Master Thesis

Master in Research and Innovation in Informatics: FIB Facultat d'Informatica de Barcelona
Computer Engineering: Politecnico di Torino

Politecnico
di Torino

Analysis and Implementation of Load Balancers in Real-Time Bidding

Author: *Pietro Oricco*

Supervisor: *Edraí Brosa Ciercoles*
FIB Tutor: *Francisco Javier Larrosa Bondia*
Politecnico di Torino Tutor: *Andrea Bottino*

January 2022

Contents

1 ABSTRACT

This document is meant to analyze the main features of a **Real-Time-Bidding (RTB)** system with a view to improve the functionality of a **Demand Side Platform DSP**. It reflects on the best way to implement a software component which defines the **balancer** module, which is a specific module meant to spread the traffic a web-platform receives over multiple back-end servers. In particular, the discussion will be centered on which load-balancing strategy and tool is the best by the point of view of a high-demand throughput system in order to avoid the overload of some compute nodes, considering that many *open-source load-balancers* can be found in the market in a great variety of forms, implementations and features; the focus will be over the needs of a *Demand Side Platform*, where performances are put at first place and the internals of the platform itself change constantly (such as the number of servers and the addresses of the servers itself).
This research will be conducted following best-practices in Software Engineering and Research field, with the purpose to aggregate the various learning contributions gathered during my Double Degree experience among Barcelona and Torino.
First, a background over the topic is provided, with a glance to the *RTB* world and the main concept that this kind of system deploys and an insight over the internals of the *balancer* component by means of proxy models and load-balancing strategies.
Second, a preliminary research over the main software solutions will be conducted, with the aim of filtering the ones that don't match the requirements provided by a professional tech company; the documentation supplied by each *balancer* will be analyzed with the objective to fill a **software evaluation matrix**, provided to highlight the various feature supplied by each balancer and to discard faulty solutions.
Then, a testing environment will be built for every solution still under evaluation in order to effectively check that the component respects the declared features. Moreover, the testing environment is exploited to discover which is the best software product by means of overall performances, requirement considered crucial for a **low-latency-high-throughput platform**; the final goal of this step is to provide a winner to the software selection process that will be implemented in the final step by means of stressing the limit of the softwares under evaluation both by means of incoming total connections and requests per second.
Finally, the ultimate candidate will be implemented inside the platform environment: it will be installed and configured over the *Infrastructure as a Service* that hosts the *Demand Side Platform* environment, mapping the agents described later in the discussion with the actual final component's configuration file. The configuration file must be meant as a template that describes the current back-end members for each server farm, which will change aside the internals of the platform itself, hence a way to configure dynamically the component will be described, together with the adjustments meant to integrate the *balancer* module in the platform and deploy it in real production environment.
In conclusion, the final goal is to observe the effects that this analysis and the consequent

implementation over the production environment metrics will cause, with the objective to improve the quality of service of the back-end by reducing the average response times from servers side and to show a possible decrease of the infrastructure costs related to the bidding process of a *DSP*. A **dashboard** will be provided in order to highlight the possible improvements this project might carry.
In addition, the aim of this thesis is to provide useful guide-lines for future works around the topic and also to highlight best-procedures for the selection and the implementation of the balancer software component, with a view to improve the overall performances based on real-case scenario requirements.

2 INTRODUCTION AND MOTIVATION

This Master Thesis has been driven by the will of testing my skills over the expectation of the labour market, with the aim of improving my abilities and sample the working requirements of a real software engineering company. This comes aside the will of enhancing the job of the so called *DevOps*, a collection of practices meant to develop and to operate over a software product, with a glimpse to the world of *AdTech* and the management of a *HTTP-level platform* involved in *RTB*, the new frontier of programmatic advertisement; in addition, this paper has been written during my period as member of the infrastructure team in *Smadex SLU* (also addressed by the Company in the next chapters), where I had the opportunity to work on their platform and to investigate over the challenging topic of the selection and implementation of a **reverse-proxy**, a fundamental component inside the system with the duty of managing the HTTP traffic and load-balancing the various servers working as part of *Smadex SLU* platform. I have always been fascinated by the concept of *AdTech* and working directly in the field has been an extremely fulfilling and valuable experience, exploiting and investigating in many aspects of the infrastructure of a so-called **Demand Side Platform** or simply **DSP**.
In this paper, I am going to inspect the market offer for **reverse-proxies** with the aim of selecting the best product to implement as the **balancer component** in a production environment basing the decision on real-case requirements given by a professional engineering company that operates over the development and the management of such-a-kind of platform. The analysis in this paper takes into consideration that many products that offer reverse-proxy features in the market are designed and developed following several standards, implementation logics and programming languages (**affecting the overall load distributions among servers and the respective response time latencies**), therefore they provide different functionalities and guarantee different performances which are considered a crucial indicator in such a platform like the one deployed by *Smadex SLU*, that is for all intents and purposes a low-latency-high-throughput system that needs to operate at the best performances possible over a **Cloud infrastructure**.
In addition, this paper is based on the analysis made by the engineer and expert of the field John Hearn [1] around the load-balancing algorithms subject, that analyse the disparate strategies around the topic, which outcome is considered extremely valuable by means of the discussion held in the next pages, in particular during the selection of the component and the successive evaluation.

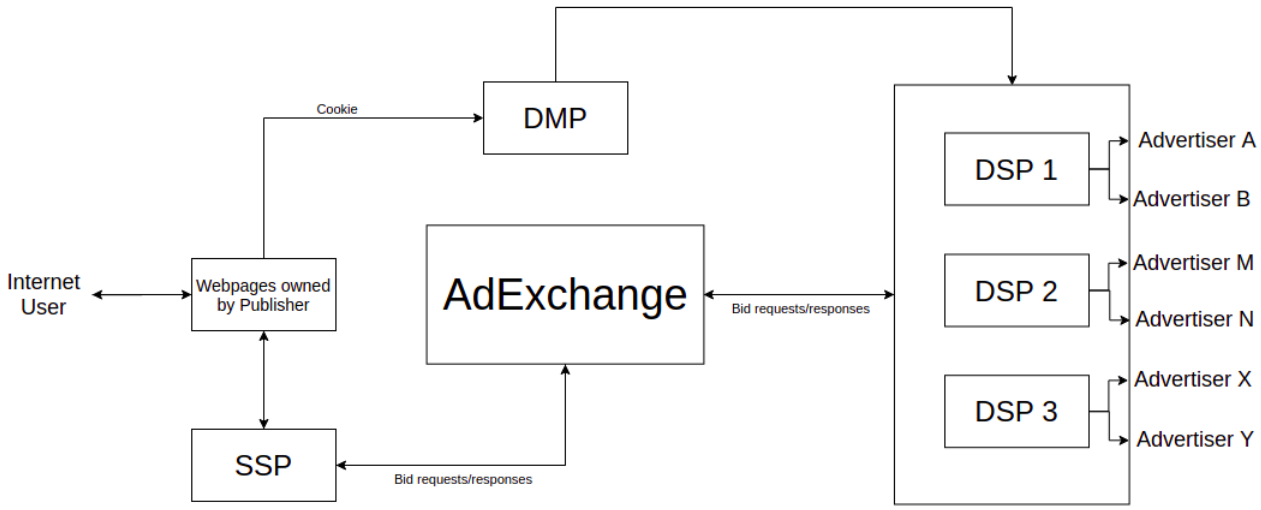
3 PRELIMINARIES

In this chapter, we will analyse the founding concepts behind this Master Thesis. First, we will overview **OpenRTB** [3], which is the standard that defines the way the platform operates, highlighting its structure by means of agents involved and general concepts like **auction and bids** related to the standard.
Then we will dig into the **reverse-proxy** topic, listing the types and the layers they operate, analysing the several methodologies provided by means of **load-balancing** and their role inside the Real-Time-Bidding context.

3.1 Overview of a Real-Time-Bidding (RTB) system and OpenRTB

Have you ever wondered how the data we share while surfing the web are managed in order to have always the most catchy advertisement on our devices? And how can the best-fitting commercial be provided in a time window of few milliseconds? The answer is called *RTB*, in other terms **Real-Time Bidding**, which is an "emerging and promising business model for online computational advertising" [2], formaly stated by **OpenRTB** [3], a networking standard that takes place at the *Application layer of OSI model* [4] and defines a way to serve ads to consumers directly based on their demographic, geographic, psychographic or behavioral attributes.
The creation of this particular standard has been made fundamental since the requirements of the market. Once upon a time, web-publishers had to negotiate directly the banner placement over an application or a website with advertisement providers, this was typically an effort and time consuming task that required many manual operations and maintenance. Alongside the escalation of the digital world and the relative increase of web-pages, web-users and the relative data stored in form of cookie, the birth of a new standard for advertisement delivery was demanded.
With the rise of programmatic advertisement and *Real-Time Bidding*, the advertisement placement have become automatized thanks to a set of algorithms and hardwares that provide the intermediary infrastructure for the web-publishers to expose their commercial spaces and for the advertisers to set up advertisement campaigns. In particular, as expressed by [29] this kind of system pattern, works thanks to separate and distinct but complementary sides: the **Demand Side Platform (DSP)** and the **Supply Side Platform (SSP)** interconnected by the intermediary figure of the **AdExchange**. Typically the *SSP* connects to the web-publishers side, providing an interface for the publisher to handle the *advertisement inventory* (a list of the banner available over a web-site) and monitor *advertisement impressions*; in other terms, it is the way of *RTB* to gather many forms of advertising demand.
On the other hand, *DSP* interfaces the advertisers that wants to buy inventories, allowing

publicists to manage the content of their advertisements (also called *Creatives*) and to access a wide range of statistics over the commercial campaign, placing at the disposal of advertisers an impressive reach of web traffic acquisition making this system extremely efficient by means of banner purchases; in addition, *DSP*s offer many features extremely valuable for targeting audiences at different levels, such as geography and nationality, device and OS, browser and demographic data (such as age or gender)[29]. These information are sampled directly from audiences exploiting **cookies**, small chunks of data describing web-page users' routines and features, deal by means of a third-party agent called **Data Management Platform (DMP)**.



Summarising, this business model defines a significant **transformative innovation in online advertising** market, providing an efficient and big-data driven way to deliver advertisements, significantly increasing the precision and effectiveness of advertisement shipment technologies. Reporting what described before, *RTB* introduces:

- Standardized management of banners and improved advertisement placement.
- Improved targeting, advertisements extracted *Ad-Hoc* from user to user.

3.2 Auction and bids

It is self-evident from the naming of this particular business model that it is dealing with **bids**, making it similar to an **auction-style** sell, where the bids are executed real-time, in a time-span that usually do not exceed a few hundreds of milliseconds. This auction takes place during the load of a web page or the boot of a mobile app, during which a massive

amount of micro-data are generated. Those micro-data are commonly called **bids** and they can be described as the dialing of **HTTP POST** requests and responses that carries all the information to elect the winner of the auction inside the HTTP packet body (usually in json format), defining a way to exchange information among *SSPs* and *DSPs* in order to guarantee banners placement. But how does this process work? As defined by [2], the following are the usual steps that happen in an *OpenRTB* auction (addressed later also as *bidding process*):

1. A web-surfer visits a web-page or a mobile app owned by a publisher registered into a *SSP*.

2. If the web-page hosts one or multiple ad spaces, the publisher's page or mobile app sends a **bid request** to the *AdExchange*, providing information about the user, the type of commercial banner (video, image, game, ...) and the relative price for its acquisition, asking for the start of an auction.

3. The *AdExchange* samples the available *DSPs* and forwards the bid-request.

4. Each *DSP* then retrieve the information calling a **DMP** which provides all the needed details for targeting the user with the right advertisement. Machine Learning algorithms are fed with the data coming from cookie history and the best-fit for the user is extracted among the advertisers and used as participant at the *auction*. The bidding price is calculated usually by a software module that is in charge of managing the budget of the advertiser for the given campaign. This step is crucial, *DSP*s must provide a bid-response for the best-fit in a certain amount of time to avoid being cut off from the *RTB* process.

5. The *AdExchange* starts the auction and collects all the **bid-responses** coming from various *DSPs*; then it evaluates the bids, determines the highest one and notifies all the agents involved in the auction with the winner of the process.

As you can notice, point 4 is extremely crucial for a *DSP*, which revenues are related to the number of auction in which is involved and, more in general, to the **number of requests per second it is able to process**; therefore the revenue for such a platform highly depends on the reliability and performances of the *DSP*'s infrastructure itself, which must be able to optimize the amount of requests per second processed and the response latencies, coming from **AdExchanges**.
Moreover, the amount of data generated aggregating all the auctions a *DSP* is involved into is huge, on the order of magnitude of millions of HTTP requests per second, hence what matters the most for such-a-kind of platform is to deliver a fast and reliable platform capable of responding to bulks of HTTP POST requests coming from the outer world.
Following, the *RTB Auction* is represented in form of Business Process Modelling diagram.

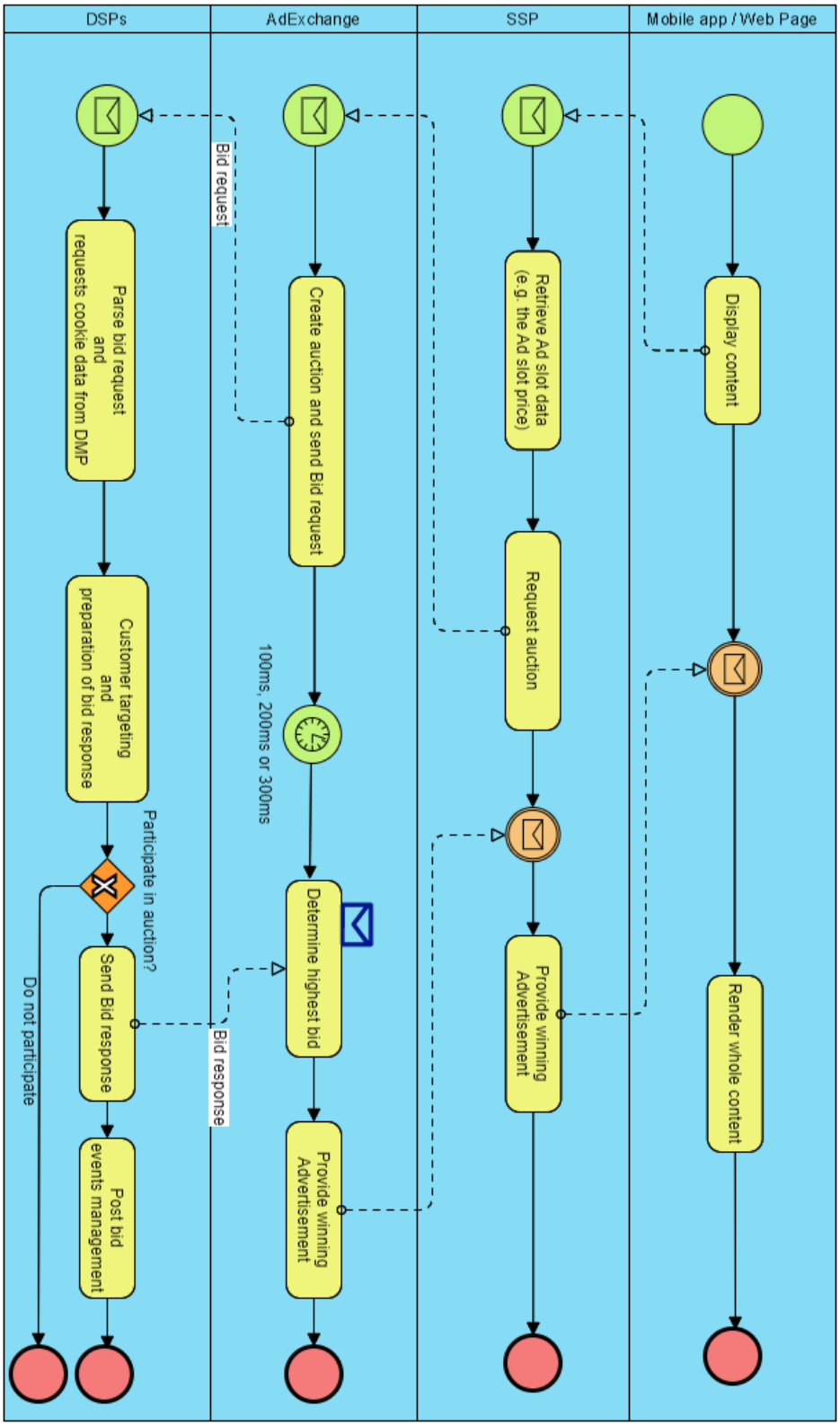
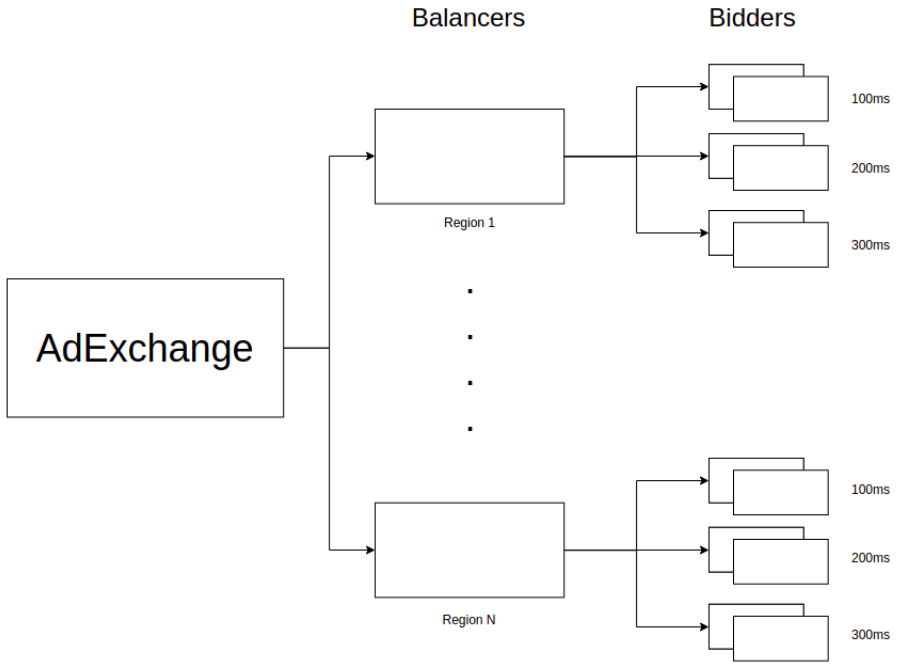


Figure 3.1: RTB auction business process model

3.3 Demand Side Platform DSP internals

As already discussed, *DSPs* main functionality is to retrieve the best fit for an ad slot related to a web-page user and to provide a response able to participate to the auction set up by the AdExchange; but how is this platform designed and connected to the outer world (i.e. *AdExchanges*)? Let's analyse the structure of the Platform taking as reference the one deployed by the Company:



Bidders are the instances in charge of receiving and forwarding bids, they cover the main role of the platform, the **bidding process**, and their main functionality is to provide a correct and in-time response to the given bid-request. They interact with the other modules of the platform in order to always have the right data in order to take part in the auction. Therefore, the bids are forwarded from the exchange to the platform following the above data-flow, with the intermediary step of the **balancer**; each *AdExchange* provides a maximum response time, hence the bidders are split in farms, grouped by the provided delay in order to match the exchange expectation. Usually, *AdExchanges* are not connected directly to the platform but they interface to the platform through balancers. This particular component is the center of the discussion in the next chapters, since it will affect directly the performances and the quality of service that the platform handles; it is implemented by means of a **reverse-proxy** and this concept will be analyzed later in the discussion. *DSPs* are founding members of the *RTB* process and with the rise of *cloud computing*, the new trend is to implement them over a **Cloud Infrastructure as a Service** (IaaS), following a geographical distribution over several regions in the world; they are placed in the most convenient areas of the globe, near

by the main *AdExchanges* locations, in order to decrease the overall round trip time of the HTTP packets. *DSPs* also provide other functionality related to the *RTB* process, such as tracking post-bid events (for instance, clicks and impressions grouped by campaign in order to provide important metrics to the advertisers) and managing the budget defined by each advertiser in each campaign. Those concepts won't be addressed in this paper, where the focus will be over the bidding process and the role of balancers inside it.

3.4 Reverse proxies and a possible use case

The term **reverse proxy** is usually given to a service that sits in front of one or more servers (such as the bidders in our scenario), accepting requests from clients for resources or processing power located on/served by the servers' backend side; these resources are then delivered to the client, appearing as if they have been originated from the reverse proxy server itself. But is there a reason for the term **reverse**? Ordinary proxies, also called **forward proxies**, are services exploited by a client or a group of clients in order to provide a common entry-point to a public network, in such a way that all the traffic generated by clients is forwarded to a single node in the network configuration: the proxy protects your inside network by hiding the actual clients IP address and using its own instead. This solution was very popular to provide Internet access before *Network-Address-Translation NAT* rise, that became the de-facto standard for this kind of use-cases.
On the other hand, reverse proxies act differently: they are normally implemented on the back-end side (i.e. interfacing with servers instead than clients) and, as expressed by [6], "a Reverse Proxy proxies on behalf of the backend HTTP server not on behalf the outside clients request, hence the term reverse". From the outside clients point of view, the Reverse Proxy is the actual HTTP server". Moreover, as expressed by the job of [5], it is possible to define common patterns of usage for reverse proxies use-cases. In particular, the author states a model considered peculiarly pertinent to the "server-shading" scenario:

- **Integration Reverse Proxy**: the author here investigates over some doubts that rise in a possible real use-case, such as how do you ensure everything is set under a consistent web application space *without exposing the server topology to clients*? Or even, how do you guarantee flexibility in network topology, by *adding or removing servers without surprising users*? Furthermore how do you *provide load balancing*, if an application server gets overloaded?
  A web site or a platform consisting of several web servers or applications can frequently change its network topology, therefore a solution needs to support changes in machine configuration in order not to break clients bookmarks, links or DNS entries to the given resource. Moreover, servers' load can highly differ, hence the reverse proxy configuration must provide a load balancing implementation able to distribute the traffic among the several back-end agents. The usage of a reverse proxy for integrating all the web servers as backend servers with a common host address (that of the reverse proxy) is then required, providing the ability to map URL paths under the common host address and to access a single specific back-end host as routing rule, guaranteeing the possibility

to provide the Integration Reverse Proxy with a TLS/SSL certificate for encrypted communications channels.

The **Integration Reverse Proxy** pattern provide many hints to light up the discussion since it is a good mapping to the use-case of a reverse proxy for a Demand Side Platform like the one deployed by the Company: bidders composing back-end farms might vary in number, depending on the load of requests per second they need to process in the given geographical region and defining an extremely volatile topology that it is really open to changes: for instance, a region can highly vary its number of bidders from day time to night time and vice-versa, but *AdExchanges* interfacing with them must always experience a stable quality of service. Reverse proxy in this use-case must guarantee a complete shading of the back-end, setting the internals of the platform as a black-box for the outer world while ensuring reach of access to the respective back-end farm for each AdExchange. Moreover, load balancing strategies offer a good topic for the optimization of the respective back-end, rising promising discussion point analysed in the next section.
In conclusion, this particular pattern described by [5] provide the right theoretical basis for the composition of requirements for the reverse proxies' software selection and it will be dig deeply later, in the section dedicated to software requirements.

