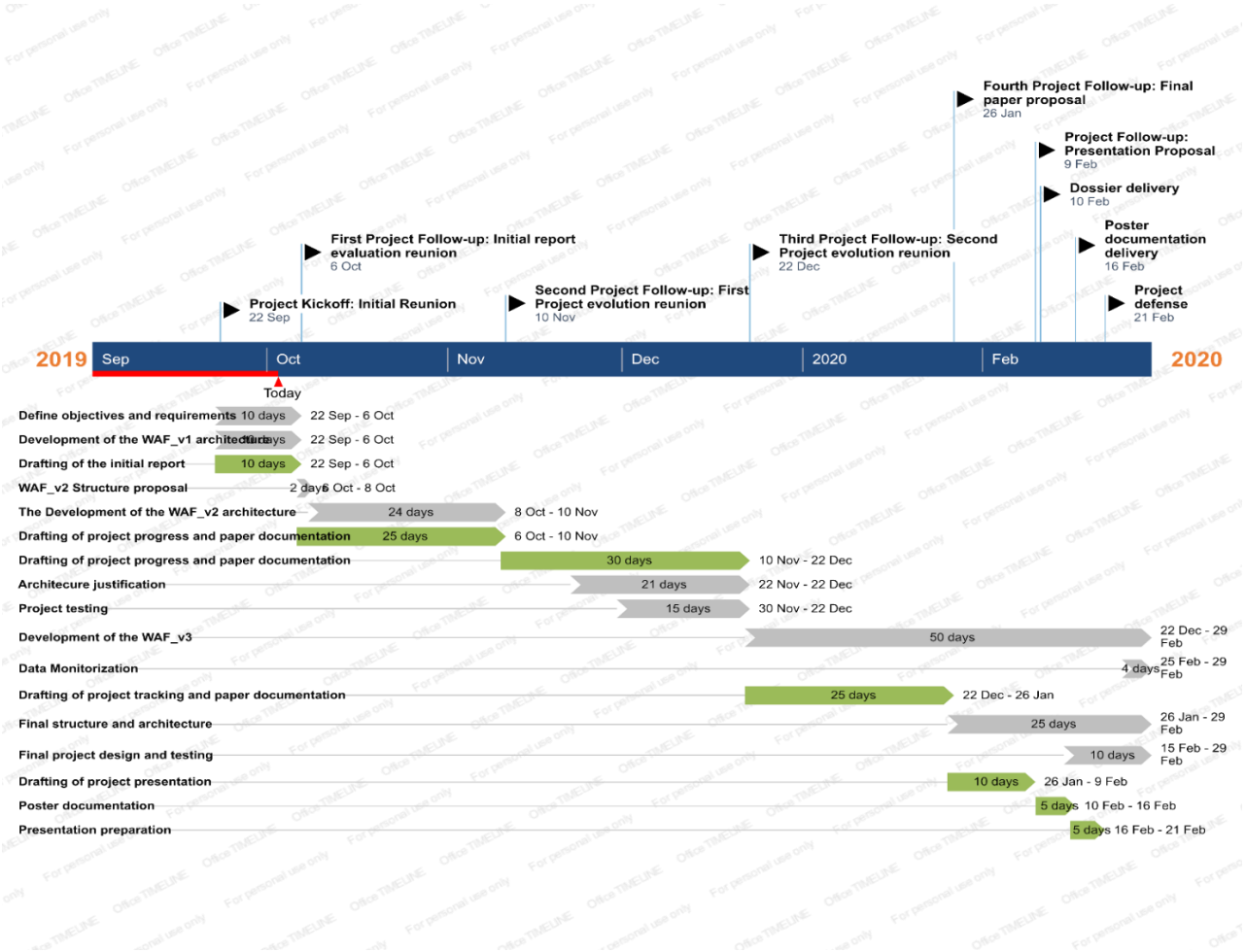

FIGURE 3:

The information below composes the details of the test conducted to WAF_V1 architecture.

root@Hernan:~/Documentos/Web_Application_Firewall/WAF_v2_Progression/WAF_v2.2/loadbalancers/loadbalancer1# ab -n 1 http://172.34.2.1:8080/index.html
This is ApacheBench, Version 2.3 <$Revision: 1807734 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/
Benchmarking 172.34.2.1 (be patient).....done
Server Software: Apache/2.4.38
Server Hostname: 172.34.2.1
Server Port: 8080
Document Path: /index.html
Document Length: 45 bytes

Concurrency Level: 1
Time taken for tests: 0.001 seconds
Complete requests: 1
Failed requests: 0
Total transferred: 291 bytes
HTML transferred: 45 bytes
Requests per second: 1082.25 [#/sec] (mean)

```
Time per request:       0.924 [ms] (mean)
Time per request:       0.924 [ms] (mean, across all concurrent requests)
Transfer rate:          307.55 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.0      0       0
Processing:     1    1   0.0      1       1
Waiting:        0    0   0.0      0       0
Total:          1    1   0.0      1       1
```

FIGURE 4:

