Containerized Video Streaming for Edge Computing

FIPA - 2019/2020

Aleksandra Minko

Cédric Collard

Abstract

Video on demand and live streaming content have become the most important source of network traffic in mobile and fixed networks in recent years. In order to be able to deliver the huge amount of video content requested by the users, content providers employ cache servers and Content Delivery Networks (CDNs). On the other hand, in this years the paradigm of Edge Computing has emerged, which consists in deploying computational resources in the access network (base stations, central offices, access points). Our project explores the advantages of deploying CDNs up to the very edge of the network.

Since resources are limited in the edge nodes, they have to be judiciously allocated, if different CDNs want to run at the edge. The first step toward a “good allocation” is the understanding of how the performance of a single CDN are impacted by the resources granted, which is the goal of this project.

In this direction, we make the following contributions: (i) we deploy our own CDN architecture in the FTTH platform of Télécom SudParis, (ii) we pack our CDN in Docker images and release them under an open source licence, to make our work reusable, (iii) we measure the video quality provided by our HTTP video streaming service and how it changes when varying its storage and bandwidth.

Acknowledgement

We would like to thank our professor Mr. Andrea Araldo for his guidance and sincerely appreciate his patience, motivation and knowledge.

We would also like to thank our other jury members, Professor Franck Gilet, and professor Mounia Lourdiane for generously offering their time.

Introduction

Streaming videos on platforms such as Netflix and Youtube represents between 60% to 82% of the global Internet traffic[1]. Because client needs in bandwidth double every three years, it is essential to find a way to reduce the traffic generated by streaming services.

In order to be able to deliver the increasing video demand, it is essential to deploy caches and Content Delivery Networks (CDNs). This solution will reduce the amount of transit traffic in operator networks and improve the quality perceived by the clients.

In the network there exist one or more servers called “Origin” servers and data centers installed strategically close to the Edge network that offer a CDN service with high delivery and high content due to user proximity. The content is stored on the Cloud servers and sent out, when needed, to the CDN if the content isn’t already cached.

Having multiple content providers to deploy their physical cache data centers right into the Edge network comes at a high monetary and time cost, and is in most cases impossible (e.g., in a base station there would be no space to install many physical racks)

Another way to achieve CDN services is to use virtualization, and in particular containerization. A container allows to package up an application and all its dependencies so the application can be quickly deployed on any environment. The primary reason behind this solution is that containers offer many benefits such as consistency, portability, and better resource allocation. In our vision, each content provider will have containerized Edge nodes easy and fast to deploy.

The concept of the containerization is decades old, but the emergence of “Docker Engine” accelerated the use of this technology.

In this project, we will use a Docker based architecture with an Origin server designated as “Cloud server” and an Edge server designated as a “CDN server”. We will consider the impact of resource allocation to a virtualized CDN and we will investigate the efficiency of the solution in terms of storage capacity, the available bandwidth and content placement by testing different scenarios discussed later on.

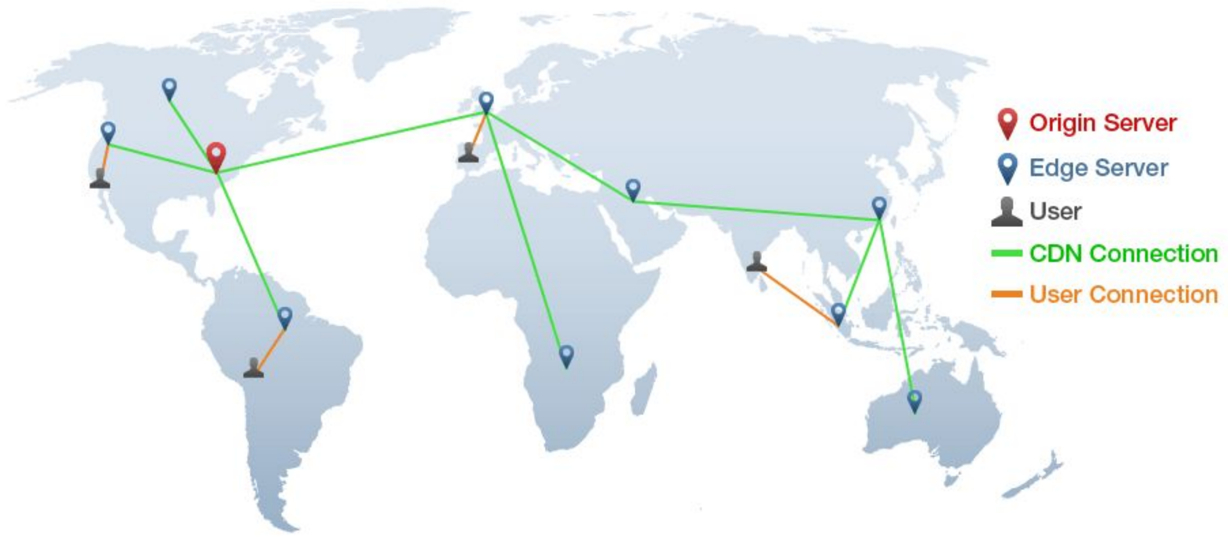
[1]https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html