3.4.1 Benefits of a reverse proxy

Reverse proxies help web applications to deploy a solid and stable entry-point for web services and applications which can be summarized in three points which are redundancy, performance and security:

- **Redundancy**: *reverse proxies* help back-ends to increase the availability of hosted servers, splitting the network traffic among them with the objective to reduce the overhead of multiple requests/connections managed by a single entity; this means that a multiple replicas of a server must be there listening to incoming proxyed traffic, grouped in a cluster of redundant instances of the same type. This feature is incredibly helpful, since it shades the front-end side to server failures. Server failure occurrences in a non-reverse-proxyed scenario would require the manual change of DNS value or routing the incoming packets to a different server, the presence of a reverse proxy avoids this tedious operation.
  Redundancy improves server availability, increasing widely the possibility to have a healthy server instance ready to process an incoming request.

- **Performance**: *reverse proxies* serving requests/connections act like the cashiers of a shop. Imagine a crowded scenario with many people waiting to be served in front of a single cash, if every person will take some time to be served, congestion will occur; let's say that after some minutes other three cashes open, setting the overall amount to 4, and a fifth employee start sitting in front of the cashes equally distributing customers among them, the performances will rapidly increase. Reverse proxies cover this case, sitting in front of servers and splitting, according to a **load balancing algorithm** (described

later in this chapter), the network traffic among back-end agents; in a scenario where servers implement a logic through algorithms and programming languages that drain resources and add an overhead to the serving time (meant as round trip time of a packet + processing time), having a reverse proxy able to distribute the load fairly would make the overall quality of service better.

- **Security**: networking also means protecting from malicious attackers and intruders. Web servers might suffer of various forms of attacks, that ranges from man-in-the-middle attacks to denial-of-service ones, which aim to sniff reserved information or block the Platform to serve incoming traffic, flooding the servers with a huge amount of requests and connections. Reverse proxies work as barrier to the outside world, providing some basic but efficient forms of protection against possible intrusions. **Distributed-Denial-of-Service (DDoS)**, **SlowLoris** and **Padding Oracle On Downgraded Legacy Encryption (POODLE)** attacks can be avoided by some modern *reverse proxies*, which furnish, in their configurations, methods to avoid these issues.

3.5 Load-balancing and RTB

A general definition of **load-balancing** is the process of traffic distribution among several computing units, where the traffic can be generally described by means of a set of *Tasks*. For our discussion purposes, it is important to address again that we are analysing the case of a server-side load-balancer that operates at HTTP level, since the *OpenRTB standard*, as already described previously, works over *Application Layer* of the OSI model; therefore each *task* corresponds to a **HTTP POST** request that needs to be processed by one of the available bidders in one of the back-end farms. In addition, the current implementation in the Platform provides the balancer component deployed on a cloud infrastructure, using multiple components with load balancing capabilities instead than a single centralized component, this strategy widely increases reliability through redundancy of the component itself. The current solution consists of virtual server instances inside an Auto Scaling Group over a Cloud Infrastructure (*Amazon Web Services AWS* specifically). Each instance has installed and configured *Nginx*, a specific software *reverse proxy and load balancer* solution that we will describe later as well, that routes the requests to the proper bidder farm, depending on the response time allowed by the exchange. Purpose of this paper is to understand if the current solution is the optimal among the available ones, considering that the component needs to respect the requirements provided by the Company but also be the one that exploits better and faster its back-end side.

3.5.1 Load-balancing algorithms

In general, there are two main distinctions of methodologies for load-balancing:

- **Static**: load balancing do not exploit back-end performances and information to make traffic partition decisions.

- **Dynamic**: algorithms that take into account the current state of each server and distribute traffic accordingly. This approach is more suitable for widely distributed systems such as cloud computing, since it is able to sample information from the back-end side and take load-balancing decisions according to the on-demand sampled data.

Many algorithms are described in the literature, the following are the most relevant ones by means of availability and recurrence among major implementations in the market:

- **Round Robin**: the most basic algorithm, it is a strategy that defines a FIFO ordering of the server called from the back-end. It is a static algorithm, hence it does not check the status of a server before forwarding a request. In case three server (called A, B and C) are available, the ordering of balance will always be A-B-C-A-B-C and so on, even if a server is extremely over-loaded. The *Nginx* based solution currently implemented in the platform exploits Round Robin.

- **Look-up table**: a static method that uses table to store the flow of data for a requests mapped with a given ID (such as IP address or URI). When the first packet of a stream arrives, a server is selected and the information is inserted into the lookup table, which allows to forward succeeding packets always to that same specific server. It suffers of memory problems, due to the fact that the table can be filled and some records must be deleted. The retrieval of a given entry can also become a problem for long look-up table, since the search of the entry can't be optimized more than a complexity of $O(\log n)$ (considering best-case scenario of binary search; in case of a normal search algorithm is applied, the complexity rise to linear).

- **Hashing**: another static algorithm featured by the exploit of an hashing function that maps the given request to a server. It is an optimization for the look-up table method that retrieve the relative server for a request with a constant complexity of $O(\log 1)$. The major hashing function works digesting the IP, the URI or a given HTTP header and provides the server to address in an efficient way.

- **Random**: the selection of the server is determined by a completely random generation scheme.

- **Least occupied**: a dynamic algorithm that checks the state of the given back-end and give access to the least occupied one. The most common schemes for determining the occupation of a farm are the count of requests or the count of connections each node is handling.

- **Power of Two** also called **Random 2**: it is the combination of two already described strategies: it mixes the logic of random and least occupied ones. When a request arrives to the balancer component, two distinct nodes are extracted randomly from a given backend, then, following a lest-occupied strategy, the request is forwarded to the node labeled as the less busy among the two.

3.5.2 Load-balancing algorithms' performances comparison

This section is dedicated to the comparison of the several algorithms expressed in the previous point, identifying those that might be relevant for optimizing the performances of each back-end farm, with a view on the best ones based on the Company and *RTB* point of view, where more specific requirements related to the given context are required.
Following the results obtained in the benchmark from HAProxy team [7], it is quite evident how *Round Robin and Random* algorithm do not furnish any kind of improvement, instead they are disastrous by means of performances; they are hence considered the most basic algorithms, which are not supplying any kind of improvement.
Then, we consider the *hashing* algorithm (the look-up table is not evaluated since a rudimental version of hash, hence less performant). *Hashing* is very efficient for static scenarios, where servers' state and number do not change along time, since it is able to provide, exploiting constant time, the required back-end and allocating server uniformly. The situation changes when a volatile back-end happens (i.e. when the scenario allows the removal and the add-on of servers; situation pretty common for a *cloud-based RTB use-case*), following the context described by [22], dynamic back-end can add a major problem which is the loss of table allocations that can get substantially changed, thus losing the benefits of previous caches.
Least-occupied strategies are the ones that usually fit the best for highly dynamic context like the Company's platform one since it can efficiently label the least busy server in a farm, but it adds an overhead related to the search of the best fit: looking for the least occupied can be time demanding if the given farm is wide; in addition, *Smadex's DSP* is linked with *AdExchanges* working with the most disparate types of connections and requests per second, there might be servers featured by a huge amount of requests per seconds on really few TCP connections making least-connection strategy to address servers already congested. In addition, when a server boots up, this kind of strategy tends to favor the establishment of connections with new born server, with the consequence to notice already congested servers few seconds after their instance creation. This is a scenario that must be avoided in order not to expose servers to bottle-necks.
Power of Two instead, provides a nice workaround for load-balancing algorithms, providing a solution able to cut off the demanding search over the servers' farm for the least occupied one. Moreover, following the research set up by my colleague at *Smadex SLU* John Hearn [1], it is evident how the Random 2 solution supplies a really efficient way of load-balancing, mixing the constant speed of retrieval of servers of the *Random* algorithm with the look-up to server status of the *least-occupied* one. As stated by [1] "while the least occupied is slightly better in terms of the spread of requests, the random 2 has some other advantages. Firstly, its slightly simpler and therefore faster in practice because only 2 servers are checked for each request rather than all of them" but also "it avoids servers which are (re)starting receiving all the load immediately". Moreover, John Hearn highlights how this strategy shows the best results for back-end load management, stating that it is the one that behaves best under varying load and

number of servers, keeping the response times lower than other algorithms. In addition, Random 2 helps to reduce the variances of the response times, guaranteeing better and more uniform reply from servers.
In conclusion, the best fit for the Company is then a **balancer component able to provide Random 2 balancing feature**, with the objective of providing a better distribution of load and uniform response times.

4 SOFTWARE SELECTION PROCESS

As proof of concept, the aim of this chapter is to focus over the selection of the best-fitting candidate for the balancer component of the system. In order to achieve this purpose, an analysis among the main solutions available in the market is done, considering those services which have reverse-proxy features and can efficiently load-balance many clusters of servers. The load-balancing service needs to be compatible with the requirements provided by the Company and with the concept explained in the preliminaries, especially for the Integration Reverse Proxy pattern and load balancing techniques, with a view to the one characterized by the best performances in terms of request-processing rate and management of back-end latencies; those requirements will be pointed out later in section 4.1, considering also that priority will be given to solutions based on free or open-source softwares available in the market. Many premium versions of the softwares under evaluation provide the full catalogue of expected requirements exploiting monthly or yearly billing. Since the selection of the balancer component should not affect the price that the Company is willing to pay, the evaluation must take into consideration *freemium* or *open-source* product with the aim of excluding the billing related to the component itself and, on the contrary, focusing if possible on the reduction of the price related to the *Infrastructure as a Service (IaaS)* hosted on the Company's cloud.
First, the candidates will be analyzed by the point of view of the proposed features; the documentation of the various softwares must drive this preliminar analysis with the objective of excluding such services that don't provide the expected specifications.
Second, the winners from the first stage will be configured and deployed over a testing environment exploiting Docker tools, aiming to prove that the product is effectively providing the declared features. The set-up will guarantee a fast and reliable deployment of multiple containers mocking the scenario of a *RTB auction*, with the aim of stressing the technology of each software load-balancer. In fact, the last stage of the experiment will research on the most efficient reverse-proxy in terms of overall performances, **benchmarking** the remaining candidates and providing the final winner.

4.1 Requirements of a RTB load-balancer

Following the needs of SMADEX SLU, we need to deliver a load-balancing software component capable of respecting the following requirements:

1. **Support HTTP/1.0, HTTP/1.1, HTTP/2 and HTTPS**: the new standards for *ISO-Layer 7* of networking, *HTTP/2* provides new features such as multiplexing (able to exploit a single TCP connection for multiple parallel requests), improved packets compression (able to eliminate a few bytes from each HTTP packet) and prioritization, that allows a better management of packets in HTTP connections. [18]

Moreover, HTTP/2 is defined both for HTTP URIs (i.e. without encryption) and for *HTTPS* URIs (i.e. Secure HTTP). Even if the standard itself does not require usage of encryption, all major browsers implementations (such as Firefox, Chrome, Safari, Opera, Edge) have stated that they will only support HTTP/2 over TLS, which makes encryption de facto mandatory [9], plus many *AdExchanges* exploit the encrypted HTTP standard to communicate with DSPs.
In addition, the balancer component is connected to *AdExchanges* exploiting *HTTP/2, HTTPS, HTTP/1.1 and HTTP/1.0* standards, while the internal connections among the reverse proxy and the bidders (both hosted on the same cloud infrastructure) are defined by the *HTTP/1.1* standard, making them mandatory compatibility requirements for the product selection.

2. **Support programmable routing**: software products under consideration must be able to provide configurable routing, allowing packets forwarding to predefined server farms (i.e. server collections aggregated by a common feature, like the maximum transmission time). Routing in networking is usually implemented by means of rules that work based on URL paths, hosts or HTTP header, meant to redirect packets to a set of endpoints. For instance, a component under evaluation must be able to address different backends forwarding packets to *entrypoint/specificFarm* (i.e. exploiting paths naming) or to *specificFarm.entrypoint* (i.e. exploiting hosts naming).

3. **Support configurable response code**: as already claimed, bidders are implemented in such a way that their number can vary according to geographical area and time-zone; moreover bidders might suffer problems and stop sending bid-responses or more simply, the bidder instance might crash for some infrastructural issues. **OpenRTB** standard [3] claims that a no-bidding response after a valid request must be labeled with **HTTP 204 No Content** by the bidder: the best-practice is to forward 204 code for each time a bidder is not participating to an auction (i.e. errors in the bidding back-end or, simply, the bidder is not interested to bid in that particular auction). This means that error codes for back-end side (such as *502 Bad Gateway, 503 Service Unavailable or 504 Gateway Timeout*) must be mapped into a HTTP 204 No Content in order to guarantee the correct behaviour of the overall system. Repeated reception of 5xx type error code would make the *AdExchanges* to close the connection to the *DSP*. In addition, following [3], every invalid call (e.g., a bid request containing a malformed or corrupt payload) must be mapped to 400 error code with no content.

4. **Support dynamic upstream farms with no-reload down-time**: load-balancing among servers can often be suffering of changes in the actors composing the server side. Servers that compose the back-end farms can suffer problems that make the given instance to enter a faulty state. A very common scenario in a *DSP* is to experience bidders not responding to requests (the so-called No-Bid state). From my practical experience on the field, the main reason for this issue to happen is a faulty execution of some "warm up" processes, the software modules in charge of loading all the information for the bidders in order to parse bid-requests and provide the given bid-response but in

general it is very common to have these kind of server issues due to multiple factors. Therefore, the component under analysis must be able to understand if a bidder is in a bad state and avoid forwarding requests to it. This process is usually controlled by **health checks**, which are described as a "request to each member of the load balancer group to establish the availability of each member server to accept" client's incoming packets [10].
On the other hand, back-end farms can experience bidders to join and leave the load balancing group due to more or less traffic to be served, hence the load balancer must recognize the group change and allow the packet routing to the new formed bidder group. The most challenging part is to provide a component able to dynamically update its configuration without experiencing a reload of the balancing service, which is considered a mandatory requirement in a low-latency-high-throughput platform where the back-end can constantly change: the Company' platform is able to handle an average of 1.5 millions of requests per second as aggregated statistics over multiple instances of balancers, with a server side framework that can change even multiple times per minute as worst case scenario, a reload of the service would cause down-time making thousands of connections and requests to be lost, not guaranteeing an optimal functioning of the platform itself; in addition, *AdExchanges* keep track of platforms response efficiency and a reloading behaviour with down-time would expose the platform to be cut-off from the RTB process; therefore, providing a service able to dynamically update the configuration without experiencing reload delay is mandatory for the functionality and the revenues of the platform itself.

5. **Support metrics retrieval**: in a company that performs *DevOps routines*, metrics are essential. They allows engineers to always have an overview about components status and functionality; "monitoring, at its heart, is about observing and determining the behavior of systems, often with an ulterior motive of determining correctness. Its purpose is to answer the ever-present question. Are my systems doing what they are supposed to?" [8]. The software product under evaluation must be able to expose metrics and allow the generation of statistics and plots in order to have a constant retrieval of information useful for the DevOps/Infrastructure team in charge of monitoring KPIs: "understanding the service your business provides and the levels at which you aim to deliver that service is the heart of monitoring" [8].

6. **Performances**: this requirement is the one considered most critical for the selection of the final candidate. The final choice must cover not only all the requirements previously stated but it must also be the best one in terms of overall performances. Specifically, SMADEX SLU Demand Side Platform manages over 52 billions HTTP requests every day with an average of 1.5 million HTTP requests per second aggregated over many bidding geographical regions distributed on a cloud infrastructure. Consider that *AdExchanges* place a very strict time constraint on the RTB process that usually is on the order of magnitude of hundreds of milliseconds, exceeding this time constraint means that the bids the DSP is sending are not taken under evaluation in the auction process, losing out on potential bids and being cut out of a sales channel because of consistently

slow bids. Being fast is mandatory in Ad-Tech world, hence maximizing the number of requests is considered crucial for having a correct behaviour of the platform.
This requirement will be evaluated by means of

- maximum requests per second that the component can handle;
- 99 percentile (i.e. the value of time that comprehends the 99 percent of the forwarded requests RTT round-trip-time).

7. **Load-balancing algorithm selection**: which load-balancing algorithms are offered by the candidate, prioritizing those that provide the best management of the back-end side load; specifically products explicitly offering *Random 2* as load-balancing strategies must be prioritized, since they might provide a significant improvement as analyzed previously in the discussion.

8. **Availability of documentation and material**: in order to properly configure each component, it is fundamental to work with a well documented software that presents all his features in a clear way. In addition, documentation is also needed in order to understand if a software is mature and enough supported to be considered valid for the installation in a production environment.

9. **Support configurable HTTP headers**: a minor way to reduce the impact of the HTTP requests on the network infrastructure. A service that provides this feature allows the removal and the modification of unused HTTP headers by means of RTB, which means less Bytes transmitted over the channels, hence an improved utilization of resources and costs of the infrastructure.

4.2 Available Candidates

In this section, I am going to point out the candidates that are taken into consideration as services for the implementation of the balancer component.
In particular, I am going to focus on the following software products:

- **HAProxy**: it is a "free, very fast and reliable solution offering high availability, load balancing, and proxying for TCP and HTTP-based applications. It is particularly suited for very high traffic web sites and powers quite a number of the world's most visited ones. Over the years it has become the de-facto standard opensource load balancer, is now shipped with most mainstream Linux distributions. Its mode of operation makes its integration into existing architectures very easy and riskless, while still offering the possibility not to expose fragile web servers to the net" [13]. It is written using *C* language, hence *HAProxy* can exploits the advantages of it, well known to be simple and efficient and to provide the best performances among the other programming languages, since it is very close to assembly and a lot of instructions are directly mapped to the machine level. Among the web site exploiting *HAProxy*, the majors are GitHub, Bitbucket, Stack Overflow, Reddit and Slack.

- **Traefik**: another mainstream solution in the market, Traefik provides a service able to dynamically change its configuration with ease, "with Traefik, there is no need to maintain and synchronize a separate configuration file: everything happens automatically, in real time (no restarts, no connection interruptions)". [15] This fact makes it an interesting solution in order to easily implement a balancer component not affected by reload delays, as expressed in the requirements section. This concept has been made possible thanks to the programming language used to implement the service, it is developed and supported through *Go* language, designed to improve programming productivity, especially for multi-threaded solutions. More than 180 company are currently reported to use *Traefik*.

- **Nginx**: its primary role was meant to be a web-server, therefore a service able to manage the access to multiple HTTP resources located in possibly multiple domains with the feature of being able to handle a huge number of concurrent connections. Since it can support a high volume of connections, "Nginx is commonly used as a reverse proxy and load-balancer to manage incoming traffic and distribute it to slower upstream servers anything from legacy database servers to microservices"[14]. It is the product that offers the widest set of features, furnishing many functionalities covering load-balancing and reverse proxy ones but also providing Mail proxy, caching and other characteristics typical of web-servers. It is coded through *C* language, exploiting all the benefits this language supplies by means of execution speed and multi-programming. It is the solution adopted by the Company, as expressed previously, at the current state of art.

- **Envoy**: Envoy is an open-source proxy operating at network's Application layer, that provide an efficient solution to address the more dynamic nature of a micro-services architecture, as opposed to the traditional applications that were mostly static. Envoy was made specifically for cloud architectures, and supports hot restart (no reload downtime) to keep a safe approach not to lose connections during a configuration change, and focuses on using the "xDS API" to manage this kind of runtime modifications. [16] Moreover, it exploits *C++*, guaranteeing similar execution speed as *C* but furnishing extra security checks providing a more improved solution by programming language point of view.

- **Apache Web Server**: the de-facto web-servers' standard for more than 25 years, written exploiting *C* programming language, it has been the most used solution in the market for HTTP services. It is featured by a complex architecture that exploits multi-processes, which allows Apache to run in either a "process-based mode, a hybrid (process and thread) mode, or an event-hybrid mode, in order to better match the demand of each specific infrastructure" [17]. Moreover, it is considered significantly slower than other solutions in the market and even more recent patches and updates have not solved its intrinsic performance problem [17].

4.3 Comparison against the requirements

Starting from the given base of candidates at 4.2, it is necessary to filter their features against the requirements reported previously. This research is made comparing the documentation of each product under analysis with the features provided by the component itself, with the objective of filtering out the less promising ones and proceeding **benchmarking** the performances, testing the remaining load-balancers.
The following table is meant to describe the meaning of each value in the matrix for the evaluation of the softwares, each row describes a requirement and the possible levels of contribution :

Requirement/Value	0	1	2
Support no reload dynamic configuration (PRIMARY)	Not supported	Supported (External Plugin)	Supported (Built-in)
Support HTTP/xx and HTTPS (PRIMARY)	Not supported	Supported	
Load balancing algorithm selection (PRIMARY)	Not configurable	Less performant ones available	Random 2 available
Support configurable response code (PRIMARY)	Not supported	Supported	
Support programmable routing (PRIMARY)	Not supported	Supported	
Support configurable HTTP headers (SECONDARY)	Not supported	Supported (External Plugin)	Supported (Built-in)
Support metrics retrieval (SECONDARY)	Not supported / Not exportable	Few available but exportable	Good collection available
Availability of documentation and material (SECONDARY)	Lack of documentation	Scarcely documented	Well documented

The evaluation has been driven following a weighted matrix model, where each necessary feature has a weight and a different range of values that can be assigned to it. The contribution of each candidate is calculated following a formula like

$$\sum_{i=0}^{N} w_i * val_i$$

where w_i is the weight and val_i is the value coupled to $requirement_i$; N is the overall number of requirements.

Every requirement has a weight assigned, the weight has been thought in order to respect a given range of values depending whether the specification is primary, i.e. addressed as crucial for the optimization and the implementation of the component, or secondary, defined in order to sharp the component selection and the experiment set-up.