The information below recruits the details from the logs from the test conducted to WAF_V1 architecture.

```
Dec 17 09:11:54 intermediary haproxy[50]:
Hostname: intermediary
Client-IP: 172.34.0.1
Client-Port: 45326
Server-Name: backendserver_vaf_v1
Server-IP: 172.34.2.2
Server-Port: 80
Date-Time: 17/Dec/2019:09:11:54.340
HTTP-Request: GET /index.html HTTP/1.0
Frontend-IP: 172.34.2.1
Frontend-Port: 8080\
```

FIGURE 5:

The information below recruits the details of the test conducted to WAF_V2 architecture.

```
root@Hernan:~/Documentos/Web_Application_Firewall/WAF_v2_Progression/WAF_v2.2/loadbalancers/loadbalancer1# ab
-n 3 http://172.34.1.2:8080/index.php
This is ApacheBench, Version 2.3 <$Revision: 1807734 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 172.34.1.2 (be patient).....done


Server Software:        Apache/2.4.38
Server Hostname:        172.34.1.2
Server Port:            8080

Document Path:          /index.php
Document Length:        454 bytes

Concurrency Level:      1
Time taken for tests:   0.016 seconds
Complete requests:      3
Failed requests:        2
   (Connect: 0, Receive: 0, Length: 2, Exceptions: 0)
Total transferred:      1976 bytes
HTML transferred:       1328 bytes
Requests per second:    188.44 [#/sec] (mean)
Time per request:       5.307 [ms] (mean)
Time per request:       5.307 [ms] (mean, across all concurrent requests)
Transfer rate:          121.21 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.1      0       0
```

Processing:	1	5	7.0	7	13
Waiting:	0	0	0.0	0	0
Total:	1	5	7.0	7	13

Percentage of the requests served within a certain time (ms)
50% 1
66% 1
75% 13
80% 13
90% 13
95% 13
98% 13
99% 13
100% 13 (longest request)

FIGURE 6:

The information below recruits the details from the logs from the test conducted to WAF_V2 architecture.

Request log for BackendServer1:

Dec 17 13:05:35 loadbalancer1 haproxy[11620]:

Hostname: loadbalancer1
Client-IP: 172.34.0.1
Client-Port: 60788
Server-Name: backendserver_1
Server-IP: 172.34.1.6
Server-Port: 80
Date-Time: 17/Dec/2019:13:05:35.855
HTTP-Request: GET /index.php HTTP/1.0
Frontend-IP: 172.34.1.2 Frontend-Port: 8080\

Request log for BackendServer2:

Dec 17 13:05:35 loadbalancer1 haproxy[11620]:
Hostname: loadbalancer1
Client-IP: 172.34.0.1
Client-Port: 60792
Server-Name: backendserver_2
Server-IP: 172.34.1.7
Server-Port: 80
Date-Time: 17/Dec/2019:13:05:35.856
HTTP-Request: GET /index.php HTTP/1.0
Frontend-IP: 172.34.1.2 Frontend-Port: 8080\

Request log for BackendServer3:

Dec 17 13:05:35 loadbalancer1 haproxy[11620]:
Hostname: loadbalancer1
Client-IP: 172.34.0.1
Client-Port: 60796
Server-Name: backendserver_3
Server-IP: 172.34.1.8
Server-Port: 80
Date-Time: 17/Dec/2019:13:05:35.857
HTTP-Request: GET /index.php HTTP/1.0
Frontend-IP: 172.34.1.2
Frontend-Port: 8080\

Figure 7:

TASKS	RESPONSIBLE	START DATE	END DATE	Duration	STATUS
Analysis WAF_V1	Hernan Espinosa	22-Sep-19	27-Sep-19	5 days	**Completed**
Design WAF_v1	Hernan Espinosa	27-Sep-19	06-Oct-19	9 days	**Completed**
Implementation WAF_V1	Hernan Espinosa	06-Oct-19	01-Nov-19	26 days	**Completed**
Test WAF_V1	Hernan Espinosa	02-Nov-19	10-Nov-19	8 days	**Completed**
Analysis WAF_V2	Hernan Espinosa	11-Nov-19	22-Nov-19	11 days	**Completed**
Design WAF_v2	Hernan Espinosa	23-Nov-19	23-Dec-19	31 days	**Completed**
Implementation WAF_V2	Hernan Espinosa	30-Nov-19	25-Dec-19	26 days	**Completed**
Test WAF_V2	Hernan Espinosa	25-Dec-19	26-Jan-20	30 days	**In progress**
Analysis WAF_V3	Hernan Espinosa	27-Jan-20	31-Dec-19	-	**Delayed**
Design WAF_V3	Hernan Espinosa	27-Jan-20	10-Jan-20	-	**Delayed**
Implementation WAF_V3	Hernan Espinosa	27-Jan-20	20-Jan-20	-	**Delayed**
Test WAF_V3	Hernan Espinosa	27-Jan-20	21-Jan-20	-	**Delayed**
Overall Conclusions	Hernan Espinosa	21-Jan-20	26-Jan-20	5 days	**In progress**
LAUNCH		26-Jan-20	27-Jan-20	1 days	