Summary

I - Contex: Content Delivery Network

1 - CDN - Content Delivery Network

CDN in short for *Content Delivery Network*, is a geographically distributed network of servers around the world, hosting services like streaming, Video On Demand (VOD) or e-commerce. Those servers are usually replicas or "caches" of an origin server. The closer the user is to the server, the faster the delivery will be.



[2]

This saves bandwidth on the core network, as users will only fetch content from the closest server, and provides better quality of service to the users (faster site load times, reduced latency and, as we will show in our project, better multimedia quality).

For the content provider, it also provides a protection against surges in traffic, reduce the hardware (memory, CPU) requirements on the servers as the workload is divided between replica servers. It's also a way to protect against Denial Of Service (DOS) attacks, as it would require massive resources to spam every server instead of spamming only one.

2 - How does it work ?

An origin server has all the catalogue of contents of a provider. It sends content to be cached on the distributed servers. The content to be cached is usually the most viewed by users. If a user asks for a content not present on the distributed server, this server requires the origin server to provide it.

[2] https://imageboss.me/docs/content-delivery-network

For the user, this process is completely transparent; the user requests a web page, the CDN will redirect it to the nearest cache server who will deliver the content. The only way for the user to notice is for them to track the requested and delivered URLs.

3 - What are the stakes ?

Video traffic will take up to 82%[3] of global Internet bandwidth in 2020. As data speed need for an average user tends to double every three year, finding ways to manage it is a real challenge.

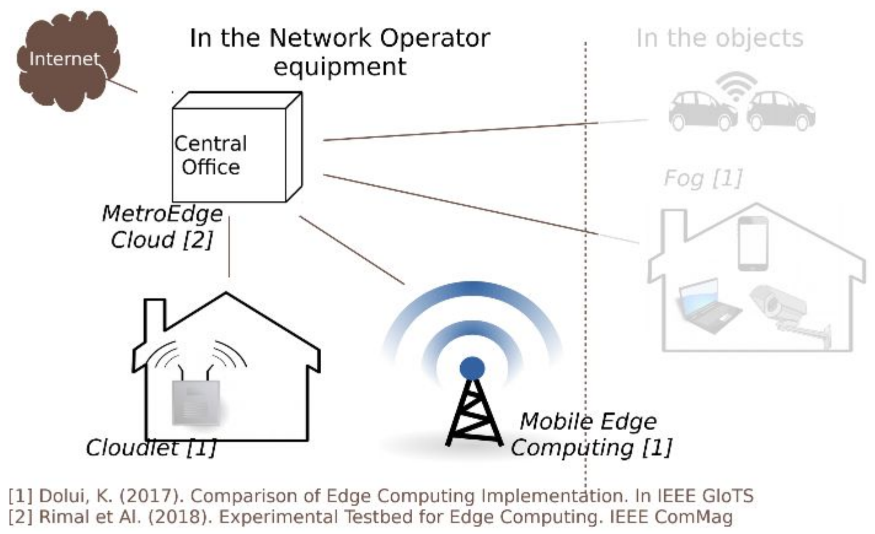
We said that CDN is a way to reduce bandwidth consumption on Internet core network, as close as possible to the users. But at the moment, most cache servers requires the installation of physical servers.

This means that every content provider or CDN has to deploy a physical server, with a full operating system and hardware, in various locations around the world. This method is slow and costly.

It seems more efficient to have one shared physical server between different content providers, as costs would be shared, and on this physical machine to have a virtualized server to enable this sharing. But where to geographically set up this shared infrastructure ?