- Primary requirements: spanning from a minimum of 0 to a maximum of 6 of contribution value. $0 \leq w_i * val_i \leq 6$
- Secondary requirements: spanning from a minimum of 0 to a maximum of 4 of contribution value. $0 \leq w_i * val_i \leq 4$

Following, the **software evaluation matrix** is reported:

Requirement	Weight	Nginx	Traefik	HAProxy	Envoy	Apache
Support dynamic configuration [0-2]	3	1	2	2	2	0
Support HTTP/xx and HTTPS [0-1]	6	1	1	1	1	1
Load balancing algorithm availability [0-2]	3	2	0	2	1	1
Support configurable response code [0-1]	6	1	1	1	1	1
Support programmable routing [0-1]	6	1	1	1	1	1
Support configurable HTTP headers [0-2]	2	2	1	2	2	2
Support metrics retrieval [0-2]	2	1	1	2	2	0
Availability of documentation and material [0-2]	2	2	2	2	1	2
		37	32	42	37	29

4.3.1 Considerations

HAProxy by now seems the most complete product, with a huge amount of built-in features provided in the basic release and scores the maximum in all the specifications, fully covering the Company's expectation and providing a good-enough set of load-balancing algorithms, including *Power of Two* algorithm. Moreover, through a feature called **Runtime API** [25], it allows to change values for members of each routing farm without down-time, making it the best viable products in the selection.
Nginx as well can be considered a good candidate at this step, providing almost every given requirement as built-in function while only the dynamic configuration of its upstream servers must be implemented by means of a plugin but, the metrics provided are quite scarse; on the other hand, as well as *HAProxy*, it provides an implementation for the **Random 2** method.
It is possible to notice the first exclusion based on the contribution calculation: from its documentation, *Apache Web Server* does not support two of the basic requirements provided, specifically the no-reload dynamic update and retrieval of metrics are not described in the documentation, hence this software must be excluded in the first stage of the analysis.
Traefik instead is a software capable of supplying a minimal amount of extra features (only *Round Robin* algorithm available and support configurable headers only by means of a plugin) but provides a fast-to-learn and easy-to-configure solution, considering the ease of exploiting its no-reloading nature and neglected down-time. Unfortunately, for the use-case covered in

this paper and as discussed in the section dedicated to Load Balancing methods, *Traefik's Round Robin* does not provide any advantage for the optimization of the back-end servers' load distribution, therefore is value-less and not convenient to proceed with this solution.
A different kind of discussion involves *Envoy* proxy; it is evident how this product is complete and covers all the specifications, ranking second only to *HAProxy* on a par with *Nginx*, showing interesting features and a really rich set of load-balancing algorithm, including the *least-request* algorithm (a version of the least-occupied one but that takes into consideration the amount of requests) and a plentiful collection of metrics to expose. The real problem with *Envoy* results in the documentation, especially for a feature like the dynamic upstream configuration which is badly documented at the point that the effort spent in researching how to build-up the feature is overcoming the actual benefit this software can bring. Ease of use is an implicit requirement that must be taken into consideration for a software selection process, badly documented products or features can lead to difficulties that might overwhelm the actual work or the future updates of the component.
For all the reason explained above, it is better to discard it and focus the analysis only on *Nginx* and *HAProxy*.

4.4 Examination of candidates

Candidates' documentations provide a good starting point to screen out the components lacking of features, but often they can describe outdated or even deprecated products, whose features might be documented but not yet compatible or performant enough for the needs of the business. The objective of this section is to stress the technology of each component, by means of increasing the desired total requests per second (for the first Performance Test) and concurrent connections (described in the second Performance Test), in order to see which are the effective limits of the softwares under analysis. Each candidate will be verified on a local environment to actively see by first hand if it actually provides the features described by its documentation and reported at 6.1 and, more important, which is the most performing one by means of maximum amount of requests per second and 99 percentile time that can be handled by a single instance of each load-balancer. The outcome of this step will define the winner of the experiment, hence the component that will be configured and tested on staging and finally deployed to production environment.
This performance test is meant to analyse the internals of each product and their implementation under high contention of resources, to see how different architectures, programming language or multi-threading model can influence the outcome and the behaviour of the candidates.

4.4.1 The set-up

This section describes the prerequisites needed in order to set the experiment up:

1. A **containerized environment** exploiting **Docker**, able to isolate component functionality inside containers and deploy them in a single-shot.

2. A **performance-test tool wrk2**, exploited in order to stress the candidates internals with the aim of finding the highest-performant one under equal conditions and same set-up.

---

1. The following section will provide a mock of the real scenario in which the selected load-balancer will operate, therefore the service will be linked to predefined farms containing servers written exploiting Express, a highly used server framework of Nodejs, whose only objective is to provide an API able to respond to HTTP POST requests in order to mimic from a high-level perspective bidders backend behavior. The following code snippet describe how a "toy" endpoint is defined at this preliminar stage:

```
const express = require('express')
const app = express()
const port = 5001

app.post('/farm_X', (req, res) => {
  res.send('Hello World from Bidder_X!')
})

app.listen(port, () => {
  console.log(`Bidder_X listening at http://localhost:${port}`)
})
```

This "dummy" server called Bidder_X only functionality will be to expose an API to react to the requests forwarded by the reverse-proxy under analysis.
Moreover, a sandbox environment is exploited: Docker [11] orchestrator is used in order to achieve a one-shot deployment of multiple services, through docker-compose [12] tool; it allows with a single command in the operating system prompt to boot and execute both our load-balancer under investigation and multiple instances of Node.js/Express, the set-up required by the experiment. The backend's mock servers are loaded in containers that exploits Node.js Docker official image.
A dockerfile is defined to properly load the container and to boot the servers.

```
FROM node:latest
COPY . /src/upstream
WORKDIR /src/upstream
RUN npm install express --save
CMD [ "node", "http-server.js" ]
```

Specifically, starting from the latest image available of Node.js, the required dependencies are installed inside each express container exploiting RUN statement with the given settings (a basic package.json in order to use Node.js is contained inside /src/upstream); then the server code is executed and the API is exposed. The following picture provides an overall view about the Docker containers interfacing with the load-balancing component:

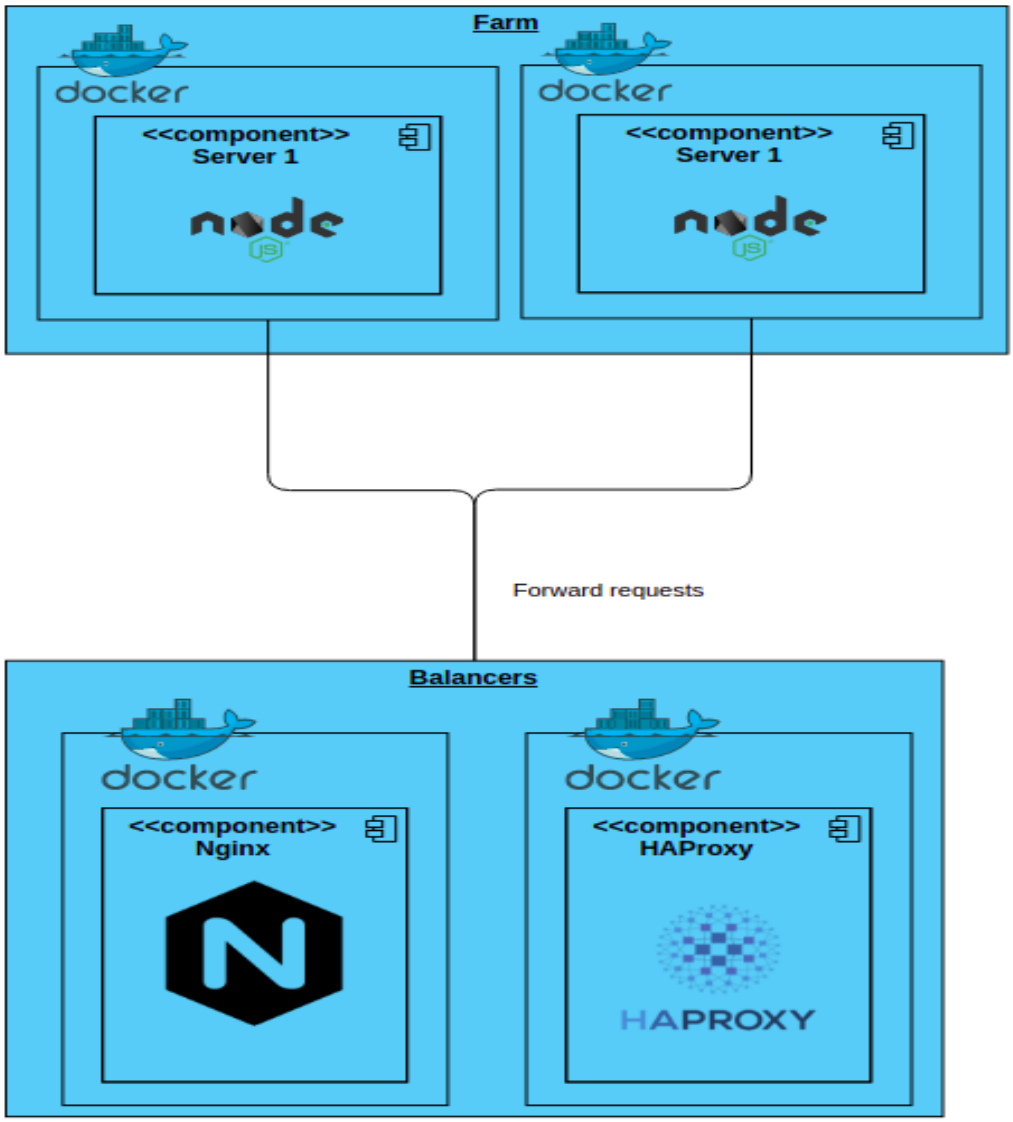


Figure 4.1: Testing environment

Each component is encapsulated into a Docker container, while the links represent the interconnections provided by Docker for the components to exchange HTTP POST requests. In order to obtain a fair comparison of the performances of each candidate, the experiment is performed over a farm made up of two servers, to test the load-balancing ability of the candidate; moreover the load-balancing algorithm selected is *round-robin*, the most basic one among others, with the aim to provide an equity scenario for the two component left. Hence, the same configuration of each load-balancer is a way to test exclusively the performances given by the various internals of the products, with the objective of extracting the most performant one.

2. For each load-balancer, one of the two farms is going to be stressed using a performance-test tool called **wrk2** [21], which is able to model a bulk stream of HTTP POST requests, providing configurable options such as number of connections, number of requests per second across all the connections and duration of the experiment. The tool will be set to forward requests to each load-balancer under evaluation, sampling the round-trip-time and generating a probability distribution for each forwarded packet, reporting also the average rate of requests per second sampled from the test. In this way it is going to be clear the critical point that will rise when the load balancer will start queuing the requests and suffering of internal delays.

4.4.2 Performance Test: increasing requests per second

The performance test will be driven on a machine with the following requirements:

- 8 cores 11th Generation Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
- 8 GB LPDDR4 RAM memory

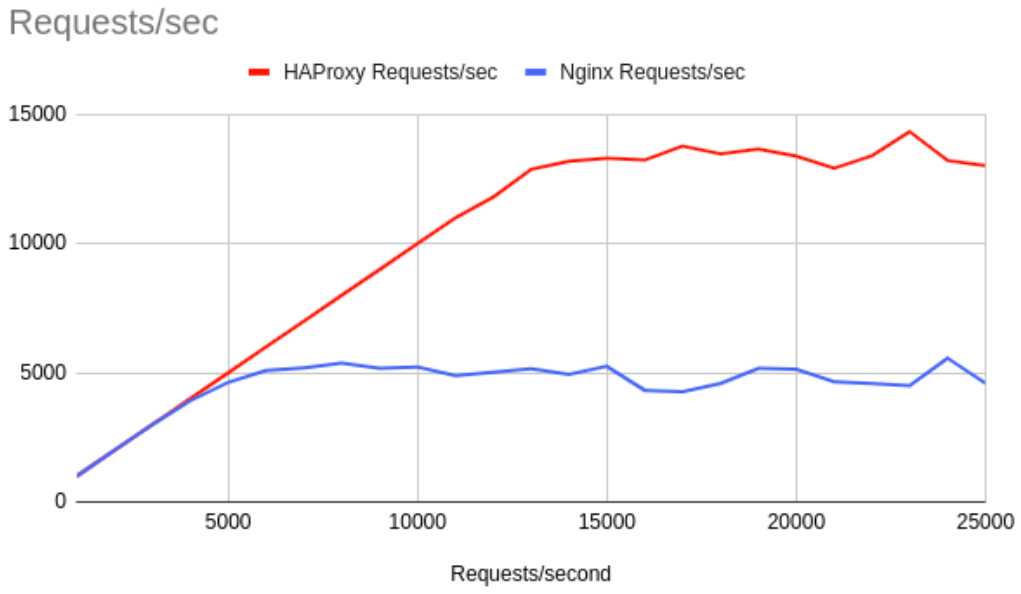
Each sample is processed during the 10 minutes duration with the previous settings. The performance-test tool wrk2 instead will have the following settings:

- 8 threads (one thread per core)
- 8 connections
- 10 minutes of overall duration (enough to guarantee a realistic outcome and to overcome wrk2 calibration time as defined by [21])
- a variable number of requests per second spanning from 1000 to 25000 with an increase of 1000.

This step is considered crucial and it will define the winner of the market analysis for this specific type of software. The evaluation will relies on two fundamental concepts:

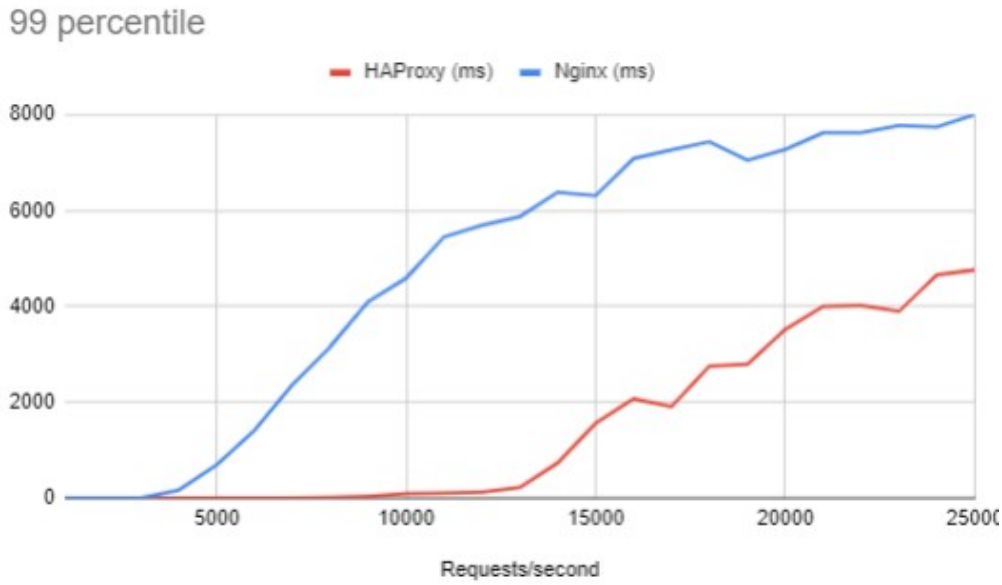
- Requests per second: the maximum amount of requests per second that the component can manage without incurring in errors, timeouts or excessive tail delays. The ideal load-balancer is the one that can process the incoming HTTP requests keeping a stable behaviour, showing a linear trend versus the bulk of requests sent by the performance-test tool (i.e. wrk2 sending a bulk of X requests/second, the load-balancer is able to handle and process the same amount X).
- 99 percentile time: this is a quality indicator mainly used in statistical analysis. In this particular scenario reflects the maximum value taken by the the 99% of the requests round-trip-time (RTT), therefore an high value of this parameter will mean that tail delays have occurred. The ideal load-balancer keeps the variance of distribution of the times low, guaranteeing a fair distribution of times. High values of this parameter mean starvation or more in general that a problem occurred; generally this issue is mainly due to high contention of resources but may be also caused by host failures and packet loss.

As already underlined, the balancer are configured with the same features to guarantee equality condition; moreover, *Nginx* and *HAProxy* are set to run as single-thread instances: for sure, if a product offers the possibility to run its executable exploiting multiple processes or threads, then the final configuration (in a production environment) must take into account this feature since capable to provide better distribution of processing resources. On the other hand, in the interests of discovering the processing limit of such components, a single-thread solution will be tested in order to ease the discovery of the technology limit of the components under evaluation. The results are the following:



The performance test is meant to extract the maximum average requests per second giving a quantitative idea of the number of requests per second a component can forward maintaining

a stable behaviour; from the first plot it is possible to notice how the number of requests per second of each component is linear up to an inflection point, each candidate then saturates around a given value of requests per second. This trend is common and recognizable for both load-balancers. *Nginx* shows the inflection point at the lowest value (which average in the saturation region is 4921 req/sec), while *HAProxy* ranks with better performances, respectively with 13231 req/sec.



A similar trend is observed from total round-trip-time expressed as 99 percentile. The times are extremely low initially for both candidates and then they start increasing steeply from the inflection point of each candidate recognized in the previous step. Once again, *Nginx* shows worse performances compared to the other two competitors showing huge tail latency delays that in some case almost reaches 8 seconds.

4.4.3 Performance test: increasing number of connections

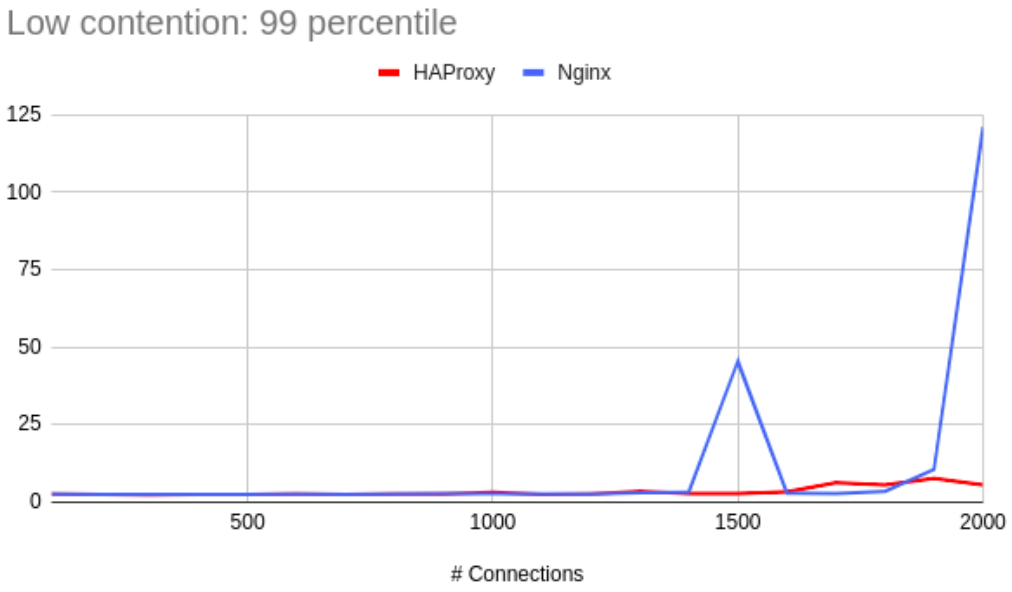
On the contrary, *Nginx* and *HAProxy* might be addressed by many various connections forwarding multiple requests to the back-end side; this scenario wants to show how an high number of connections can be a criticality for the balancers under evaluation, since it can expose requests to starvation and drastically change their performances due to higher level of parallelism. This experiment wants to show the effect of high concurrency and resources contention over the candidate 99 percentile and maximum Round Trip Time that might be suffering to spikes due to the relevant level of parallelism; to achieve this, it is necessary to enable the candidates to exploit their multi-programming ability. In particular, both *Nginx* and *HAProxy* works exploiting an event-loop strategy meant to wait for incoming HTTP requests and to assign each request to a ”worker”. The main difference is the definition of the worker: *Nginx* follows a multi-processing strategy, assigning one or many requests to a

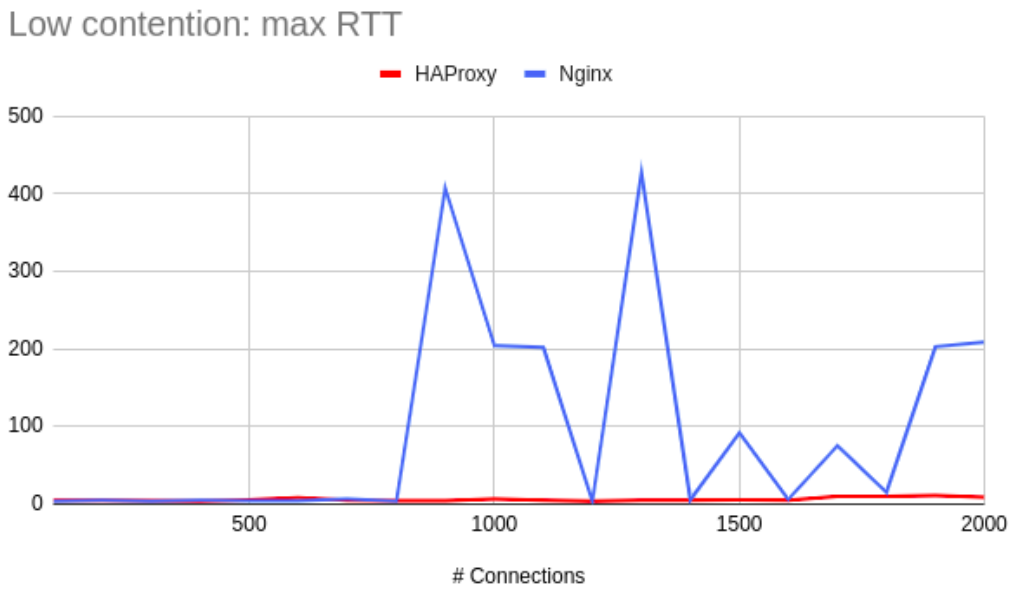
process forked from the main one; on the other hand, *HAProxy* supports also the multi-processing method but it is set as deprecated in the latest versions, the new state of art for the product is to distribute requests among threads, a light-way version of process instances. The following results are obtained setting the number of *Nginx*'s processes and *HAProxy*'s threads to the number of physical CPU available in the machine, hence eight as defined in the previous experiment.
Moreover, this section splits the experiment in two sub-cases, providing a scenario for high contention (heavy load of requests per second over the connections) and one for low contention (soft load over the connections):

wrk2 for low contention scenario will have the following settings:

- 8 threads (one thread per core)
- 1000 requests per second
- 10 minutes of overall duration
- a variable number of connections spanning from 100 to 2000 with an increase of 100.