4 - Edge computing

A new method of implementation is actually being studied to implement cache servers closer to the users, directly at an Access Point or Central Office of an access network :



[1] Dolui, K. (2017). Comparison of Edge Computing Implementation. In IEEE GIoTS
[2] Rimal et Al. (2018). Experimental Testbed for Edge Computing. IEEE ComMag

[4]

[3]https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html
[4] Andrea Araldo - Telecom SudParis

This proximity scales up a traditional CDN distribution from only a few locations around the world to almost everywhere. This approach reduces the upstream traffic on the Internet, but would also allow the deployment of CDNs on operator's network, sharing the infrastructure costs between multiple Content Providers and CDNs.

But this would completely change the deployment methods, as there would be millions of Edge caches to set up; the Network Operator would manage the physical server and Content Providers would virtualize their CDN servers on it.

A possible solution to virtualize the CDN servers would be through containerization of CDN caches. But how to manage resources allocation between the various CDN's containers ?

5 - The resource allocation problem

The efficiency of content delivery depends of the allocated bandwidth, the storage capacity and the content placement. Therefore, the storage and the bandwidth allocated influence the global cost of the solution.

Storage capacity : by increasing the storage capacity of the data centers, content providers become able to serve more content to their subscribers from their edge caches. But the cost of their infrastructure depends on the amount of added storage.

Bandwidth allocation : by optimizing the allocation of bandwidth, content providers improve the quality of experience (QoE) of the subscribers. However, the challenge is to scale the allocated bandwidth to cope with the variability of subscriber needs. Another important aspect is to find the balance between the bandwidth allocated between the users and the cache servers and the bandwidth allocated between the users and the Origin servers.

Content placement : By placing the most popular content items close to the users (subscribers), content providers become able to improve the global performance of their systems. The popularity of the items is determined by the collected statistics of user accesses.

Content providers decide whether to store the content item partially or with all possible qualities on the cache servers, e.g., 360p, 720p, 1080p. The decision is taken depending on the cache allocation policies implemented on the system.

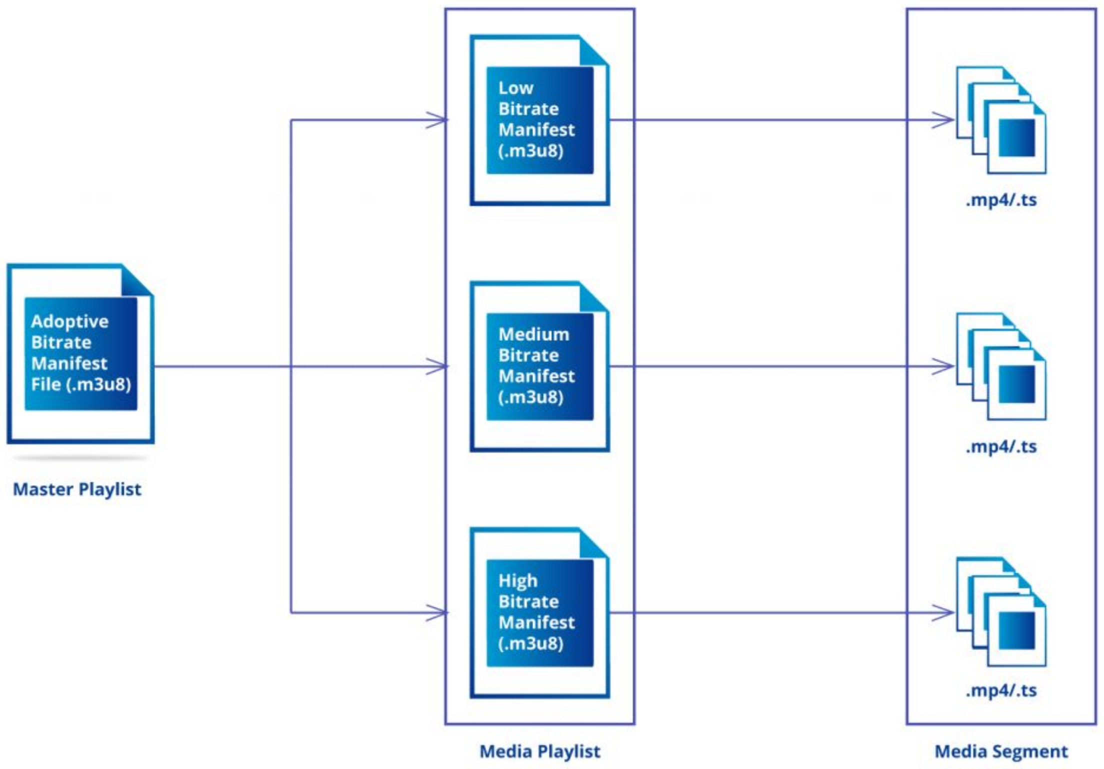
II - Context: Streaming protocol

In our project, we chose to use Apple's HTTP Live Streaming (HLS) protocol to deliver video to users.