The achieved results are the following:

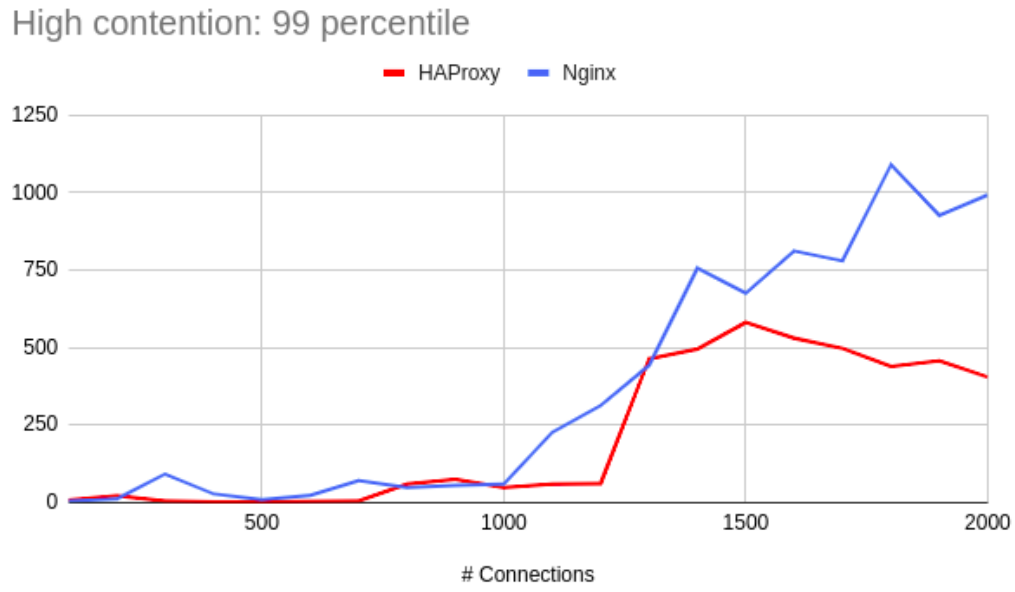


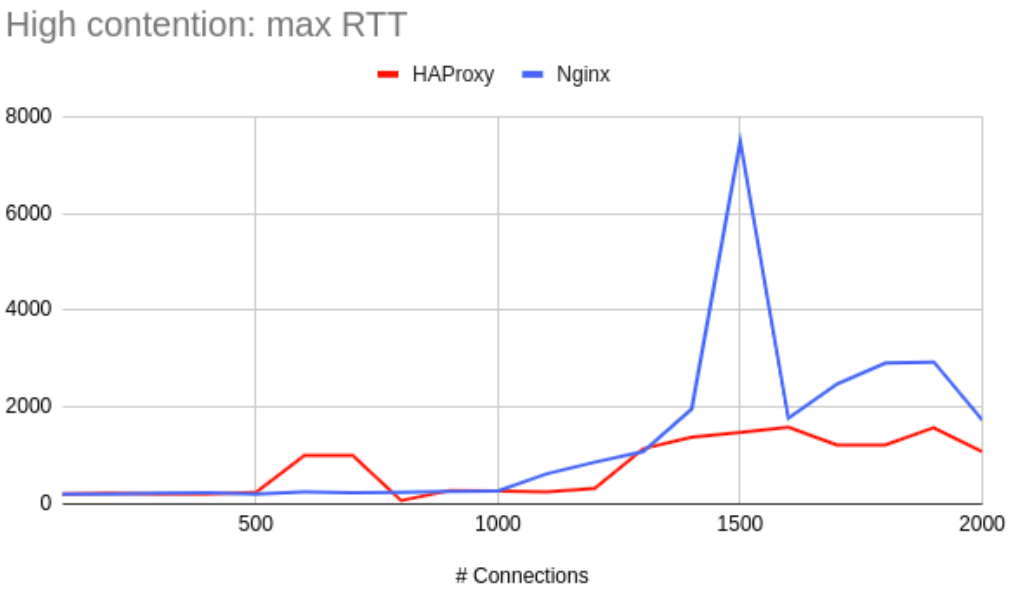


wrk2 for high contention scenario will have the following settings:

- 8 threads (one thread per core)
- 15000 requests per second
- 10 minutes of overall duration
- a variable number of connections spanning from 100 to 2000 with an increase of 100.

The achieved results are the following:





From the previous plots, it is possible to observe how *HAProxy* behaves better exploiting multi-programming in a high-demand environment cutting off the response times and providing an efficient balancing of the resources and of times probability distribution, with a performance degradation that is noticeable only for high contention scenario. *Nginx* on the other hand seems being much more affected by spikes: even with fewer number of requests it is already possible to notice spikes and response time degradation, with results that become even worse in the high contention case with observed maximum delays of almost 8 seconds for the slowest request.

4.5 Conclusions of the software selection process

It is then evident how a product is definitely better than the other: *HAProxy* is the undisputed winner of the software selection, both considering the results provided by means of the software evaluation matrix at point 4.3 and the outcome of the performance test. *HAProxy* is the product that by itself is able to encapsulate all the feature needed to integrate the balancer component inside the platform held by the Company with the add-on to provide the best performances among the compatible products.
Moreover, the results underline how a load balancer can actually add an overhead to the packets round trip time by setting a delay which changes depending on the load balancer service and its implementations; a product that is characterized by low variances in responses will improve the overall backends' quality of service, guaranteeing a better management of the servers' load and reducing the possibility to have slowness and delayed packets.

5 CONFIGURING HAPROXY TO PLATFORM ENVIRONMENT

As highlighted previously, *HAProxy* is the most promising product to be implemented as the balancer component since it sums up the best features over requirements, considering the outcome shown in the section relative to the performance test over a single reverse proxy instance too.
This chapter is meant to show how to configure the selected component and integrate it in the DSP software modules logic, deploying it in a *production* environment where to test its functionality and to observe whether the back-end performances are affected by the implementation of the new balancer module based on *HAProxy*; an overview of the balancer configuration will be provided, with the objective to provide a realistic mapping of the features that a *balancer* component must deploy in a *RTB* use-case. This purpose requires to provide the right configuration for the component, but it is not enough to deploy a balancer able to respond to the various change in the infrastructure, where bidders might join and leave back-end farms; therefore, a way to properly initialize and dynamically configure *HAProxy* will be defined; in addition, a way to sample the stats of the component must be defined in order to integrate the current metrics aggregators and plotters already available inside the platform (i.e. *Statsd, Grafana, CloudWatch*). The initialization, the dynamic update of the configuration and the metrics retrieval will be provided by means of Bash script that interacts with the balancer component by means of *HAProxy*'s interfaces exposure. The following component diagram provides an overall description of the various components and actors involved in the process:

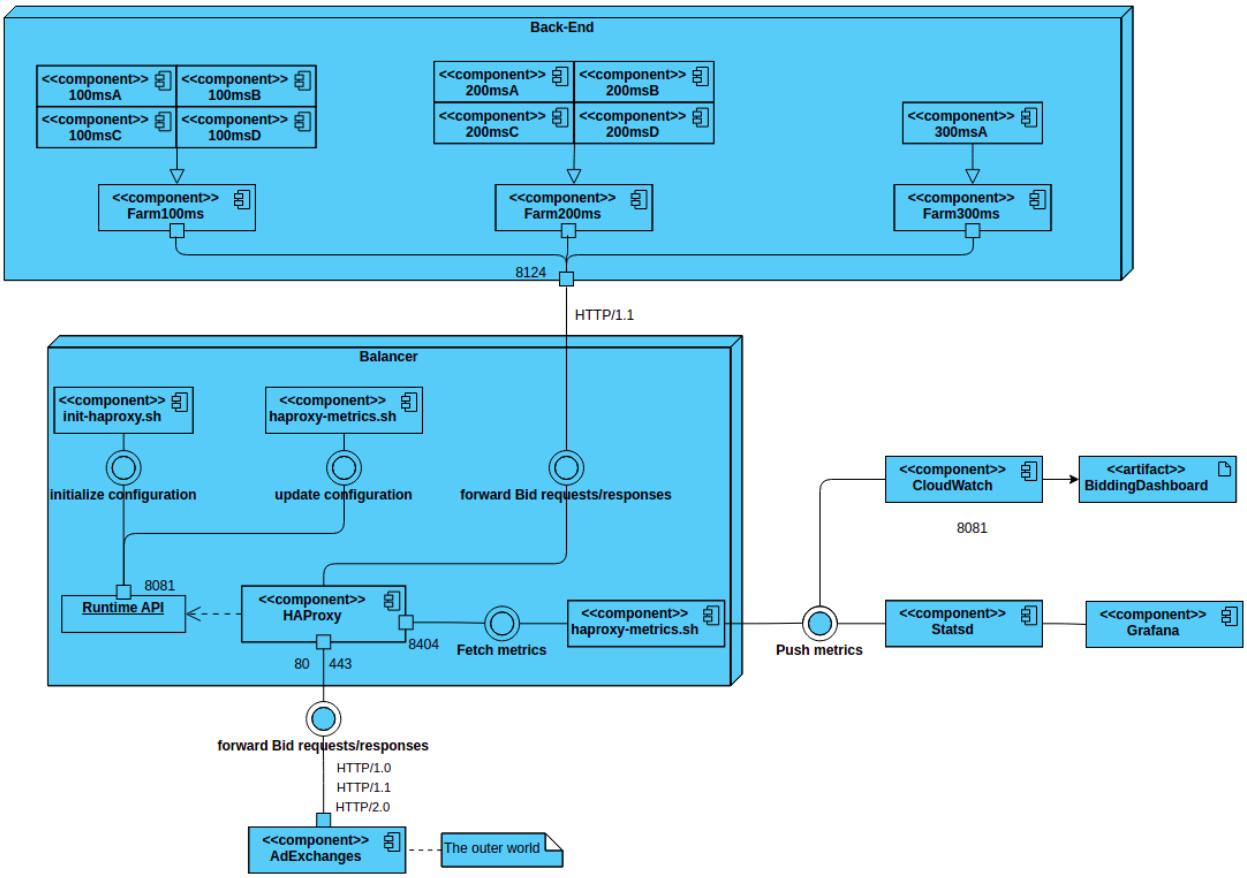


Figure 5.1: Balancer module component diagram

5.1 HAProxy configuration overview

HAProxy's configuration file shows two relevant parts: the front-end and the back-end sections which interfaces respectively with *AdExchanges* and Bidders. The following sections reports the *HAProxy* configuration, here it is reported a use-case diagram in order to summarise and overview the use-case that the HAProxy configuration will cover:

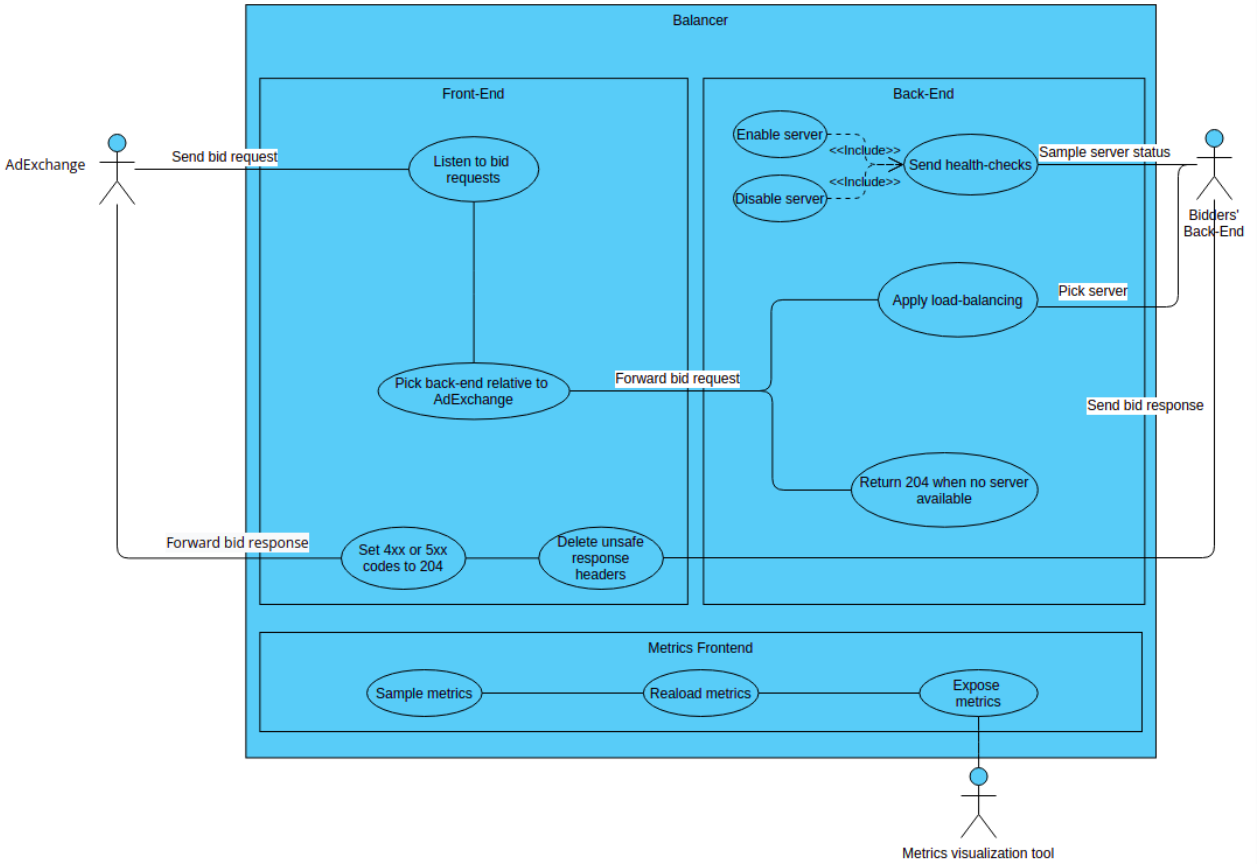


Figure 5.2: Use-case diagram for HAProxy

Front-End main role is to listen and re-dispatch bid requests towards back-ends. Two relevant front-ends are defined, respectively for listening to incoming requests and for exposing the internal metrics of the component. Each back-end is dedicated to at least one *AdExchange*, multiple back-ends are defined.

5.1.1 The Front-end side

The front-end side exposed by *HAProxy* defines the entrypoint for bids in the platform, listening to *HTTP requests* coming from *AdExchanges*

The following code snippet shows a sample of a configuration file for the frontend:

```
frontend http-in
        bind 127.0.0.1:80
        bind 127.0.0.1:443 ssl crt /etc/haproxy/ssl/server-certificate.pem
        mode    http
        option forwardfor                                          (1.)

        acl is_exchange123 path -i -m str /ad/rtb/123/bid
        acl is_exchange456 path -i -m str /ad/rtb/456/bid
        acl is_exchange456 path -i -m str /ad/rtb/789/bid          (2.)

        acl valid_method method POST
        http-request return status 444 if ! valid_method
        acl status_5xx status 500 502 503 504
        http-response set-status 204 if status_5xx                 (3.)

        use_backend exchange_123_100msA  if is_exchange123
        use_backend exchange_456_100msB  if is_exchange456
        use_backend exchange_789_100msC  if is_exchange789
        default_backend errors                                     (4.)

        http-request del-header Api-Version
        http-response del-header Server
        http-response del-header X-Powered-By                      (5.)
```

1. The *bind* statement defines the IP codes our load-balancer is listening to; in particular the balancer will receive traffic on any IP address 127.0.0.1 (**localhost**) assigned to the server at port 80 and 443. The *HAProxy*'s frontend supports HTTP connections for HTTP versions 1.1 and 2 at port 80 from clients; in addition, HTTPS encryption protocol can be used through standard port 443 given that a *PEM* certificate is provided to the *HAProxy* instance configuration. PEM file must contain the private key and certificate references.
   Furthermore, the *option forwardfor* is inserted; since *HAProxy* works as a reverse proxy, sitting in front of a client and a server, the server will recognize it as a client. The option forward for allows *HAProxy* to exploit the *X-Forwarded-For* header, bypassing in this way the issue just cited and providing to back-ends the right sending address.

2. From a front-end, there is the possibility to define rules called **Access Control Lists** (ACLs) that allows *HAProxy* to take forwarding decisions based on the request headers, status or on any other environmental data. *ACLs* permit the frontend to retrieve the right back-end in charge of managing the cluster of servers in that farm. The *use_backend* statement is used to redirect the packets coming from an *AdExchange*

(defined by an ID, in this example 123 and 456) to the proper back-end according to the *ACL* rule. The path is hence configured to respect the standard of HTTP requests path, which follows the schema "/ad/rtb/ID/bid" to map the requests coming from a given AdExchange to the right farm of servers.

3. Exploiting *ACLs*, it is also possible to define rules for incoming HTTP packets. Through *http-response* and *set-status* statements, it is possible to rewrite the HTTP header response code to *204* instead than an Error response. Each *5xx Server Error* is mapped into a *Sucess 2xx* type, avoiding misleading behaviours from client side.
   In addition, it is also possible to define conditions to block undesired requests and return a default response for them; in our case, *OpenRTB* works exclusively by sending information among agents through *POST* requests, any other method should be avoided in order to guarantee the correct functioning of the component. In case of a non-POST request, it will be returned then a *444 Connection Closed Without Response* directly, avoiding the forward to any possible back-end as expressed by *OpenRTB* standard [3]: invalid calls, such as a bid request containing a malformed or corrupt payload, should return a *HTTP 4xx* type with no content.

4. As already expressed, *use_backend* is exploited to redirect requests according to routing schema; in addition, it is possible to define a default behaviour in case of non-matching routes with the previously defined ACLs. This is done to filter out requests not coming from the mapped *AdExchanges* to guarantee a better management of security, avoiding possible intruders to reach the server side. The naming of the various back-ends is also an important aspect to be considered. Each back-end is mapped to one or multiple *AdExchanges*, which interacts with a specified bidders' farm featured by the maximum response time accepted by the AdExchange itself. The available number of farms changes in each region according to the demand of bids in that geographical area and the number of *AdExchanges* to serve in it, but the most generic scenario implies three general time limit for responses, which are 100ms, 200ms and 300ms, distributed over 4 different sub-farms for 100 and 200ms ones (addressed by the letters A, B, C and D) and over a single one for 300ms. Since the Platform operates over a dynamic environment, bidders in each sub-farm change in numbers and IPs, therefore in order to update the configuration, it is required to provide to the right balancer back-end the respective server in case of changes over the bidders side. The naming of the back-end will be used to retrieve the respective back-end to update at bidders' remotion/addition occurrency in the associated farm. Example: "bidder remotion in 100msA, update servers list of exchange_123_100msA".

5. The HTTP headers manipulation. HTTP requests and responses are filtered by the proxy at headers granularity level and they can be removed or modified by the proxy itself: many security experts warns that it is better to shade this kind of technical details of the servers, avoiding to expose critical and fragile information about the servers implementation that might furnish to intruders and attackers an assist to understand the fragility of the back-end side. For instance, the two headers "Server" and "X-

Powered-By" provides data about the type of server and the technology stack used, deleting it we ensure not to give away this details for free.

5.1.2 The Back-end side

Back-ends section instead define the endpoints that in our scenario are the bidders. Each exchange is mapped to a set of bidders that define a farm. Each farm can be composed by a multiple number of bidders that varies from back-end to back-end and, as expressed already, can change during time. The following is my purpose of general back-end's configuration for the *RTB* process, it is meant to give a general overview about the logic behind every *AdExchange*'s mapping inside the balancer component where the actual load-balancing algorithm is applied:

```
backend exchange_123_100msA
    balance random(2)                                    (1.)

    mode http
    option httpchk GET /heartbeat                        (2.)

    acl is_down nbsrv eq 0
    http-request return status 204 if is_down            (3.)

    timeout connect 70ms                                 (4.)

    server server1 127.0.0.1:8124
    server server2 127.0.0.1:8124
    server server3 127.0.0.1:8124
    (...)
    server server200 127.0.0.1:8124                      (5.)
```

1. The load-balancing strategies to adopt. *Power of Two* algorithm can be selected in the shown way.

2. The HTTP option section. The "check" statement aside *option httpchk* trigger health checks, useful for sampling back-end health status; it allows also to define the method and the path to use to sample the state of the servers. Company's implementation of bidders expose an endpoint /heartbeat to provide an easy way to sample server status. Therefore, the back-end healthiness is checked periodically sending HTTP request to each server. Servers that provide a *200 Response Code* are labeled as UP by the health-checks, while those who respond with a different code are marked as DOWN and disabled, in order to stop forwarding requests to a non-working server.

3. Each back-end might cause operational problems if every server in it is set to be down, triggering *HAProxy* to directly respond with *5xx* response. This issue is caused by the

health checks informing *HAProxy* that one of its endpoints is set to be DOWN; in order to avoid this behaviour, an ACL is formed, using *nbsrv* keyword, exploited to sample the number of servers in active state. If no server is found to be up and running, then a 204 response code is returned to avoid *AdExchanges* to cut-off the estabilishment of connections with the Company's bidding platform.

4. The total amount of time before a connection under establishment is shut down. It is fundamental to provide this kind of timeouts to avoid *HAProxy* to wait for unlimited time for TCP connection hand-shake success, issue that would cause the instance to rapidly saturate the CPU and causing malfunctions. It is required to be set to a value lower than the expected response time for the given farm, here it is set to 70ms in order to terminate slow connections establishment.

5. This section is dedicated to the addresses of the bidders in the farm. Every bidder is responding to requests received at port 8124 and is featured by a *private IPv4* address generated by the cloud's Infrastructure as a Service.
   At start-up, the component is featured by 200 **place-holder** servers, featured by loop-back address (i.e localhost). A place-holder is set in order to define an empty entry in the given *HAProxy* back-end to define a possible spot where to write a real bidder IP address. This solution is used to exploit fully the **Runtime API**, embedded directly in *HAProxy* service, which allows the dynamic configuration of the service without reloads (as explained in the next section) by means of commands to enable, disable and change each server entry; server entries can not be explicitly removed or added through the API but only be updated, therefore the place-holders numbers mock the worst-case-scenario for the highest amount of bidders in a region, in order to always have empty-slots in the server list to be filled with bidders' IPs. This section will be reviewed later on during the discussion to show the logic and the way to initialize and update the server list of each back-end.

Moreover, two other back-ends are defined in order to sharp the functionality of the component itself: it is mandatory to have a redirection point for non-matching paths (i.e. an endpoint for the errors) but also to expose a way to check the status of the balancer component without propagating the status check to the bidders. For these reasons **errors** and **ready** endpoints are exhibited:

```
backend errors
    mode http
    http-request return status 444

backend ready
    mode http
    http-request return status 200
```

Requests featured by non-matching paths are directly returned to the client with a *444* error code with No content following OpenRTB specifications [3]; on the other hand, requests carrying */ready* will be sent back with a 200 status, meaning that the component is up and ready to listen requests at the defined ports.

5.1.3 Tuning HAProxy

After having configured *HAProxy*'s entrypoints, endpoints and settings, it is required to provide the right tuning to the service, in such a way that it is possible to optimize the behaviour of the back-end side and of the proxy itself. Timeouts and options have to be exploited in order to design a component that perfectly fits the needs of a *DSP* but that also protect the Platform as first barrier from malicious attacks.
Analysing deeply the bidding process requirements, it is evident how the use-case under consideration is different from an ordinary scenario that involves balancers sitting in front of servers, usually this kind of services interacts directly with customers/users HTTP requests trying to address a particular server to process some kind of information: in a context of a general web application open publicly, thousands if not millions of user wants to connect to the balancer to access the back-end processing power.
From a Demand Side Platform point of view instead, the customers (i.e. the agents forwarding requests) are static and pre-defined and are addressed as the *AdExchanges*. Commonly, *DSPs* interacts with a limited number of *AdExchanges*, usually on the order of magnitude of hundreds, therefore adjustments over the settings can be done to address this scenario of **fixed** clients. In such a scenario, where customers interacts with the Platform in a settled number, the utilization of **connection keep-aliveness** can play a fundamental role, reducing the connections establishment overhead over the balancers' CPU and fully exploiting keep-alive performances increase. In addition, *AdExchanges*' traffic vary a lot during the day and connections can remain idle for some periods of time. In a non-RTB scenario, long idle connections can badly affect the behaviors of servers, exposing them to malicious attacks such as *DDoS*, where attackers may try to establish a huge number of connections from multiple peers in a short amount of time; idle periods are expected in a *RTB* scenario as well but timeouts must be tuned in order to avoid possible attackers to block the Platform to provide service: *AdExchanges* interacts with the platform using the most disparate form of protocols and throughput and may remain silent for periods; allowing idle periods would definitely improve the CPU overhead of the Platform instances, avoiding the establishment of a new connection in case of a bid request after long time of inactivity but on the other hand, excessive timeouts value can expose the Platform to Down of Service kind of attack. As reported by *OWASP* (Open Web Application Security Project), an online community that provides free articles, documentation and tools for Web Application security, suggests to "set session timeout to the minimal value possible depending on the context of the application" and to "avoid infinite session timeout" [23].
In particular, my *HAProxy* configuration implementation provides a "default" section where the tuning options are reported:

```
defaults
  option http-keep-alive                                    (1.)
  timeout http-keep-alive 120s
  timeout client 120s
  timeout server 120s                                       (2.)
  timeout http-request 4s                                   (3.)

  default-server maxconn 64 slowstart 1m
       check port 8124 alpn http/1.1                        (4.)
```

1. The **option http-keep-alive** is defined in order to allow a single TCP connection to remain open for multiple HTTP requests/responses, improving the overall connections management from *HAProxy* threads point of view allowing connections persistency. As suggested by *OpenRTB* standard as best practice, "one of the simplest and most effective ways of improving connection performance is to enable HTTP Persistent Connections, also known as Keep-Alive. This has a profound impact on overall performance by reducing connection management overhead as well as CPU utilization on both sides of the interface" [3].