1 - Http Live Streaming - HLS

HLS is a streaming protocol developed by Apple that uses HTTP and the most common streaming protocol in use today[5].

In HLS, a video is divided into segments (files) in the .ts format of few seconds each. These .ts files contain H.264 encoded video and AAC encoded audio. The segments are linked inside a m3u8 playlist. M3u8 is a file format from the m3u (MPEG version 3) family; a file.m3u8 is a simple text file with the different file locations constituting the playlist. This playlist file is also called **Manifest** file.


Adoptive Bitrate Manifest File (.m3u8)
Master Playlist
Low Bitrate Manifest (.m3u8)
Medium Bitrate Manifest (.m3u8)
High Bitrate Manifest (.m3u8)
Media Playlist
.mp4/.ts
.mp4/.ts
.mp4/.ts
Media Segment
[6]

HLS is adaptable to network performances, as it can link together multiple lists of segments of different resolutions (quality levels). The client will choose inside this playlist the right

[5] https://www.wowza.com/blog/hls-streaming-protocol
[6] https://www.synopi.com/hls-http-live-streaming/

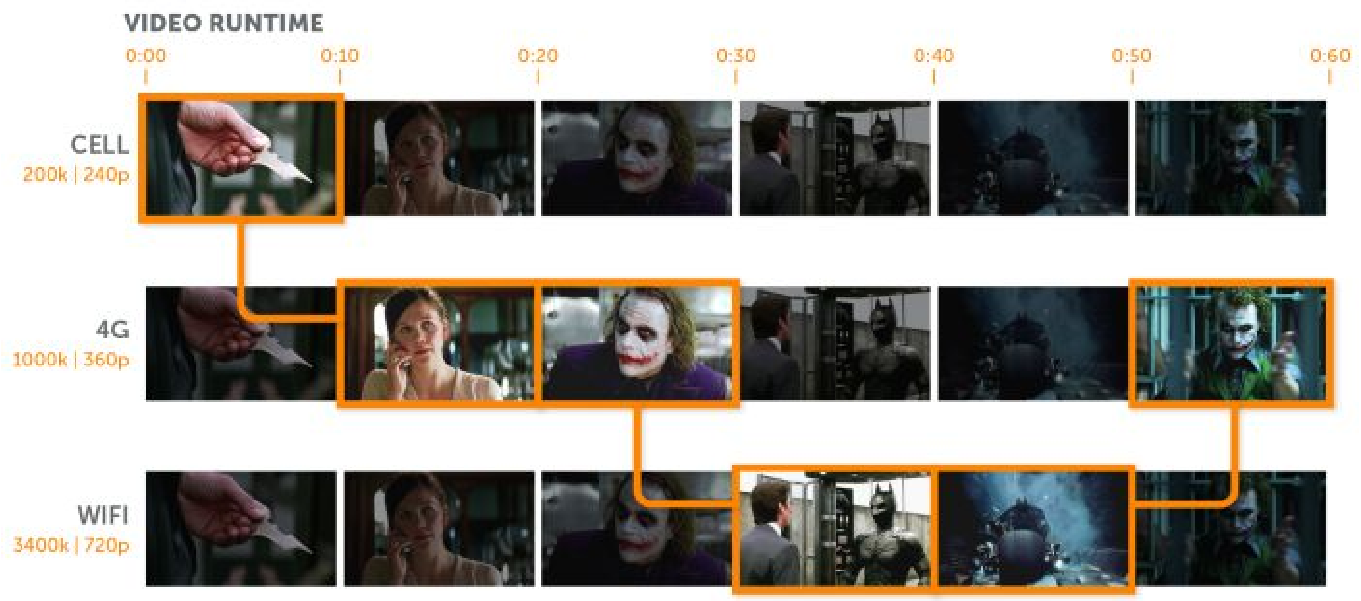
quality depending on its capabilities and network condition; it is a way to provide adaptive bitrate streaming. For instance, a smartphone with a small screen will probably not request video chunks at full HD quality. If the network connection is poor (low bandwidth or unstable connectivity) it is better to request chunks at low quality, since they require less bandwidth.

Users download an extended M3U playlist file which contains several URIs corresponding to media files. In the next sections, we will explain how we created a web server that can serve those files over the network.