2. the amount of time for the connection from *AdExchanges* to *HAProxy* (**timeout client**) and from *HAProxy* to servers (**timeout server**) to stay idle before it gets shut down. **timeout keep-alive** defines the amount of time between a HTTP requests and the successive on the same TCP connection before the connection is shut down. These timeouts together generally reference **connections idle time**.
   As reported by *OWASP*, "common idle timeouts ranges are 2-5 minutes for high-value applications and 15-30 minutes for low risk applications. Absolute timeouts depend on how long a user usually uses the application. If the application is intended to be used by an office worker for a full day, an appropriate absolute timeout range could be between 4 and 8 hours" [24]. DSPs define a use-case where the connections are highly exploited and excessive idle periods must be avoided, being a low-latency-high-throughput system where massive amount of requests hit balancers continuously would suggest to provide low idle timeout values following the OWASP guide-lines. Moreover, *timeout client* helps the detection and the avoidance of *DDoS* attacks: *DDoS* aims overloading servers opening huge amounts of idle connections; setting *timeout client* to a value will force *HAProxy* to close the connection after the timeout value expiration. Letting this parameter undefined can expose the whole Platform to this kind of cyber risk.

3. **timeout http-request** defines the maximum time for receiving the full list of HTTP headers from client side. It is particularly useful to avoid **Slowloris** attacks, a particular and improved form of Down of Service issue, more difficult to detect, that aims to send the HTTP headers very slowly, avoiding the connection to be cut off because of inactivity but overloading anyways the server resources; including this peculiar timeout

would allow *HAProxy* to detect HTTP request slowness, cutting down the possible Slowloris connection attempts.

4. As defined by *HAProxy* documentation [19], **default-server** statement defines the default option for each server in the section scope. It is defined in the default section in order to provide the declared options to every server in the configuration.
   Specifically, the following options are defined:

   - **maxconn 64** defines the maximum concurrent connections over a server. It is set to 64 in order to map the number of threads of the back-end servers. In such a way, it is guaranteed that the number of connections to the server side won't affect badly its resources utilization, providing a one to one mapping for number of threads and maximum connections to each server.
   - **slowstart 1m** defines the behaviour for servers moving from DOWN to UP state; since bidders are affected by a warm-up time at launch and a consequent higher CPU utilization, it is better to provide a period of time where the connections are gradually open, slowly reaching the *maxconn* value, reducing the stress for a bidder and the relative CPU overhead due to connection establishment.
   - **check port 8124** forces *HAProxy* to send Health-Checks to the bidders at the pre-defined port 8124.
   - As expressed at the beginning of this chapter, the Platform receive HTTP traffic featured by various HTTP protocols. Internally, the communication is established with protocol HTTP/1.1 only, therefore **alpn http/1.1** ensures the right protocol choice for balancer-to-bidders communication.

5.2 Configuring dynamically without experiencing down-time: the Runtime API

HAProxy provides a built-in software module available in its distributions, able to expose an endpoint that provides various features such as updating *ACLs*, front-end and back-end section, showing internal stats and changing load-balancing weights without reloading the service and experiencing down-time. This feature is called **Runtime API** and it provides a general way to update *HAProxy*'s configuration during run time.
As expressed by [20], the defining trait of the *Runtime API* is that all configuration changes are applied in memory, but do not alter the *HAProxy* configuration file on disk; this solution is designed in order to exploit better standards for performance and security: *HAProxy* never reads its configuration from the filesystem after completing its initial startup. During startup, it reads the configuration file, along with any other supporting files like TLS/SSL certificates, and then keeps a representation of those files in memory. The Runtime API modifies those in-memory representations only without propagating changes to the disk, ensuring a full no-reload solution. So, after calling a *Runtime API* function, *HAProxy* is able to apply

changes in-memory without propagating the change to the disk; therefore, a change in the configuration will be kept inside the memory, meaning that changes in the configuration will not be persistent. A reload of the service will restore the initial configurations. This aspect might be a limit for the overall functionality of the component itself, but for the use-case of this master thesis it just fits perfect: the main feature that the Runtime API needs to cover is the dynamic update of bidders inside the back-ends' servers lists, whose IP addresses change constantly; therefore, persistency is not necessary for the use-case under analysis but instead, the API ensures *HAProxy* not to reload the configuration for the changes to take effect. Avoiding a reload helps speed up operations and uses fewer computing resources but also, it guarantees the component to be listening for incoming requests and connections all the time, that, in case of a reload, may be lost. Rejecting incoming connections from *AdExchanges* leads to a scenario where AdExchanges connections are refused, which might bring *AdExchanges* to cut off connections with the platform. Therefore, this behaviour must be avoided from the design of the component itself.
Runtime API must be defined from *HAProxy* configuration file:

```
global
  stats socket ipv4@127.0.0.1:8081 level admin expose-fd listeners
```

The standard way to interact with it is the following:

```
echo "<command>" | socat stdio tcp4-connect:127.0.0.1:8081
```

It is a Unix shell command in the form of pipe which purpose is to forward a command compatible with the API to the API itself at socket level.
The two following sub-sections show how to implement the *Runtime API* calls for initializing the servers and to provide *HAProxy* bidders addresses information.

5.2.1 Initializing the balancer

As described before, each back-end is initialized with 200 **place-holder** addresses. But those servers are considered "active" as they are real bidders instances, therefore *HAProxy* must boot having these servers disabled in order to avoid to forward real bids to 127.0.0.1:8124 that will head to a server error (contrary to bidders, balancer component do not expose port 8124) and in general, to avoid *HAProxy* to forward useless health-check requests to the place-holder endpoints.
The place-holder entry sums up the localhost address and the disabled state in order to provide a way to identify idle (free) entries in the back-end server list. Therefore, in initialization stage, the *Runtime API* is exploited to scroll the possible back-ends and to disable all the servers inside them; the real bidders addresses will be added with the script described in the section dedicated to dynamic updates and at farms' bidders change. Here it is a UML activity diagram representation:

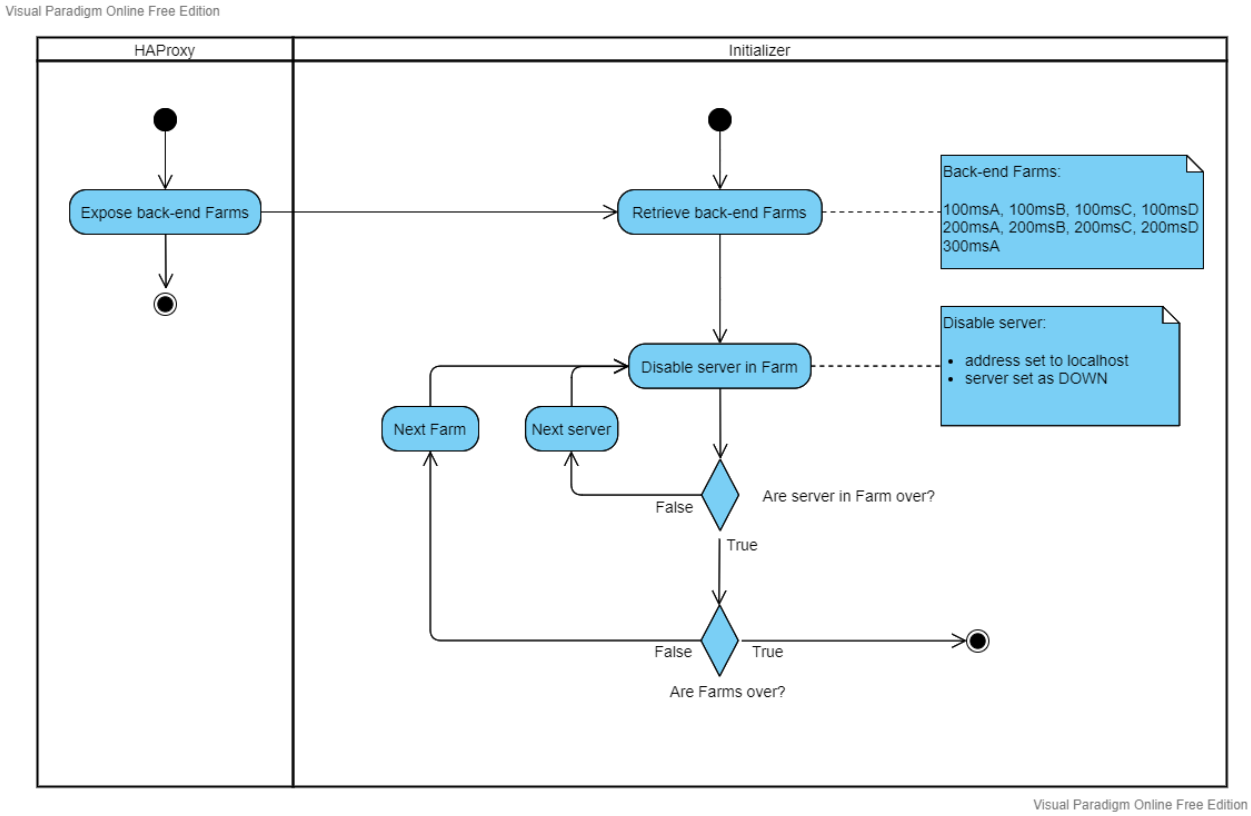


Figure 5.3: Init process UML Activity diagram

Following, the relative Bash script:

```
sudo service haproxy restart

BACKENDS=$(echo "show backend"
 | socat stdio tcp4-connect:127.0.0.1:8081)

for BACKEND in "${BACKENDS[@]}"; do
  i=1
  while [ "$i" -le 200 ]; do
    echo "disable server $BACKEND/server$i"
         | socat stdio tcp4-connect:127.0.0.1:8081
    ((i=i+1))
  done
done
```

At first, *HAProxy* service is restarted and the configuration file is read from disk. Then, the list of available backends is retrieved directly from *HAProxy* configuration exploiting the Runtime API.
Finally, the various back-ends are listed and each server is disabled through the API call.

5.2.2 Dynamically updating the balancer

The following UML Activity Diagram summarize the Dynamic Update process logic:

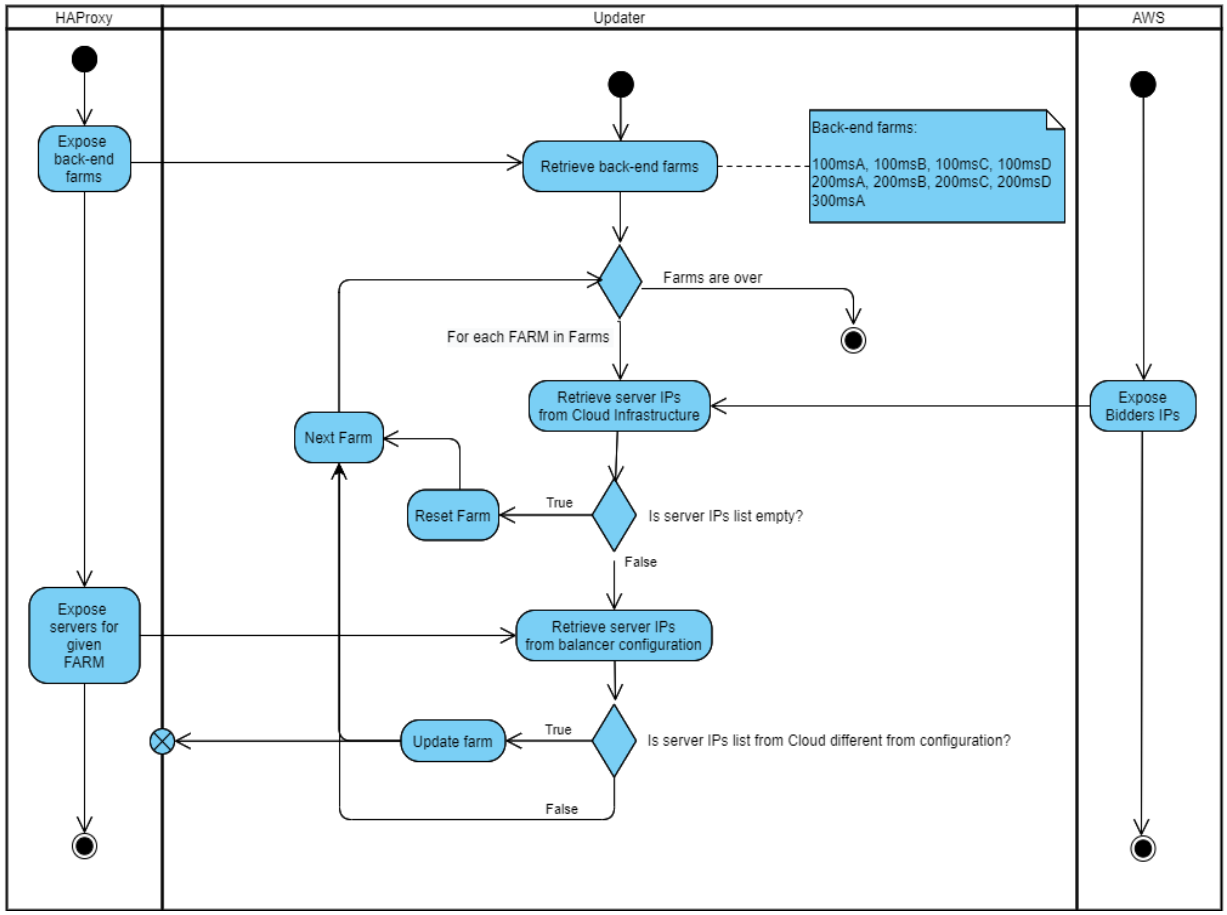


Figure 5.4: Dynamic update UML Activity Diagram

The logic behind the dynamic update of the configuration algorithm is the following: it is triggered at component startup right after initialization phase and periodically to react back-end changes. The main idea is to scroll every available farm of bidders inside the region (a subset of the following: 100msA, 100msB, 100msC, 100msD, 200msA, 200msB, 200msC, 200msD, 300msA) and sample from the Cloud infrastructure the available bidders (called in the example BIDDER_LIST) and the respective IP address in order to compare with the ones currently stored inside *HAProxy* instance (called in the example SERVER_LIST). The comparison is done through the *isEqualUpstreamList* function that compares the two lists of addresses. If a change is detected, one of two possible scenario might happen: back-ends identified as empty (i.e. no bidders available over the Cloud Infrastructure) are processed to reset the defined bidders through the *resetBackend* function, setting them back to the *place-holder* server in order to model a back-end actually free of bidders; on the other hand, non-empty back-ends' server list is scrolled in order to remove bidders that turned the state into DOWN and following, new bidders are added to the list. Following, the functions described in the previous Activity Diagram are written exploiting *Bash* command language and they are reported here below:

```
function isEqualUpstreamList() {
  local BIDDER_LIST=$1
  local SERVER_LIST=$2

  diff -q <(echo "$BIDDER_LIST") <(echo "$SERVER_LIST")
  # Returns 0 when no diff == no changes.
  # 1 otherwise == changes detected.
  return $?
}
```

The comparison function simply performs the check through "diff" commands to effectively check that the two lists of IPs provided is the same (return code equal 0) or differ (return code equal 1).

```
function resetBackend() {
  local FARM=$1

  BACKEND=$(echo "show servers state"
            | socat stdio tcp4-connect:127.0.0.1:8081 | grep $FARM
            | cut -f2 -d" " | uniq)

  SERVER_NAMES=$(echo "show servers state"
            | socat stdio tcp4-connect:127.0.0.1:8081 | grep $FARM
            | cut -f4 -d" " | uniq)
```

```
  SERVER_NAMES_ARRAY=(`echo ${SERVER_NAMES}`)

  IPS=$(echo "show servers state"
    | socat stdio tcp4-connect:127.0.0.1:8081 | grep $FARM
    | cut -f5 -d" " | uniq)

  IPS_ARRAY=($IPS)

  echo "Resetting Backend for $FARM"

  i=0
  for SERVER_NAME in "${SERVER_NAMES_ARRAY[@]}"; do
    # Check if placeholder is available
    if [ "${IPS_ARRAY[$i]}" != "127.0.0.1" ]; then
      echo "disable server $BACKEND/$SERVER_NAME;
            set server $BACKEND/$SERVER_NAME addr 127.0.0.1"
            | socat stdio tcp4-connect:127.0.0.1:8081
      break
    fi
    ((i=i+1))
  done
}
```

The function resetBackend highly exploits the Runtime API to retrieve *HAProxy* back-end information; in particular, the API allows the retrieval of the relative BACKEND in the form **exchange_backendId_farmLatency** (i.e. exchange_123_100msA), SERVER_NAMES list such as the *HAProxy* internal server naming (server1, server2, ... , server200) and *IPS*, containing real bidders IPs and place-holder entries (localhost address plus disabled state). The retrieval of this information is due to the *show servers state* command sent to the *Runtime API* that simply shows the full server list of each back-end with the following format separated by spaces: **id back_end server server_name ip**. Piping the API outcome to *grep* and *cut* shell commands is done in order to extract the wanted field related to the the given farm.
Once extracted the needed data, the server list is browsed and for every non-place-holders entries (in other terms, real bidders IPs) the server state is set to "disabled".

```
function updateBackend() {
  local BIDDER_LIST=$1
  local FARM=$2
  local SERVER_LIST=$3

  BACKEND=$(echo "show servers state"
```

```
          | socat stdio tcp4-connect:127.0.0.1:8081
          | grep $FARM | cut -f2 -d" " | uniq)
SERVER_NAMES=$(echo "show servers state"
          | socat stdio tcp4-connect:127.0.0.1:8081
          | grep $FARM | cut -f4 -d" " | uniq)
SERVER_NAMES_ARRAY=(`echo ${SERVER_NAMES}`);
IPS=$(echo "show servers state"
          | socat stdio tcp4-connect:127.0.0.1:8081
          | grep $FARM | cut -f5 -d" " )
IPS_ARRAY=($IPS)

# Two steps are defined:
# Step 1: delete servers not available anymore
while IFS= read -r SERVER; do
  echo "$BIDDER_LIST" | grep -q "$SERVER" >> /dev/null
  if [ "$?" -ne 0 ]; then
    # Server NOT found in BIDDER_LIST,
    # so it exists in SERVER_LIST and needs to be removed.
    i=0
    for SERVER_NAME in "${SERVER_NAMES_ARRAY[@]}"; do
      # Search for SERVER
      if [ "${IPS_ARRAY[$i]}" == "$SERVER" ]; then
        echo "disable server $BACKEND/$SERVER_NAME;
            set server $BACKEND/$SERVER_NAME addr 127.0.0.1"
            | socat stdio tcp4-connect:127.0.0.1:8081
        break
      fi
      ((i=i+1))
    done
  fi
done <<< "$SERVER_LIST"
```

The first section of the *updateBackend* function is thought in order to remove those addresses which are not available anymore over the Platform back-end but still present in *HAProxy* configurations. After the retrieval of information from the API, each server inside the back-end related to the farm is verified to be present inside BIDDER_LIST. This condition to fail would mean that the bidder was available before in the platform but it has been removed, therefore the server is set to the "place-holder" value and disabled.

```
# Update vars to retrieve new placeholders set free after the
# previous while
SERVER_LIST=$(echo "show servers state"
```

```
                | socat stdio tcp4-connect:127.0.0.1:8081
                | grep --regexp "exchange_*"
                | grep $FARM | awk '{print $2, $5, $6}'
                | grep -v -w "127.0.0.1" | cut -f2 -d" " | sort)

  IPS=$(echo "show servers state"
    | socat stdio tcp4-connect:127.0.0.1:8081
    | grep $FARM | cut -f5 -d" " )
  IPS_ARRAY=($IPS)

  # Step 2: add new servers
  while IFS= read -r SERVER; do
    echo "$SERVER_LIST" | grep -q "$SERVER" >> /dev/null
    if [ "$?" -ne 0 ]; then
      # Scrolling farm to retrieve first entry where to set the server
      i=0
      for SERVER_NAME in "${SERVER_NAMES_ARRAY[@]}"; do
        # Check if placeholder is available
        if [ "${IPS_ARRAY[$i]}" == "127.0.0.1" ]; then
          echo "enable server $BACKEND/$SERVER_NAME;
                set server $BACKEND/$SERVER_NAME addr $SERVER"
                | socat stdio tcp4-connect:127.0.0.1:8081
          break
        fi
        ((i=i+1))
      done
    fi
  done <<< "$BIDDER_LIST"
}
```

The API is called again in order to refresh *HAProxy* internals' information step in order to retrieve every new entry set to place-holder in the remotion step. Then, the reverse logic as before is applied: if an IP address is available in BIDDER_LIST and not in SERVER_LIST then it has to be added to the *HAProxy* backend server list, therefore the first entry containing a place-holder value has to be updated to contain the new bidder IP address.

5.3 Metrics retrieval

One of the main purposes of the Company's platform is to always have a fresh collections of metrics to monitor various statistics and to detect possible malfunctions in the platform components. This is achieved exploiting *CloudWatch*, a management and monitoring service from Amazon Web Services that gathers metrics and display them, and *Statsd*, a service that aggregates and summarizes application metrics, as observable in the previously defined system

component diagram and forwards them to another monitoring service, such as **Grafana**. *CloudWatch* is internally used to display stats related to a specific process (such as the bidding process) by means of dashboards, while *Statsd+Grafana* hosts all the metrics and KPIs of the platform. Therefore, *CloudWatch* is meant to summarise the metrics related to the primary indicators for the bidding process (visualized to sample the "healthiness" and quality of service of the Platform bidding dynamics itself), such as the number of concurrent connections and the total requests per second forwarded by the balancer, while on the other hand, Statsd metrics aggregate the whole system statistics, useful to sample errors occurred with the Balancer instances, such as saturation of the hard-drive or wrong behaviours in the set-up of the network connections with *AdExchanges*. They operate following several standards and protocols and they might trigger unwanted or dangerous behaviour for the balancer component: an *AdExchange* forwarding Bid Requests exploiting HTTP/1.0 might not specify the *Connection: Keep-Alive* header, forcing the balancer to open a connection for each incoming packet and exposing the component to the possibility of running out of sockets and, therefore, starting rejecting and timing out incoming connections, causing errors and failure that can be sampled taking a look to the TCP stats from the OS level.
This section's aim is to provide a way to sample metrics from *HAProxy* and forward them to the monitoring services, proposing a way to extract several heterogeneous statistical data from the balancer instance.

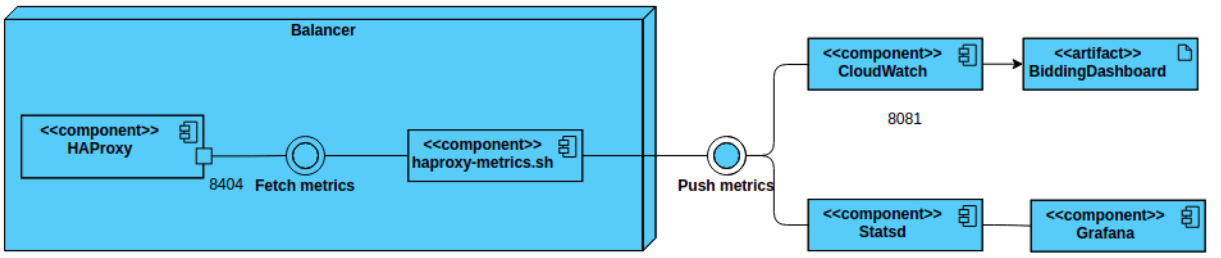


Figure 5.5: Metrics retrieval component diagram

Which are specifically the exact metrics that a Demand Side Platform wants to monitor from the balancer server instance?

- Active connections;
- Requests per second and total requests;
- General OSI Transport Layer info, such as TCP timeouts, delayed ACKs, received ACKs and TCP hand-shake failures;
- Balancer instance Disk occupation.

Those indicators are essential in order to understand the health of an HTTP component and the stability of connections with the *AdExchanges*; in addition, processed requests per

second and total requests are Key Performance Indicator for the Company, therefore a way to extract it from the component must be defined.
HAProxy allows a way to provide metrics, ensuring a way to expose an endpoint by means of the definition of a specific front-end.

```
frontend stats
  bind 127.0.0.1:8404
  mode http
  stats enable
  stats refresh 2m
  http-request use-service prometheus-exporter if { path /metrics }
```

Fresh stats are available at localhost port 8404 every minute; a work-around to push metrics is defined in order to provide readable stats using the *prometheus-exporter* statement: this is done in order to provide an easy way to retrieve a list of metrics separated by new line values at localhost:8404/metrics, useful later in the discussion to easily retrieve metrics from *HAProxy* exploiting a Bash script.
Therefore, here it is provided the way to get metrics from the balancer component and push them to the monitoring services:

```
#Get different metrics from haproxy
READY=$(curl -s "http://127.0.0.1:80/ready")
EXIT_CODE=$?
if [ $EXIT_CODE -ne 0 ]; then
  echod "error: Couldn't get information from
    http://127.0.0.1:80/ready"
  exit $EXIT_CODE
fi

# HAPRoxy's metrics endpoint
RESPONSE=$(curl -s "http://127.0.0.1:8404/metrics")
```

Then, the status of *HAProxy* is checked: if it is up and running, the endpoint for metrics retrieval is defined, otherwise the script exits triggering error code.

```
#Active connections metric
FRONTEND_CONNECTIONS=$(echo "$RESPONSE"
    | grep haproxy_frontend_connections_total
    | grep --regex http-in | cut -f2 -d" ")
STATS_CONNECTIONS=$(echo "$RESPONSE"
    | grep haproxy_frontend_connections_total
    | grep --regex stats | cut -f2 -d" ")
TOTAL_CONNECTIONS=$(echo "$((FRONTEND_CONNECTIONS
  + STATS_CONNECTIONS))")
```

```
if [ $ENVIRONMENT == "production" ]; then
    send_metric_cloudwatch
      "Balancer-ConnectionCount" $TOTAL_CONNECTIONS
fi
send_metric_statsd "connection.total" $TOTAL_CONNECTIONS
send_metric_statsd "connection.writing" $FRONTEND_CONNECTIONS
```

Connections' metrics are directly available over the endpoint exposed by *HAProxy*, they are extracted by means of command pipe with grep (to retrieve haproxy_frontend_connections_total for each front-end). Once again, the metrics are pushed to *CloudWatch* only for production environment.

```
#Request counter metric
# retrieve last window values so we calculate current period value
TMP_REQ_FILE_PATH=/tmp/reqs-record.tmp
M_KEY="Balancer-RequestCount"
TOTAL_REQUESTS=$(echo "$RESPONSE" |
  grep haproxy_frontend_http_requests_total
    | grep --regex http-in | cut -f2 -d" ")
if [ ! -f $TMP_REQ_FILE_PATH ]; then
  echod "info: Previous metric [$M_KEY] was empty and
    it is not possible to calculate current requests" >&2
  echo $TOTAL_REQUESTS > $TMP_REQ_FILE_PATH
  exit 0
else
  PREVIOUS_REQUESTS=`cat $TMP_REQ_FILE_PATH`
  READING_REQUESTS=`echo $(( $TOTAL_REQUESTS - $PREVIOUS_REQUESTS ))`
  echo $TOTAL_REQUESTS > $TMP_REQ_FILE_PATH
fi

if [ $ENVIRONMENT == "production" ]; then
    send_metric_cloudwatch $M_KEY $READING_REQUESTS
fi
#Reqs per second: Total requests on the period divided by the period.
send_metric_statsd "requests.perSecond"
  $(expr $READING_REQUESTS / 300) "gauge"
```

Requests per second are instead slightly more tricky to extract. *HAProxy* provides the value for the total number of received requests without providing effectively the requests per second value; therefore a file is exploited to store the number of requests received over the previous time window, sampled from the *HAProxy* metrics endpoint. The requests per second (QPS) are then calculated as the difference between the two time frames divided by the time window length, considering that the periodicity (dt) of retrieval of metrics from *HAProxy* is every 5

minutes (300 seconds):

$$QPS = \frac{tot\_req_{current} - tot\_req_{file}}{dt}$$

```
#Disk usage metric (%)
M_KEY=Balancer-Disk
M_VAL=$(df | grep '/dev/root' | awk '{print $5}')
if [ $ENVIRONMENT == "production" ]; then
    send_metric_cloudwatch $M_KEY $M_VAL "Percent"
fi
```

Since the disk metrics are not reported by *HAProxy* component itself, the disk metrics are directly extracted using *df*, a standard Unix command used to display the amount of available disk space, piped with other generic Unix commands in order to retrieve the Disk occupancy percentage for the root user fylesystem. If the instance is running in production environment, then the metrics are pushed to CloudWatch.

```
#Metrics about Tcp connections via nstat
METRICS="Tcptimeouts TcpExtDelayedACKs
    TcpExtTCPDSACKRecv TcpAttemptFails TcpEstabResets"
for METRIC in $METRICS
  do
    LINE=`nstat -z | grep -i $METRIC`
    M_KEY=`echo $LINE | awk '{print $1}'`
    M_VALUE=`echo $LINE | awk '{print $2}'`
    send_metric_statsd M_KEY $M_VALUE "Count"
  done

#Metrics about network bandwidth
METRICS=`ifstat -T -i ens5 1 1 | tail -1f`
M_KEY="network.bandwidth.in"
M_VALUE=`echo "$METRICS" | awk '{print $3}'`
send_metric_statsd $M_KEY $M_VALUE "gauge"

M_KEY="network.bandwidth.out"
M_VALUE=`echo "$METRICS" | awk '{print $4}'`
send_metric_statsd $M_KEY $M_VALUE "gauge"
```

This code snippet exploits *nstat* and *ifstat* (two Unix commands used to pull network metrics from the kernel and display them to the user); *nstat* is piped to grep to retrieve the wanted metric, while *ifstat* has been provided with the main network interface of the balancer instance and by two 1s, which force *ifstat* to sample once the network bandwidth statistics.

5.4 Setting up script execution periodicity

After having exposed the different scripts and the configuration involved in the logic of the component, the balancer instances require that these scripts are executed following a periodicity; in particular, it is required a solution to allow the balancer to automatically detect changes in the upstream configuration (deltas in the bidders side) and to push the necessary metrics to the respective services every "dt" minutes. In order to achieve this logic, **Cron** is exploited, which is a Unix utility that runs as long-running process or daemon, meant to schedule tasks to be executed sometime in the future; this is normally used to schedule a job that is executed periodically. Cron exploits the usage of **crontabs**, special files that contain instructions for the *Cron* daemon; in other terms, *Crontab* provides to *Cron* the configuration of the periodicity for each script to be run. It follows a syntax example for *Crontab* row, comprehending the script and the time reference for the execution of the script itself:

minute hour day month weekday command

For instance, if an example script called "example.sh" contained inside /tmp folder that is needed to be run every hour, the following line would be provided to the Crontab:

*** */1 * * * run-this-one /tmp/example.sh**

with the character "*" meant as a wildcard (i.e. any value) and run-this-one as a wrapper script that runs no more than one unique instance of the provided command with a unique set of arguments.
The following lines report the *HAProxy* balancer instance's crontab content:

```
LOGS=/var/log/haproxy
ENVIRONMENT=production

# Sending HAProxy Balancer metrics
*/5 * * * * run-this-one /usr/local/bin/haproxy-metrics.sh
    >> $LOGS/cron-send-metrics.log 2>&1                    (1.)

# Automatic configuration of HAProxy.
0 */2 * * * run-this-one /usr/local/bin/sync-file
    $ENVIRONMENT-upstream-list.conf /etc/farms/
    >> $LOGS/cron-sync-haproxy-farms.log 2>&1              (2.)
*/1 * * * * run-this-one /usr/local/bin/sync-haproxy-upstreams.sh
    >> $LOGS/cron-sync-haproxy-upstreams.log 2>&1          (3.)
```

1. The script that contains the metrics logic, therefore every 5 minutes the script is called to retrieve fresh stats from *HAProxy* and push them to the Metrics aggregator services of the Platform.

2. *$ENVIRONMENT-upstream-list.conf* and */etc/farms/* are respectively the file and the folder where the information about the farms available in the region is stored. The script is triggered at the first minute of every two hours (i.e 00:00, 02:00, 04:00 and so on). It is a fundamental dependency of the ynamic update script to always have the farms of bidders available in the geographical region.

3. The textbfdynamic configuration with no down-time script. It is called every minute, to ensure that the configuration is up-to-date with the most recent bidders back-end setting.

In addition, each Cron job redirects its output to a log file, useful to debug the component in case of any faulty behaviour.
The following sequence diagram overviews the *Cron*'s triggering of the previously defined script:

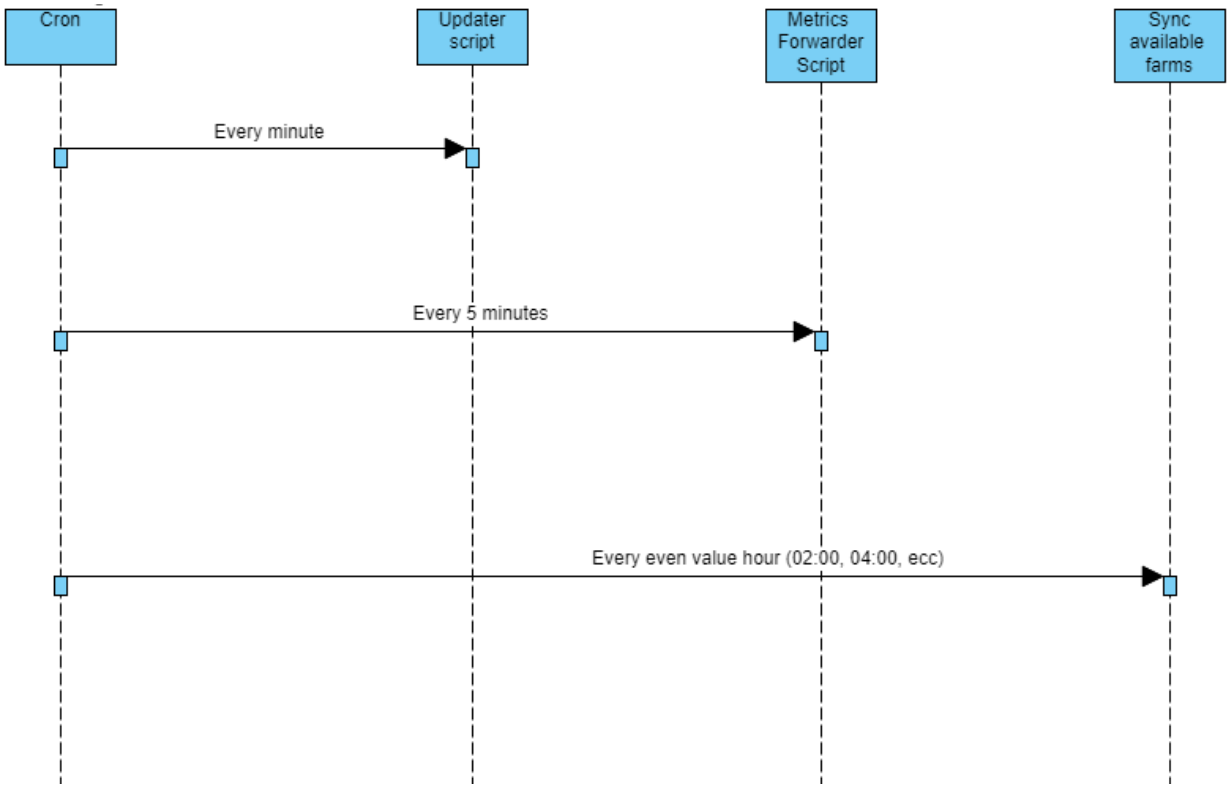


Figure 5.6: Cron overview

5.5 Deploying HAProxy in the Platform logic

Up to now, I have described the way to configure the balancer component and how to interact with it, from the initialization to the metrics logic, passing from the dynamic update of the configuration. But how do we build up and deploy the instance that will effectively run inside the Platform environment with all the previously defined files?
The current trend in software engineering is to perform **Continuous Integration and Continuous Delivery**, in other terms, *DevOps* and Software Engineering practises meant to improve software development speed and quality. They aim at building, testing, and releasing software with a view to rapidly put software into production and get it working as expected.
Specifically, *Continuous Integration CI* aims to give developers the charge of incorporating code updates into a single and shared repository, while textContinuous Delivery CD extends the concept of CI; code updates are incorporated in a shared repository, then built and reviewed before being released in the production environment. Improvements and/or changes to the code are built, checked, and packaged automatically before being released into production [28]. It is usually meant by means of *CICD* pipeline, made up of several steps, that is usually triggered once a code update happens, by the *Version Control System VCS*. Usual steps are to test, package the code into an artifact (usually a compressed zip file), push the artifact over a storage system and finally, disclosing the artifact by deploying it over a development environment. Following it is reported the meant CICD pipeline by means of an UML sequence diagram:

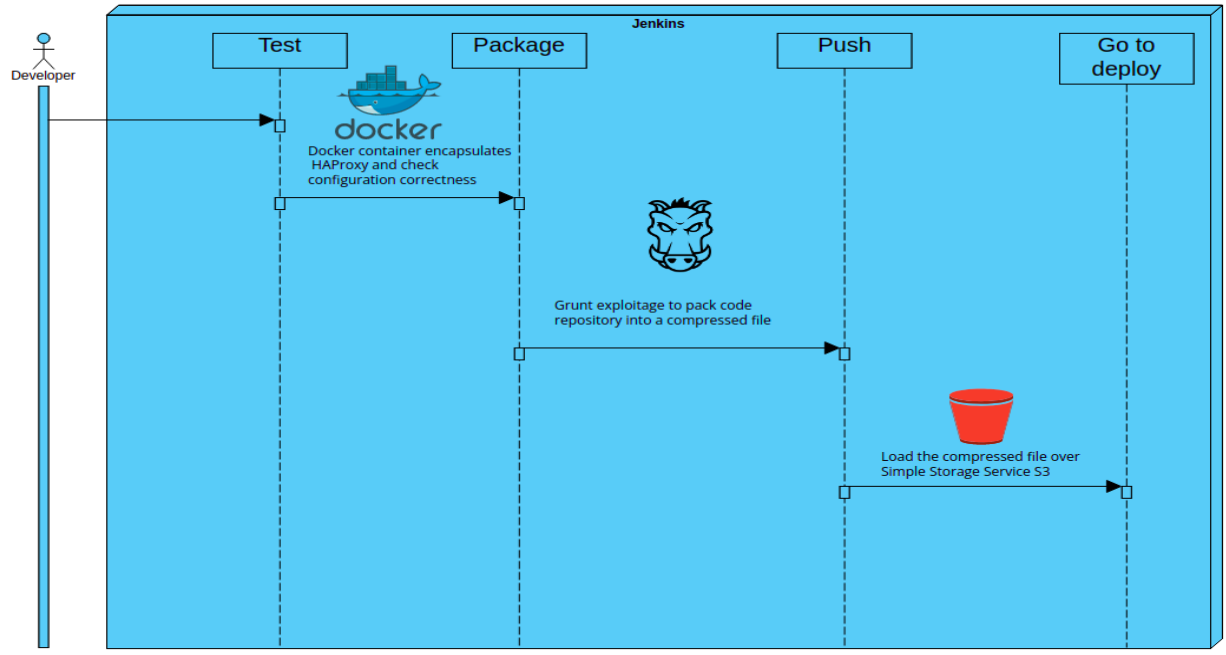


Figure 5.7: UML Sequence Diagram for deployment pipeline

The rise of *Jenkins* and other *Continuous Integration and Continuous Delivery* tools have unleashed a new trend for software deployment strategies, addressed as **pipeline-as-code**, that provides a way to define deployment pipelines through source code (for instance *Groovy* or declarative YAML approach), available in the same repository as ordinary source code, supplying a maintainable and improved way for modules deployment.
The following is my proposal for *pipeline-as-code*, implemented using **Jenkinsfile**, a file that contains the definition of a Jenkins Pipeline in form of source code. It is peculiar the way it is used over the repository, which do not contain properly code but **configuration-as-code** and configuration scripts:

```
def pushRegions(list) {
  for (int i = 0; i < list.size(); ++i) {
    sh "aws s3 cp --region ${list[i]}
        target/dist/balancer-haproxy.zip
        s3://region-${list[i]}/codedeploy/balancer/haproxy/
            ${VERSION}/balancer-haproxy.zip"
  }
}

def notifyOnFailure(myStage) {
  echo "Notifying failure in $myStage stage"
  def msg = "echo \"${myStage} stage failed for balancer
    version ${VERSION} (${BRANCH_NAME}).
    Click on the following link to check the build: ${BUILD_URL}\""

  if (env.BRANCH_NAME == "master") {
    slack.failure message: sh(script: msg.toString(),
        returnStdout: true)
  }
  else {
    withEnv(["SLACK_CHANNEL=${SLACK_USER_ID}"]) {
      slack.failure message: sh(script: msg.toString(),
        returnStdout: true)
    }
  }
}

pipeline {
  parameters {
    booleanParam(name: 'DEPLOY_TO_PRODUCTION', defaultValue: false,
        description: 'Check it to deploy this build to production')
    booleanParam(name: 'SKIP_TESTS', defaultValue: false,
```

```
      description: 'Skip unit and integration tests')
}

environment {
  SERVICE = "balancer"
  DEPLOY_TO_PRODUCTION = "${params.DEPLOY_TO_PRODUCTION}"
  DEPLOY_JOB = "balancer-prepare"
  SKIP_TESTS = "${params.SKIP_TESTS}"
  PACKAGE_VERSION = sh(script:'echo -n $(cat package.json
      | jq -jr .version)', returnStdout: true)
}
stages {
  stage('Test') {
    when {
      environment name: 'SKIP_TESTS', value: 'false'
    }
    steps {
      echo "Building version ${VERSION}"
      sh '''
        npm run docker:up
        npm run docker:haproxy:check || ERR=1
        npm run docker:down
        exit $ERR
      '''
    }
    post {
      failure {
        notifyOnFailure("Test")
      }
    }
  }
  stage('Package') {
    steps {
      sh """
        npm install
        grunt packageHAProxy
      """
    }
    post {
      failure {
        notifyOnFailure('Package')
      }
    }
```

```
    }
    stage('Push') {
      steps {
        script {
          if (env.BRANCH_NAME == 'master' ||
            env.DEPLOY_TO_PRODUCTION == 'true') {

            pushRegions(['us-east-1', 'us-east-2',
                'eu-central-1', 'ap-northeast-1', 'ap-southeast-1'])

          } else {
            pushRegions(['us-east-1'])
          }

          build job: "${env.DEPLOY_JOB}",propagate: true,wait: false,
          parameters: [
            string(name: 'VERSION', value: env.VERSION)
          ]
        }
      }
      post {
        failure {
          notifyOnFailure('Push')
        }
      }
    }
  }
}
```

The deployment pipeline for the balancer module starts with the **test** step; here the balancer module is tested, since it do not contain proper unit or integration tests, the stage just checks the *HAProxy*'s configuration syntax correctness. Once again, docker is exploited in order to mimic a sandbox where to fastly install and configure *HAProxy* with the previously provided configuration.
npm run docker:up and *npm run docker:down* boot up and shut down an *HAProxy* based docker container, while *npm run docker:haproxy:check* is the command in charge of checking the configuration syntax correctness.
Then the **package** step comes: the repository is parsed exploiting *Grunt*, a tool used to automatically perform frequent tasks such as compilation and compression of repositories, that copies and compress all the needed files and dependencies into a .zip file. The **push** stage instead loads the compressed repository over the geographical regions that host bidders and balancers. The global function defined at the beginning of the pipeline has been coded in order to provide a common method to load the zip file over AWS **Simple Storage Service**

S3. Finally, once loaded the zip file, the job ”balancer-prepare” is called in order to set-up the proper deployment over **AWS infrastructure-as-service EC2**.
CodeDeploy (another service offered by AWS) is exploited for the effective deployment and it is activated only by means of Jenkins' server graphical user interface trigger.
CodeDeploy is the service in charge of setting up the *EC2* instances, i.e. virtual servers, therefore it needs to be aware of all the steps required to boot an instance, installing all the required dependencies and scripts.
To summarise, here it is provided a resume of the deployment pipeline service components:

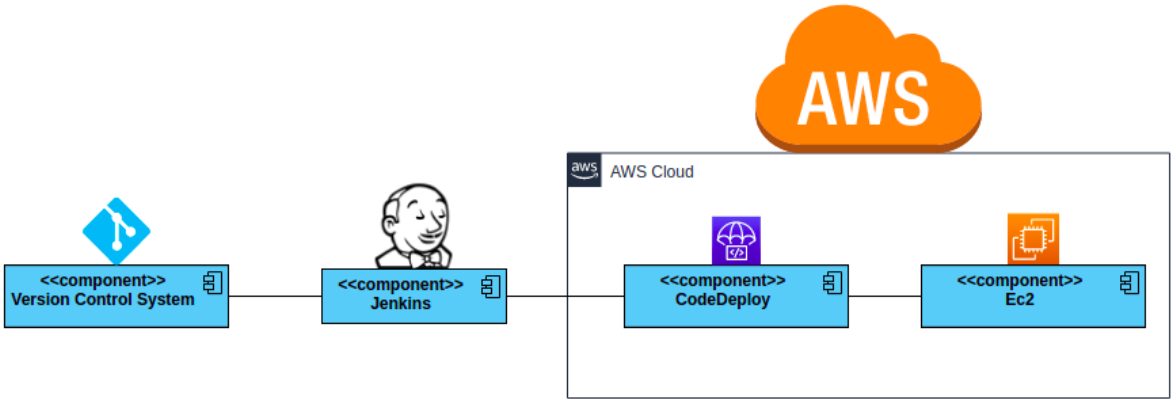


Figure 5.8: Deployment pipeline components

The following is the representation of the balancer **repository**, which contains all the required dependencies (such as the scripts described in the previous sections as well as *HAProxy* configuration file, TLS certificate and error files) including a folder called "deploy" that contains the script defined by the module developer and exploited by CodeDeploy to properly boot a virtual server.

```
balancer
├── bin
│   ├── haproxy-metrics.sh
│   ├── init-haproxy.sh
│   └── sync-haproxy-upstreams.sh
├── deploy
│   ├── stop
│   ├── clean
│   ├── setup-root
│   ├── configuration-reload
│   ├── set-crontab
│   ├── start-other-services
│   └── status-check
└── files
    ├── cron
    │   └── crontab-production.txt
    └── haproxy
        ├── haproxy.cfg
        └── ssl
            └── server-certificate.pem
```

Each script inside "deploy" folder defines a step for the instance system set-up, considering that a virtual server must be launched over EC2 a priori:

- **stop**: stops the service cron, in this way every periodical update is stopped until cron is restarted. It allows to perform the deployment without affecting the metrics services hosted on the platform's cloud, which might be forwarded during the deployment process.

```
service cron stop
```

- **clean**: delete the content of Unix system folders that contain component's fundamental scripts, binaries and files in order not to have noise from other previous deployments.
- **setup-root**: set-up system environment variables (such as the geographical region the instance is running on and the environment (production or staging)) and installs *HAProxy* and Statsd services.
- **configuration-reload**: the aim of this script is to mock at the initialization stage the content of the previously described *Crontab*. it calls the same scripts as *Crontab* to load

the farms available in the region, initialize *HAProxy* by means of *initializer* script and finally updates the configuration through *updater*.

```
run-this-one /usr/local/bin/sync-file
    $ENVIRONMENT-upstream-list.conf /etc/farms/
run-this-one /usr/local/bin/init-haproxy.sh
run-this-one /usr/local/bin/sync-haproxy-upstreams.sh
```

- **set-crontab**: in this step, according to the ENVIRONMENT variable, the respective *cron* instructions are loaded inside Crontab for user Root. This step guarantees that the instance is loaded with script running following the defined periodicity.

```
touch /tmp/crontab-file
cat $CRONS_PATH/crontab-${ENVIRONMENT}.txt
    > /tmp/crontab-file
sudo crontab < /tmp/crontab-file
```

- **start-other-services**: the system is ready and loaded with all the required dependencies, therefore it is time to restart the previously stopped *cron* service and *Statsd*.

```
service cron restart
service statsd restart
```

- **status-check**: this is the final check to sample the healthiness of the component. The configuration syntax is tested again in order to check if all the set-up stage ended smoothly. As defined in *HAProxy* configuration, the balancer exposes through the main frontend the resource /ready that is set to return *200 status code* by default. If *HAProxy* has been installed, initialized and configured properly and then started up correctly, the /ready endpoint would be reachable by means of HTTP request, as described in the following code snippet:

```
function checkHttpEndpoint() {
  local HEALTHCHECK=$1
  echo "Checking status of $HEALTHCHECK"
  sleep 10
  status=$(curl -X POST --write-out \\n%{http_code}
        --silent --output /dev/null "$HEALTHCHECK")
  [ "$status" -eq "200" ] || exit 1;
}

haproxy -c
[ "$?" == 0 ] || exit 1
checkHttpEndpoint 'http://localhost/ready'
```

If all the previous script don't fail, the deployment is disclosed in every available *HAProxy* instance over the Infrastructure service hosted on *AWS*.