Adaptive Bitrate Streaming With HLS

To deliver the highest quality stream possible, HLS streaming dynamically adapts the resolution of each video. This is called adaptive bitrate streaming,

Rather than creating one bitrate, a **transcoder** located in the server is used to create multiple streams at different bitrates and multiple resolutions. The transcoding to different resolutions (i.e., quality levels) occurs at ingestion time, i.e., when videos are added to the catalog. The clients then request the highest-resolution possible, depending on the user's screen and connection speed.



7

[7] https://www.wowza.com/blog/hls-streaming-protocol

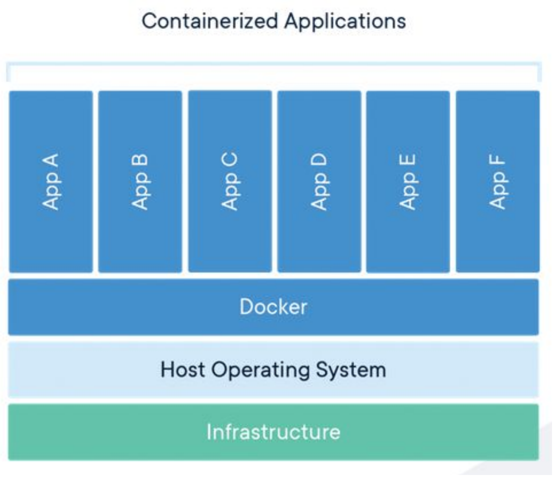
III - Context: Containerization

With the development of containerization, it is more efficient to virtualize the cache servers of content providers on one physical server deployed directly by an Internet Service Provider.

1 - What's a container ?

A container is an emergent way to virtualize a machine without having to manage an operating system (OS), as the machine host will share his kernel (and its resources) with all containers directly and dynamically, depending on their needs.

A container is a unit that packages up code and all its dependencies (libraries), so that it has everything it needs to run directly on any environment.



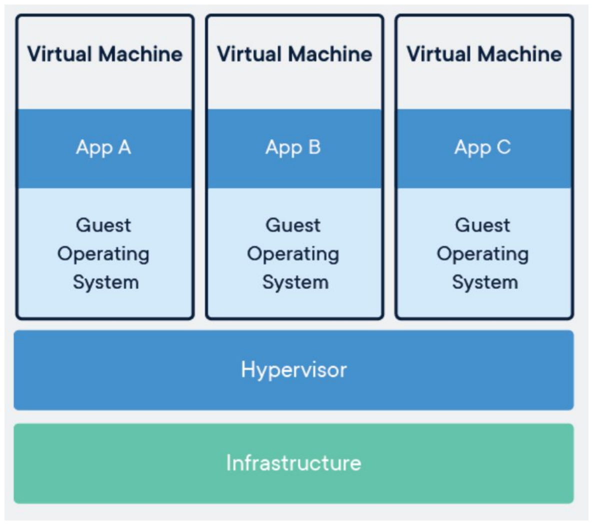
[8]

The most well-known containerization software is "**Docker Engine**". A docker container is very lightweight and it includes the code, the runtime libraries and all the needed settings. This allows for fast deployment, as no boot time is required for a container to start working and no hypervisor is needed. This is most effective for fast application deployment and for workload based scalability.

2 - Containers versus Virtual Machines

In traditional virtualization, several virtual machines run on top of one physical machine using a hypervisor which allocates computing resources to each virtual machine as needed. All virtual machines have an operating system (called guest operating system) and all necessary binaries which can make them sometimes slow to boot depending on available resources.

[8] Docker official website
https://www.docker.com/



By contrast, a container, doesn't require an operating system or a hypervisor. This leads to improved performance, as an application's instructions do not have to pass through the guest operating system and the hypervisor to reach the CPU.

Although virtual machines and containerization are two ways of abstraction from the hardware, they have different use-case.

We believe that in our project, containerization will be the better choice, since content providers would want to easily deploy numerous and light servers, without having to manage an environment or resources. Using containers, they would also be able to scale easily their apps and deploy on-the-fly their servers depending on the workload. For example, Netflix is completely containerized and uses millions of containers.[9]

3 - Docker Engine

Docker is a platform for developers that allows to develop and run applications. The engine creates a server-side daemon process that hosts the containers, networks and storage volumes.

Docker Images vs. Containers

A docker image is built from the instructions for a complete and executable version of an application. When the Docker user runs an image, it becomes one or multiple instances of that container. We can have many running containers of the same image.

[9] Netflix Blog. https://netflixtechblog.com/titus-the-netflix-container-management-platform-is-now-open-source-f868c9fb5436

4 - Dockerfile

A dockerfile is a text file that simply contains the build instructions to build an application image. The automatic build will insure the latest version of all the application dependencies. Although premade images are available for use from the Docker Hub, it is sometimes better to make a specific Dockerfile as we did in our project.

Here is the end-to-end workflow showing the various commands and their associated inputs and outputs :

```
+------------+  docker build   +--------------+  docker run -dt   +-----------+  docker exec -it   +------+
| Dockerfile | --------------> |    Image     | ---------------> | Container | -----------------> | Bash |
+------------+                 +--------------+                   +-----------+                    +------+
```

To list the images, execute on the command line:

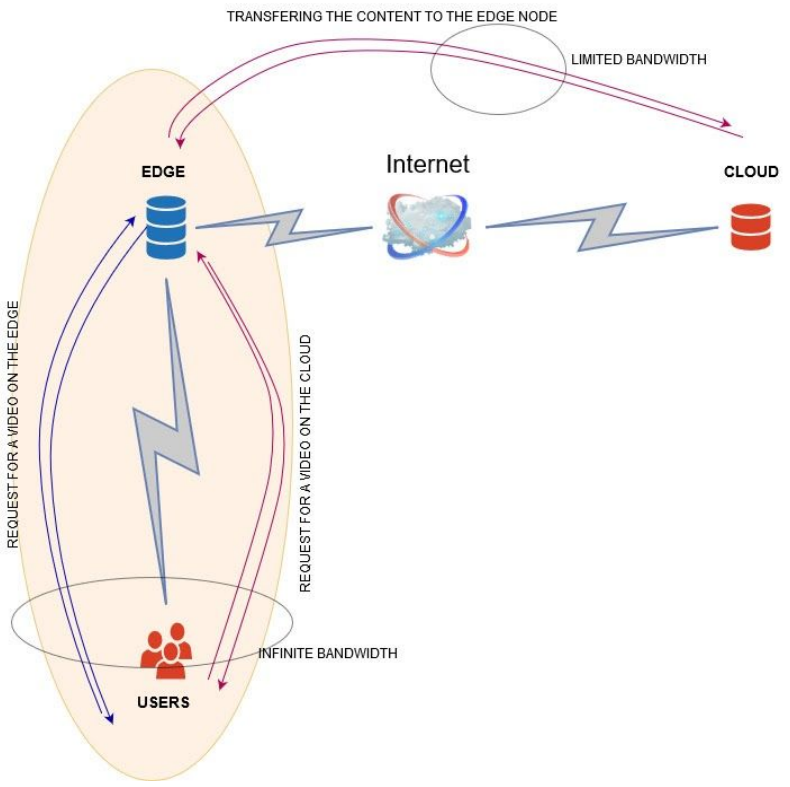
```
docker image ls
```

To list the containers, execute :

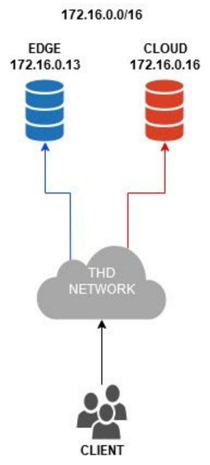
```
docker ps
```

IV - Experimental infrastructure deployment

In this project, we deployed the following containerized architecture, including an Origin server and an Edge server within the school's THD (FTTH) platform:


TRANSFERING THE CONTENT TO THE EDGE NODE
LIMITED BANDWIDTH
EDGE
Internet
CLOUD
REQUEST FOR A VIDEO ON THE EDGE
REQUEST FOR A VIDEO ON THE CLOUD
INFINITE BANDWIDTH
USERS

In a real life application of this architecture, there would be illimited bandwidth in the Access Network; upstream bandwidth between the Edge and Cloud, in contrast, would be limited, since shared by many users. To represent in our testbed this scenario, we used the following experimental infrastructure:



The servers are hosted on ESX (VMware hypervisor). The Origin server and the Edge server are both docker images representing one content provider server, such as Netflix; we will be focusing our study with only one content provider or CDN.

The Edge & the Cloud (or Origin) servers are both **Video-On-Demand** (VOD) servers. A VOD server provides a catalogue of videos for a user to choose from. We chose **Http Live Streaming** (HLS) protocol to serve the videos to the clients over the network. HLS uses **adaptive bitrate streaming** which is a technology designed to deliver videos to the user in the most efficient way and the highest quality possible depending on network performances and capabilities measured by the client during streaming.