6 METRICS VISUALIZATION AND MONITORING

In order to effectively validate if the component is behaving in the expected manner or even improving the current solution installed in the Platform, it is fundamental to provide some way to graphically visualize and evaluate metrics; in other terms the creation of a **dashboard** is therefore required. A dashboard is a display of data, where its primary intention is to provide information immediately upon looking, such as Key Performance Indicators. In our strict case, the set up of a dashboard would help to gather an immediate insight of a bidding region, whether the balancer and the respective bidders in the geographical area are behaving in the expected way, defining a way to collect several metrics under **widgets**: a dashboard is an aggregation of widgets, or in other terms, visual representations of individual informative statistics about healthiness and performance. As discussed previously, the purposed *HAProxy* solution has been designed providing a way to export metrics, endpoint to endpoint from *HAProxy* stats page and instance to *AWS* metrics pool, by means of the script presented in section 5.3. In such a way, we ensure that the service *CloudWatch* is always provided with the latest information related to the component, defining a data pipe from the component itself to the cloud service.
The performances of the balancer affects the relative back-end and the respective bidder's performances, therefore the dashboard is meant to collect metrics coming from different components (the balancer and the bidder) with a view to observe the effect of the *HAProxy* configuration over the bidders' efficiency. But which are explicitly the information we want to display and monitor?

- **Total requests**: the number of total requests processed by the balancer. It is one of the indicators of traffic flowing in the component and possibly suggests if *HAProxy* is suffering processing requests. A drop down of total request would mean that something wrong is happening with the *HAProxy* configuration or the instance itself.

- **Total connections**: current amount of parallel connections with the component. *HAProxy* is configured to open *Keep Alive connections* with both client and server sides: noticing big spikes in the connection count would mean that keep-aliveness is not respected or not working properly, hence badly affecting the functionality of the component and the Platform bidding process itself. Moreover, alongside *total requests*, it defines the way to understand the type and the amount of traffic each bidding region is handling. It will be useful later, to define the test scenarios for the deployment of the balancer component.

- **Bidders latencies**: the overall timing information about bidders in one farm, grouped by expected round-trip time latency, reporting information about percentile 95 and

average time for bids round trip (advertisement request time plus bid response time) and bid response. This kind of metric is really useful to sample the quickness of the back-end side to respond to incoming bids, with a view to the probability distribution. A lower variance in response time would mean increasing the quality of the Platform responses. Following the results obtained at section 4.4, the implementations of the service provided by *HAProxy* should lower on average the response times of the Platform, deploying a more robust solution meant to lower the distribution of times with respect to the current solution based on *Nginx*.

- **Shedding**: it is an indicator to observe the delayed responses over a bid-request. A region showing high Shedding's values means that multiple bid requests are responded out of time, forcing the bidders to forward a response featured by code *204 No Content*, therefore losing the possibility to actively participate to the auction related to the incoming bid. It is an indicator that describes the quality of service of the platform.

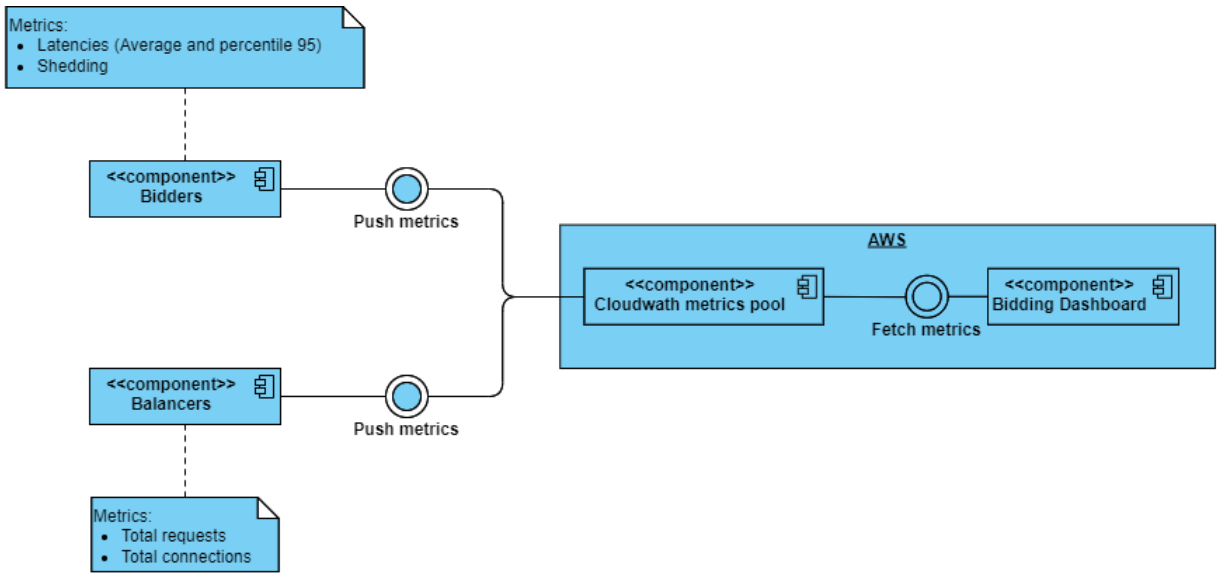


Figure 6.1: Agents forwarding metrics to CloudWatch

6.1 Implementation of the bidding dashboard

Amazon Web Services AWS provides a way to define and create resources starting from a template, i.e. a pattern host on a file usually of *YAML* or *JSON* format, meant to provide a uniform way to configure services in a predictable and pre-defined way.
Templates are the basis for *CloudFormation*, another service provided by *AWS*, meant to digest this kind of pattern files in order to define a "stack", a collection of *AWS* resources manageable as a single unit; in other terms, stacks contain a set of resources to be provided

to a specific *AWS* service through *CloudFormation*'s updates. Therefore, coding in a declarative way a template allows developers to define a resource, then trigger the *CloudFormation* update with the aim to create the given resource over a given cloud service, which in our specific case is *CloudWatch*.

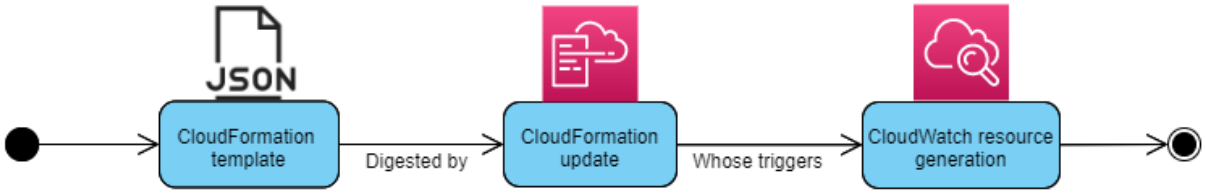


Figure 6.2: CloudFormation update process

Moreover, the dashboard is meant to be deployed in every bidding region, therefore it must parametrize the AWS region value in order to uniquely define a resource in a single template, but for multiple geographical areas.
In detail, here it is provided the way to set up the Bidding Dashboard over a JSON template:

```
"BiddingDashboard": {
  "Type": "AWS::CloudWatch::Dashboard",
  "Properties": {
    "DashboardName": { "Fn::Sub": "bidding-${AWS::Region}" },
    "DashboardBody": { ... }
}
```

AWS::CloudWatch::Dashboard [26] furnishes the pattern to build up a Dashboard resource that accepts some properties values:

- The **dashboard name**, i.e. a unique resource naming over the Cloud. Using the intrinsic function *Fn::Sub*, which substitutes variables in an input string with values available during the update of a stack. In this case, the region value will be available only during *CloudFormation* update of a defined region, therefore must be defined as a "update-time" parameter.
- The **dashboard body**, the actual content of the dashboard where widgets will be hosted. Following *AWS* documentation, the *DashboardBody* [27] property must be of type *JSON* String. Since the widget definition follow a *JSON* objects syntax, a way to commute an object to a string must be defined.

Going deeper, here it is provided the widgets definition in JSON format for the total connection and requests received by the load balancers in the given region and for the bidders latency values (the code snippet reports only the latency for the 200msB farm, since the others are exact replicas of the one reported below, but with different values for expected latency and farm):

```
"DashboardBody": {
  "Fn::Join": ["\n", [
    {",
      "\"type\": \"metric\",",
      "\"properties\": {",
        "\"metrics\": [",
          "[ \"kpis\", \"KPI-Balancer-ConnectionCount\" ],",
          "[ \"kpis\", \"KPI-Balancer-RequestCount\" ]",
        "],",
        "\"view\": \"timeSeries\",",
        "\"stacked\": false,",
        { "Fn::Sub": "\"region\": \"${AWS::Region}\"," },
        "\"stat\": \"Average\",",
        "\"period\": 300,",
        "\"title\": \"All Balancers 100ms, 200ms & 300ms\"",
      "}",
    "}, {",
      "\"type\": \"metric\",",
        "\"properties\": {",
          "\"metrics\": [",
            "[ \"kpis\", \"KPI-Bid-Count\",
              \"Latency\", \"200msB\" ],",
            "[ \"kpis\", \"KPI-Bid-Response-Time-Mean\",
              \"Latency\", \"200msB\" } ],",
            "[ \"kpis\", \"KPI-Bid-Response-Time-P95\",
              \"Latency\", \"200msB\" ],",
            "[ \"kpis\", \"KPI-Response-Time-Mean\",
              \"Latency\", \"200msB\" ],",
            "[ \"kpis\", \"KPI-Response-Time-P95\",
              \"Latency\", \"200msB\" ],",
          "],",
          "\"view\": \"timeSeries\",",
          "\"stacked\": false,",
          { "Fn::Sub": "\"region\": \"${AWS::Region}\"," },
          "\"title\": \"Latency 200msB Bidders\",",
          "\"stat\": \"Average\",",
          "\"period\": 300",
        "}",
    "}, {",
      "\"type\": \"metric\",",
        "\"properties\": {",
          "\"metrics\": [",
```

```
                "[ \"kpis\", \"Shed\",
                    \"Latency\", \"100ms\"],",
                "[ \"kpis\", \"Shed\",
                    \"Latency\", \"200ms\"],",
                "[ \"kpis\", \"Shed\",
                    \"Latency\", \"300ms\"]",
            "],",
            "\"view\": \"timeSeries\",",
            "\"stacked\": false,",
            { "Fn::Sub": "\"region\": \"${AWS::Region}\"," },
            "\"title\": \"Shedding\",",
            "\"period\": 300,",
            "\"stat\": \"Sum\"",
        "}",
    "}",
]]
```

The function *Fn::Join* (another intrinsic function, similar to *Fn::Sub*) allows strings chaining, providing the character \n (i.e. new line) as separator for all the following comma-separated strings; in such a way, the *JSON* string type for DashboardBody is respected.
The "metrics" property instead, define the collection of statistics the widget will show. Here it is specifically reported the widget that will contain data coming from balancers and exported with the script at 5.3; each metric is defined under a "namespace" (a set of conceptually similar metrics defining a metrics pool), in this case kpis, and by a unique name.
Each widget embeds the region value, therefore the region must be parametrized using *Fn::Sub*; in this way, each time we reach a region with a *CloudFormation* update, the metrics related only to the region particularly will be displayed.

6.2 Effects after deployment: metrics' visual debug

In the previous chapters and sections, I explained how to configure and the requirements needed to configure the balancing solution compatible with the Company needs and the bidding process for *RTB*, as well the major benefits of *HAProxy* have been sort out, pointing it as the best component in order to reduce back-end average latency both in terms of *HAProxy* technical specification (tested in a benchmark at section 4.4) and algorithms specification, with possibility to have *Random 2* algorithm and all the benefits it carries for the load balancing strategy (as pointed out in Preliminaries 3.5) but which are the real effects of the *HAProxy*'s implementation?
Since the component interacts with HTTP traffic to be dispatched, it's performances might be affected by the features of the HTTP traffic itself, depending on the *AdExchanges*' HTTP policy, which might open towards a specific farm, multiple connections for few requests or few connections highly exploited by various requests: the shape of the traffic might change the expected outcome. More in general, the amount of traffic managed by a farm of server highly

differs too, therefore we might notice different behaviors for the reported metrics, according to the density of traffic: some farms and geographical bidding regions are less stressed, while others handle high HTTP loads, therefore for this reason, the results might change too. Hence, the following section will report two cases:

1. a **best-case scenario** test will be conducted, over a scarcely loaded region, in order to build up the expectations that this research aims to propagate over all the other geographical regions and farms. From an operational point of view, it is required to deploy the component first in a less-congested farm, where the balancer component will be less stressed.

2. Then, the focus will shift over a highly stressed farm, with the hope to show significant improvements even with non-optimal pre-conditions (bad shape of traffic and high number of requests per second), in order to check if a **general-case scenario** follows the best-case one.

6.2.1 Effects after deployment: best-case scenario

Let's start deploying the new balancer component over a low-congestion region featured by an optimal shape of traffic.

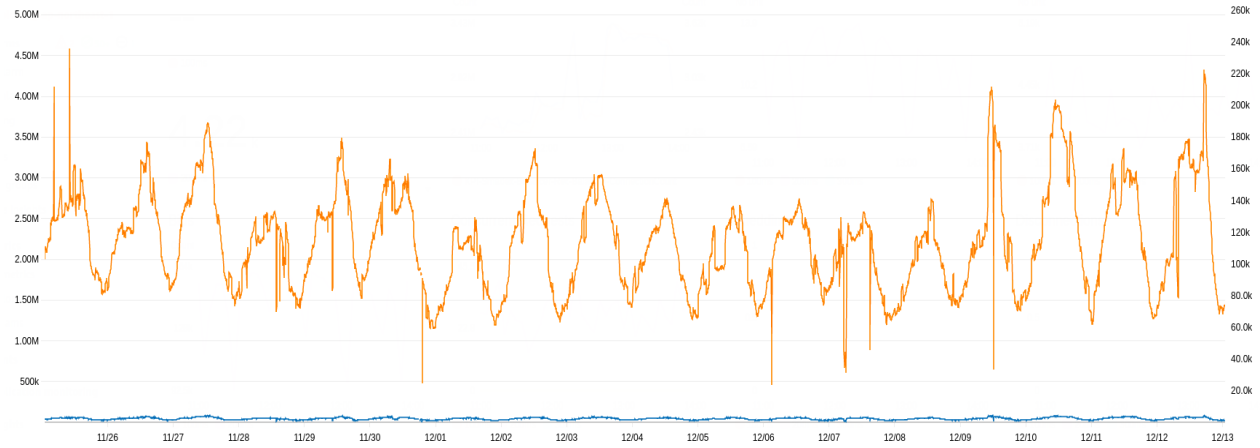


Figure 6.3: Region's shape of traffic: received Connections in blue, Requests in orange

The previous picture describes the traffic over the bidding region. It shows that the connections are in the order of magnitude of thousands (few thousands of connections accepted by balancers), while the incoming requests usually scale from approximately 1 to 4 milions. This region reports the lowest amount of requests over the platform's geographical areas and the traffic shape is considered the optimal one, since few connections manage a high amount of HTTP requests.
Once the component has been successfully deployed over a bidding region, the dashboard

and its metrics widgets come to the aid for this purpose, underlining the long and short term effects over the statistics of a geographical bidding region.
First, let's check the back-end latency to see if *HAProxy*'s deployment has changed the overall behavior of the component during time:

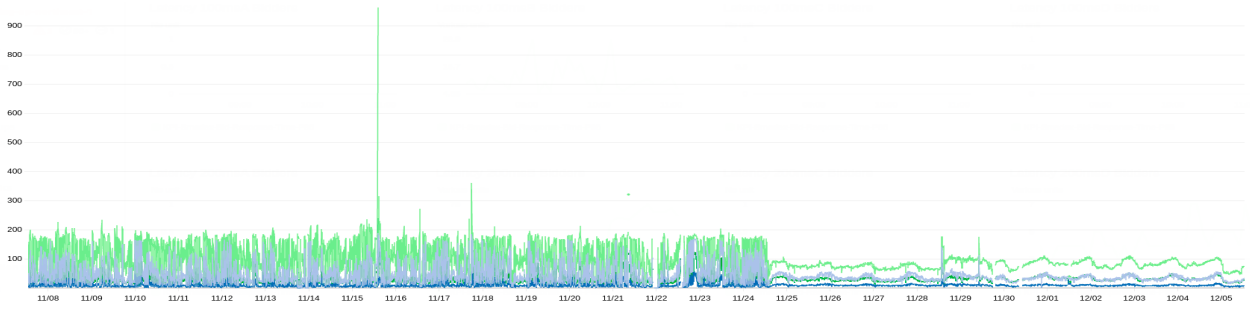


Figure 6.4: Farm's latencies (response time and bid-response time: average, percentile95)

From the previous picture, the metrics regarding a farm of bidders are reported, in particular the bid response time (i.e. the total time for the bid request to reach the back-end server from the balancer plus the bidder processing time plus the trip back to the client). This farm specifically interacts with *AdExchanges* which expects the total round trip time not to exceed 200ms.
The latency metrics shew in this particular farm, maximum values (expressed as percentile 95) of almost 200ms with peaks that exceed the threshold hitting in some cases almost a second of packet round trip time. *HAProxy*'s deployment happens November 24th and it is possible to observe how the metric related to the latency of the farm experienced a drop down, with latency times almost halved with respect to the scenario pre-deployment. *HAProxy* and Random 2 together can effectively **reduce the variances of response times**, both for average and percentile cases, showing an overall better management of the farm's back-end that results in a more compact, uniform and fair distribution among servers. This behavior is in-line with the expectation set by the performance test and the discussion over load balancing methodologies.
On the other hand, reducing the effect of the latency over servers brings another secondary but equally important improvement. Bidders are implemented over the Cloud infrastructure exploiting an **Auto Scaling Group**, a term often used in *Infrastructure-as-Code* templates to point to cluster of entities able to scale up or down automatically depending a pre-defined rule; bidders specifically exploit the latency metrics to scale up and down: an *auto scaling group* showing high latency values will be scaled up to allow the back-end to better distribute the traffic and providing the expected quality of service, or in case of really low response times, the Auto Scaling Group scales down in order to cut off extra costs related to under-utilization of instances. Better back-end utilization through the proposed balancer allows the servers to respond better, avoiding the need of booting other extra server instances to improve the quality of service. In a cloud infrastructure, where billing is applied on a utilization time basis, cutting off the number of virtual servers would mean decreasing their

relative cost: **less server instances needed means decreasing on average the cost of Cloud infrastructure**, which is considered a crucial KPI for the Platform (less cost having the same performances means increase of the revenue).