In this project the servers are containerized using Docker Engine.

We made all our code available open-source, under Commons Creative CC-BY licence, and can be downloaded from our Github.

1 - Containerization

We used the most developed and well known containerization system, Docker. Docker allows us to access a library of already existing images on its Hub. The way Docker works is to make an image, which is a model of the container that we will want to deploy, already configured, from which you will run containers. That allowed us to deploy really fast applications, already functioning and ready to go.

In our project we used Video.js script for our client, which requests video chunks one after the other, and we created our own web page that allows the user to access and watch the content provider’s catalogue.

If you want to know more about Docker command line and Dockerfile, please refer to Docker's official website or our CheatSheet.

2 - The VOD Server

We used **Node js** to create our own web server that serves HLS files over the network. The server is asynchronous and avoids the allocation of resources for I/O tasks (Interactions with the system's disk).

“Asynchronous” means that the javascript functions in node js do not wait until a non javascript operation completes (like I/O). This approach is realized through the use of callbacks in the source code of the web server and in our case, it allows multiple requests.

Node.js is an open-source server-side programming platform that allows running javascript outside of a browser. NodeJS is also non-blocking and event-driven I/O system. This means, it does not wait for one function call to complete in order to call the next one. This is why we chose to use it.

Our node js server uses “in-memory streams” (The streams available on the server’s disk) and can only send HLS media files that have been properly transcoded and saved in the right catalogue directory in the server’s root. The media transcoding is explained in the next section.

To run the server inside the container we need to use the command : `node cdn.js` in the dockerfile to start the server instantly. The server listens on port 8000.

The static HLS files can be served to the users by accessing the following link :

`http://Server_@IP:8000/name_of_the_file.m3u8`

The complete source code is available on github. As the Origin server “Cloud” and the Edge server “CDN” work the same way, we use the same source code for those two servers.

3 - Static HLS transcoding

There are different ways of organizing content on the Edge node. The content provider could, for example, save the media contents under all possible resolutions on the Edge node, if the node **has enough storage memory**.

Or it could only save the highest resolution of the media content and then use the adaptive bitrate transcoding on the fly to send the correct resolution in response to a client capabilities, but it would then need the appropriate **CPU resources**.

We will consider in our studies the first method of implementation, also known as static (or offline) transcoding. Since end users may have different screen sizes and different network performances, we will create multiple renditions of our catalogue videos with different resolutions. This is called Multi Bit rate.

For each resolution, segments are created and linked in a m3u8 playlist (Manifest file). The playlist is the first thing a video client downloads when requesting a video.

We used an open-sourced bash script, based on **ffmpeg** (a transcoding tool), which will create a m3u8 playlist with 4 renditions of 4 resolutions:

- 1080p (1920x1080) (Original)
- 720p (1280x720)
- 480p (842x480)
- 360p (640x360)

Here’s the result of running this script applied on example.mp4 video :

```
bash create-vod-hls.sh example.mp4
      |- playlist.m3u8
      |- 360p.m3u8
      |- 360p_001.ts
      |- 360p_002.ts
      |- 480p.m3u8
      |- 480p_001.ts
      |- 480p_002.ts
      |- 720p.m3u8
      |- 720p_001.ts
      |- 720p_002.ts
      |- 1080p.m3u8
      |- 1080p_001.ts
      |- 1080p_002.ts
```

The player on the client’s side, which uses **video js framework,** will select in this playlist the adequate resolution, depending on the bandwidth (we’re still in static transcoding and not on transcoding on the fly in this case). The source code for the used script are in the resources.

In our example, we took a single video to transcode and then saved it on our servers. This video weighs about 36 Mo. Although we considered a catalog of multiple videos (up to 10000), they will all correspond to this video.

4 - Web page with video player

On our servers, we added an html page with a player based on **Video js** framework and we added to it multiple functionalities such as playing random videos according to Zipf distribution algorithm, explained in the next section. Note that the code of this web page resides on the server, but it will be loaded by the client and executed by the client’s browser.

Video js is a an open source web player built with JavaScript and CSS libraries allowing to easily set up an HTML5 video player on websites.