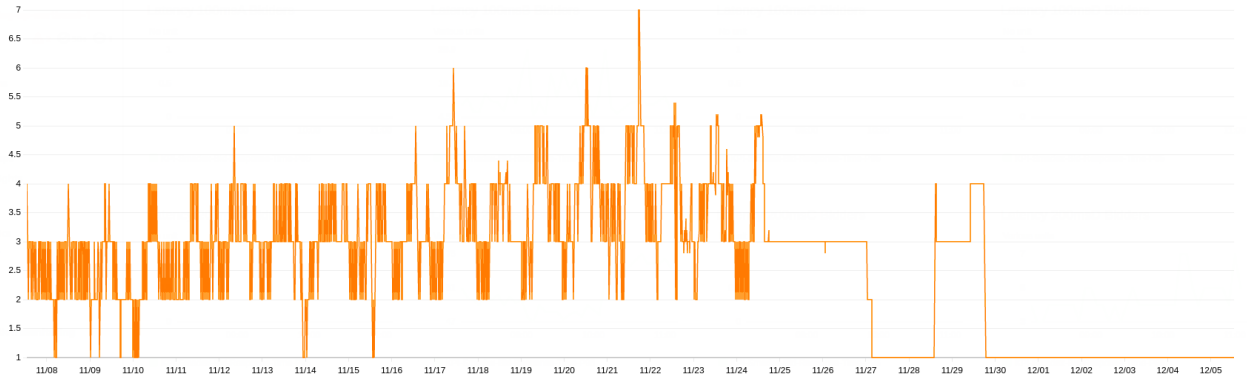


Figure 6.5: Farm's number of servers

The number of servers metric for the farm shows a steep decrease in number, as well as the variability of this metric after the deployment. If servers response time shows spikes and peaks, then the number of server instances will be affected too, and the infrastructure will start booting servers to compensate those latency issues. Adding and removing servers might be a drawback as well for the Platform, since the establishment of new connections with the new born servers adds an overhead in the CPU of the balancers and the bidders and therefore might affect the global quality of service, and should be done only when it is strictly necessary to add extra processing power. For this farm specifically, the number of server instances is decreased from the pre-HAProxy worst-case utilization on 21/11 (from 2 to 7 instances) up to a single stable entity, setting itself as a reference to replicate in other farms. Moreover, the Shedding value as well show drastical improvements:

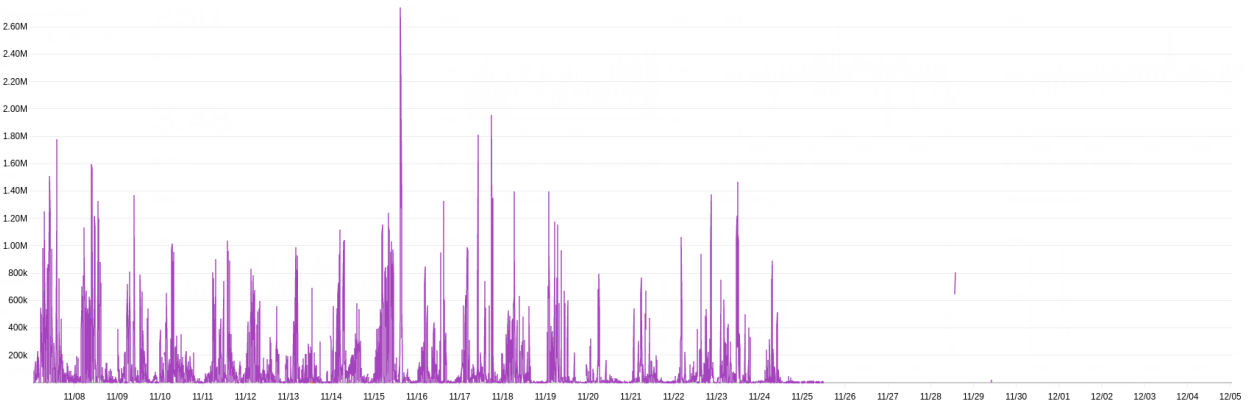


Figure 6.6: Observed shedding in the region

reducing the spread of response times distribution helped the Shedding value as well, which value dropped down after the component's deployment. This means that the possibility to actively participate to an auction increased drastically, thanks to the reduced overall response times; therefore, the **platform quality of service increased**.
In conclusion, getting uniform responses from the servers helps the infrastructure itself to better scale back-ends, with an overall trend to decrease the total number of instances needed to manage the same amount of bids; in addition, having less virtual servers instances over the infrastructure reduces the total infrastructure cost, in this case only the cost of one instance instead than the worst case scenario pre-HAProxy deployment of 7 instances. Moreover, following the Shedding decrease, the possibility for a bib response to take active part of the auction increases drastically.
This experiment has defined the possible benefits observable from a less-congested region, setting the outcomes that each region would be expected to follow.

6.2.2 Effects after deployment: general-case scenario

Following the outcomes of the previous section, this part of the analysis will focus on non-optimal regions featured by an higher load of requests and various shapes of traffic, focusing on a *generic-case scenario*, i.e. a replica of the experiment set in the previous section, to validate if the migration to HAProxy has brought the wanted and already observed benefits. Let's take as reference a region featured by the following network traffic:

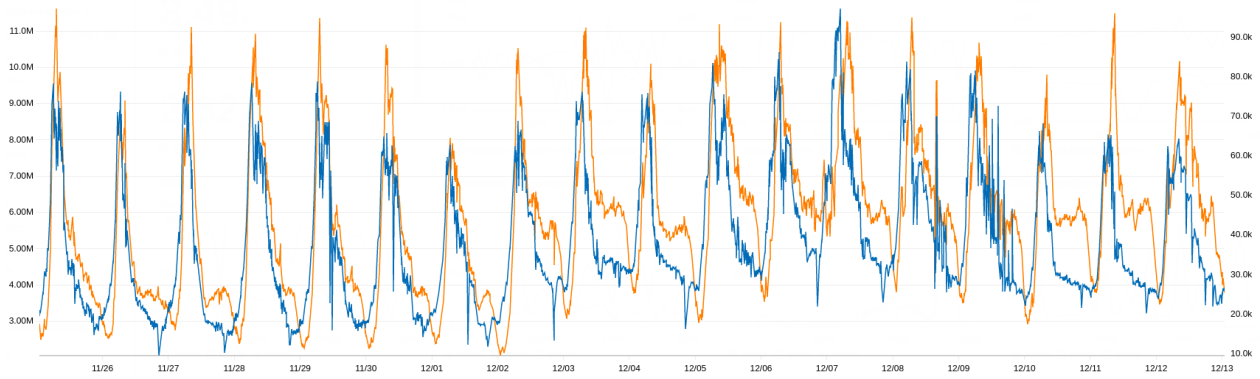


Figure 6.7: Region's shape of traffic: received Connections in blue, Requests in orange

The region shows an increased load of HTTP traffic over the platform's balancers, with requests hitting peaks of requests exceeding 11 million value (more than the double with respect as before) and values of connections varying from 10 thousands to 90 thousands current active requests (approximately 10 or 90 times more than the previous best-case scenario example). Moreover, the connections highly varies during time and this might cause higher overhead over the system resources, since the plot suggests that connections are opened and closed way more frequently with respect to the best-case scenario.
Let's now repeat the steps estabilished previously, in order to check if the latencies, the number of instances and the shedding have improved.

Figure 6.8: Latencies from the analyzed farms (response time and bid-response time: average, percentile95)

Taking two farms from the same region, it is possible to notice that after the deployment happened on 16th of December, the latencies haven't decreased hugely as before for the first analyzed farm; despite an imrpovement , the contribution provided by *HAProxy* has anyways furnished a better distribution of the response times, reducing the possibility to encounter spikes while responding to bids and decreasing the response time of a couple of tenth of milliseconds.
On the contrary, the second farm widely reduced its response times, showing huge improvements over the latency.

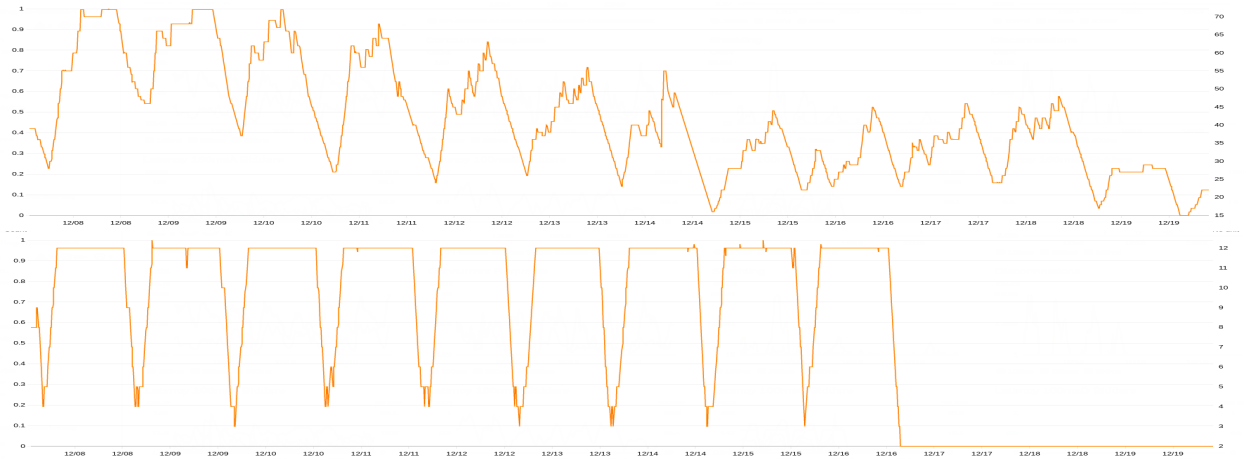


Figure 6.9: Number of servers from the analyzed farms

Continuing the discussion, the number of instances belonging to a farm is dependent on the latency that affects the back-end. A slight reduction of times would mean a small reduction of instances, on the contrary, a significant decrease of times would mean a drop down of the bidder instances utilization. The number of instances from the two farms under discussion is

reported. It is possible to notice how the first farm servers start scaling after the deployment in a range that goes from 15 to 49, respect to the scenario pre-deployment where the instances number were scaling from a minimum of 15 to a maximum of 70. The second reported farm instead, shows once more a steep decrease in the number of servers, with the overall amount of servers that stabilize over two bidder instances. Once again, the expectation set in the best-case scenario are met.

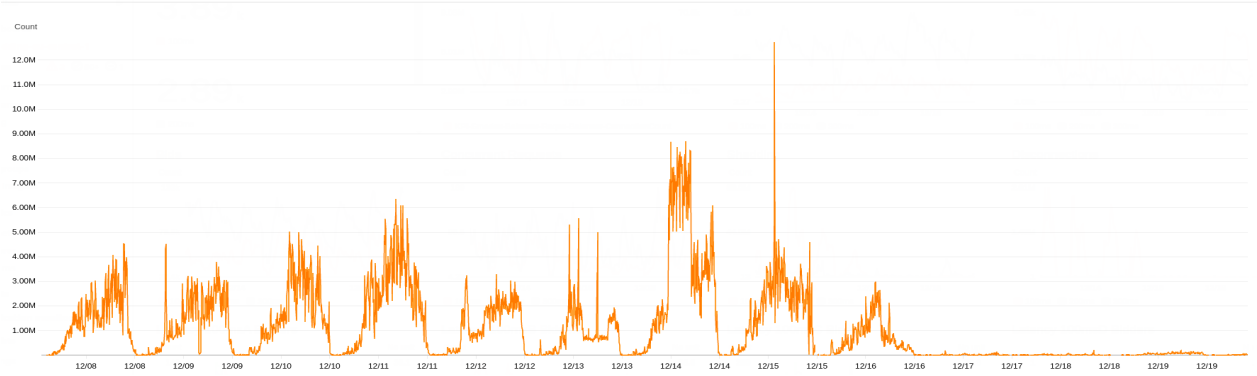


Figure 6.10: Observed shedding in the region

Finally, also the shedding value follows the one estabilished in the best-case scenario, showing negligible values once the component reaches production environment.
This outcome underlines how the traffic sent by AdExchanges highly varies, at the point that the magnitude of the benefits brought by *HAProxy* depends on the feature of the traffic itself. On average, it has been possible to show that the configuration provided based on the analysis of the best component available in the market with respect to the requirements declared in section 4.1.

7 CONCLUSIONS

I have highlighted throughout this work that the RTB process and its metrics are affected by the type of the selected *reverse proxy*, made possible through the software selection process and the kind of load balancing algorithm specified. In particular, I demonstrated that through software engineering' best practices, such as performance software benchmarking and analysis of requirements through an evaluation metrics, it is possible to establish which *reverse proxy* fits the best with respect to the *RTB* demand of a *DSP*.
Following the guide-lines instituted by chapter 5, it has been provided how to configure the best-fitting component, furnishing the way to define the configuration creation in order to mask the bidder side from the *AdExchanges* one, supplying also some tuning tips and cyber security warnings against threats that might affect or disrupt the performances of the platform itself. Code reporting by means of *Unified-Modeling-Language UML* diagrams, such as component, use-case, activity and sequence diagrams have been highly exploited to overview the reported code snippets' meaning from an high-level point of view, in order to underline the logic and functionalities behind every script or configuration file.
Furthermore, a steering from the speech has been made, focusing on *Continuous Integration and Continuous Delivery CICD* and my purpose of implementation for it through *pipeline-as-code*.
Finally, after the development of a dashboard to resume the RTB process main statistics, it has been possible to observe the various effects after the selected balancer's deployment over production metrics. The assumption established by the load balancing algorithm selection (section 3.5.2) and by the *reverse proxy* benchmark analysis found confirmation in the empirical results obtained at section 6.2. To summarise, the following has been demonstrated:

- *Power of Two* algorithm combined with the selection of the most performant component guaranteed a better management of the bidder side, consequently a **reduction of the response times latency**, which have been observed in general to keep a more stable behavior after the *HAProxy* deployment, featured by lower response time variance as it can be noticed by the reduction of the average and percentile 95 response times.

- Aside the reduction of the average response times' latency, it is noticeable a decrease in the utilization of cloud infrastructure's virtual server utilization, which varies from farm to farm and from region to region according to the reasons cited previously in the discussion, such as load factors (measured as number of incoming requests) and traffic shape (number of connections vs number of requests sent over those connections). Less virtual servers to maintain means **improved costs related to the utilization of the cloud infrastructure and AWS billing**.

- Responding faster also means to **improve the quality of service**: after the deployment of *HAProxy*, the *shedding* value has widely decreased, meaning that the aggre-

gated amount of bid requests that are responded out of time (therefore not taking part to the auction) has been significantly reduced, underlining how a more compact and uniform distribution of requests provides an improved reaction to incoming bids, guaranteeing much higher possibilities to take part in the auction estabilished by one of the *AdExchanges*.

Moreover, many tools, softwares and programming languages have been exploited during the development of this project: performance test exploiting *wrk2* over a *docker* environment, *CICD* with *Jenkins server*, configuration of various *reverse proxies*, learning how to work with *AWS* Cloud and the insight on its services, *BASH* scripting and learning how to tune and configure *HAProxy* have been studied during the flow of this master thesis.

7.1 Acknowledgments

I would like to thank the following people, without whom I would not have been able to complete this research, and without whom I would not have made it through my masters degree!
The Tech team at Smadex SLU, in particular my supervisor, Software Engineer Edraí Brosa, whose insight and knowledge into the subject matter steered me through this research. Special thanks to John Hearn, whose work and interest over the load balancing algorithm in *RTB* topic paved the way for the concept explained in this project. Thanks to my professors in FIB and Politecnico di Torino, Javier Larrosa and Andrea Bottino, that followed my work and helped me to develop this written document.
I would love to spend a couple of words about the unfortunate conditions every student is living during these hard days, with the pandemic and the smart-working condition everyone has been facing: those hard times increased the difficulties to learn new concept and technologies by first hand and have left in many people the feeling of abandon and loneliness, especially for those facing new challenges and new working experiences; on the other hand, these months have widely increased my self-learning abilities, helping me to improve my skills over new concepts, tools, programming languages and infrastructures that before this experience I would never be able to tackle by myself. To conclude, my biggest thanks go to my family for all the support you have shown me through this experience, the culmination of a year and half of Double Degree experience abroad, in particular to my mother that has always been there listening to my issues and worries.
Last but not least, thanks from the deepest of my heart to my wonderful girlfriend Sara, that has been on my side for all the time, leaving her home place to move to another country, supporting me over good and, especially, bad times: sorry for being even more bad-tempered than normal during this thesis writing, I owe you a travel wherever and whenever you want!

8 GLOSSARY

- **Amazon Web Services AWS**: One of the leaders in the market in the field of *cloud computing*, it furnishes more than 150 services related to computing, deployments, storage, databases, machine learning and many others. Among the services mentioned in this paper:

  - **Elastic Compute Cloud EC2**: one of the main products of Amazon Web Services, it allows the rental of virtual servers (also addressed as virtual machine), where to execute software modules and application. Users can acquire cloud instances from this service paying for the time of utilization of the given instance; each instance costs more or less depending on the instance type. *Elastic Compute Cloud* is generally identified as Amazon's *Infrastructure as a Service (IaaS)*.
  - **CloudFormation**: the service dedicated to provide *Infrastructure as Code* ability. It allows, through the coding of templates in *JSON* or *YAML* formats, to create resources over the cloud infrastructure. Many kind of resources can be created over the cloud, in this paper in particular, it has been described how to model a Dashboard for the *CloudWatch* service.
  - **CloudWatch**: the service dedicate to monitor and optimize resource utilization. It gathers statistics from other services hosted in the cloud and allows developers to troubleshoot and set alarms over the hosted metrics.
  - **Codedeploy**: AWS CodeDeploy is a fully managed deployment service that automates software deployments to a variety of computing services like *Amazon EC2*.
  - **Simple Storage Service**: product meant to provide storage features. It can be exploited to store any type of object, which allows for uses like storage for Internet applications, data lakes for analytics, data archives and backup and recovery.

- **Docker**: it is a container management service. The keywords of Docker are develop, ship and run anywhere. The whole idea of Docker is for developers to easily develop applications, ship them into containers which can then be deployed anywhere. It allows to generate containers from *Docker images*. It provides virtualization and, since containers are isolated, it guarantees security and it allows multiple containers to run simultaneously on the given host.

- **Docker image**: a Docker image is a file used to define a template for booting up containers, providing a set of instructions meant to build the container. In other terms, it executes code in a Docker container. A single Docker image can be re-used multiple times for containers generation.

- **Docker Compose**: it is a tool for defining and running multi-container Docker applications. With Compose, you use a *YAML* file to configure your applications services. Then, with a single command, you create and start all the services from your configuration. Therefore, in a single shot, it triggers the deployment of several containers that can communicate over the same network.
- **Grafana**: Grafana is an open source solution for monitoring, running data analytics, pulling up metrics that make sense of the massive amount of data and to monitor apps with the help of customizable dashboards.
- **Grunt**: it is a tool also identified as task runner. It is usually meant to execute frequent activities, such as repositories' linting, compression, compiling and unit testing.
- **Groovy**: object oriented programming language. It is usually addressed as an alternative for Java. Through Groovy, it is possible to define *pipeline-as-code* files, useful in this paper to define a way to deploy software modules exploiting *Jenkins*.
- **Statsd**: it is a network daemon written in Node.js to collect, aggregate, and send developer-defined application metrics to a separate system for graphical analysis. It creates a data pipe from the instances to be monitored to a monitoring service. In this thesis flow, it is mentioned to be coupled with *Grafana*.
- **Jenkins**: written in Java, it is an open-source automation server meant to provide *Continuous Integration and Continuos Delivery*. It automates building, testing and deploying steps. **Jenkinsfile**s might be associated to software modules in order to provide to Jenkins server the steps coded in *Groovy* in forms of instruction through *pipeline-as-code* templates.
- **Privacy Enhanced Mail PEM**: file format typically used to store encryption information. It is container format that may include just the public certificate or may include an entire certificate chain including public key, private key and root certificates.

Bibliography

[1] J. Hearn, (April 3rd, 2021), "Load Balancing Strategies and their Distributions". Retrieved from https://john-hearn.info/articles/load-balancing-strategies

[2] Y. Yuan, F. Wang, J. Li and R. Qin, (2014), "A survey on real time bidding advertising", Proceedings of 2014 IEEE International Conference on Service Operations and Logistics, and Informatics, pp. 418-423.

[3] IAB Technology Lab, (December 2016), "OpenRTB API Specification Version 2.5", Real Time Bidding (RTB) Project. Retrieved from https://iabtechlab.com/wp-content/uploads/2016/07/OpenRTB-API-Specification-Version-2-5-FINAL.pdf

[4] J. D. Day, H. Zimmermann, (January 1984), "The (Un)Revised OSI Reference Model", Proceedings of the IEEE (P IEEE)

[5] Peter Sommerlad, (2003), "Reverse Proxy Patterns". Retrieved from https://hillside.net/europlop/HillsideEurope/Papers/EuroPLoP2003/2003_Sommerlad_ReverseProxy

[6] Art Stricek, (January 10, 2002), "A Reverse Proxy Is A Proxy By Any Other Name". Retrieved from https://www.miga.org/sites/default/files/archive/Documents/reverseproxy22.pdf

[7] Willy Tarreau, (February 2019), "Test Driving Power of Two Random Choices Load Balancing". Retrieved from https://www.haproxy.com/blog/power-of-two-load-balancing/

[8] T. Schlossnagle, (2017,) "Monitoring in a DevOps world". Retrieved from https://queue.acm.org/detail.cfm?id=3178371

[9] HTTP2 official Github page, "HTTP/2 Frequently Asked Questions". Retrieved from https://http2.github.io/faq/#does-http2-require-encryption

[10] IBM Documentation, "Health checks". Retrieved from https://www.ibm.com/docs/en/datapower-gateways/10.0.1?topic=groups-health-checks

[11] Docker official web-site. Retrieved from https://www.docker.com/

[12] Overview of Docker Compose. Retrieved from https://docs.docker.com/compose/

[13] HAProxy official web-site. Retrieved from http://www.haproxy.org/

[14] Nginx official web-site. Retrieved from https://www.nginx.com/resources/glossary/nginx/

[15] Traefik official web-site. Retrieved from https://doc.traefik.io/traefik/

[16] Envoy official web-site, "What is Envoy". Retrieved from https://www.envoyproxy.io/docs/envoy/latest/intro/what_is_envoy

[17] "Apache HTTP Server", Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Apache_HTTP_Server

[18] Cloudflare, "HTTP/2 vs. HTTP/1.1: How do they affect web performance?". Retrieved from https://www.cloudflare.com/learning/performance/http2-vs-http1.1/

[19] Willy Tarreau, "Configuration Manual version 2.4.0", (2021/05/14). Retrieved from https://cbonte.github.io/haproxy-dconv/2.4/configuration.html#4-http-reuse

[20] Nick Ramirez, (August 2021), "The HAProxy APIs", Retrieved from https://www.haproxy.com/blog/haproxy-apis/

[21] wrk2 GitHub page. Retrieved from https://github.com/giltene/wrk2

[22] Nishanth G., (April 16, 2021), "Hashing in context of Load Balancing". Retrieved from https://medium.com/nerd-for-tech/hashing-in-context-of-load-balancing-392b317fe40e

[23] OWASP, "Session Timeout". Retrieved from https://owasp.org/www-community/Session_Timeout

[24] OWASP, OWASP Cheat Sheets, "Session Management Cheat Sheet". Retrieved from https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html#idle-timeout

[25] Moemen Mhedhbi, (November 28, 2017), "Dynamic Configuration with the HAProxy Runtime API". Retrieved from https://www.haproxy.com/blog/dynamic-configuration-haproxy-runtime-api/

[26] AWS Documentation, "AWS::CloudWatch::Dashboard". Retrieved from https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-resource-cloudwatch-dashboard.html

[27] AWS Documentation, "Dashboard Body Structure and Syntax". Retrieved from https://docs.aws.amazon.com/AmazonCloudWatch/latest/APIReference/CloudWatch-Dashboard-Body-Structure.html

[28] International Journal of Emerging Technologies and Innovative Research (www.jetir.org), (February, 2018) "Understanding DevOps & Bridging the Gap from Continuous Integration to Continuous Delivery", ISSN:2349-5162, Vol.5, Issue 2, page no.1420-1424.

[29] Silvia Barros, (May 21, 2018), "What is a DSP (Demand-Side Platform)? A Complex New World". Retrieved from https://www.mobidea.com/academy/demand-side-platforms/