The player is then available on http://<address ip of the server> on the client browser and will it start streaming **automatically**, **alternating** requests between the Origin server and the Edge server:

5 - Automating requests

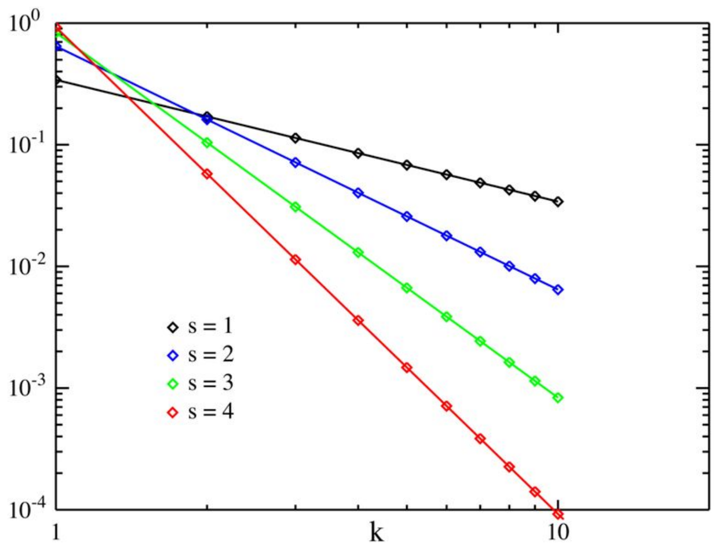
We needed a way to automate request generation to study the overall quality distribution of our video received by clients. The model widely used for content popularity in the Internet is the Zipf distribution (or ZIpf law). Ordering the videos from the most popular (video 1) to the least popular, this law says that, if we take any request, the probability the i-th video is demanded is:

$$P(i) = H \times \frac{1}{i^{\alpha}}$$

with **H** a constant, **i** the video asked for, **α** (sometimes denoted by **s** depending on sources) an indice between 0,8 and 1,2 which influence the steepness of the curve. H is calculated such that the sum of all P(i) is 1, so that we have a well-defined probability distribution.

This law models well the frequency of words in a text, but also approximates well the probability of a video to be asked for by a client.[10] If we take a look at the Zipf distribution of words (probability depending on the frequency of use), it looks like this:

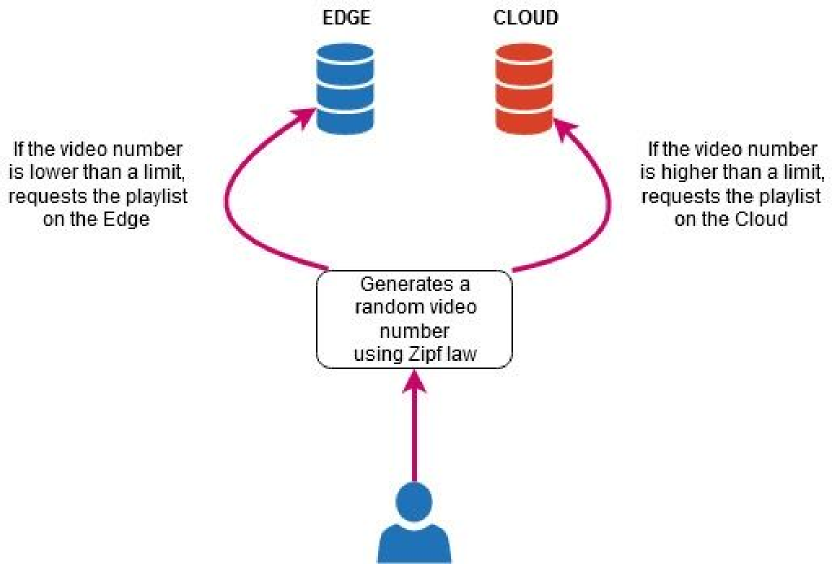
[10] Breslau, L., Cao, P., Fan, L., Phillips, G., & Shenker, S. (1999). Web Caching and Zipf-like Distributions : Evidence and Implications. In IEEE INFOCOM.



[11]

In our case, instead of the frequency of use of a word, we are interested in the viewing frequency of a video. So the most viewed video will have a high probability of **H**, the next will have a probability of $P(2) = \frac{H}{2}$ (if **α**=1) and so on.

We will use the Zipf law in our scripts to automate requests between Edge and Cloud server. We assume that the number of **l** most popular videos are in the Edge. Then, we continuously generate a random number **i** from the Zipf distribution. If **i**<=**l**, it means the request for video **i** can be set to the Edge. Otherwise, it will be sent to the Cloud.



EDGE
CLOUD
If the video number is lower than a limit, requests the playlist on the Edge
If the video number is higher than a limit, requests the playlist on the Cloud
Generates a random video number using Zipf law

We used this javascript prob.js source code for the Zipf law random generator.

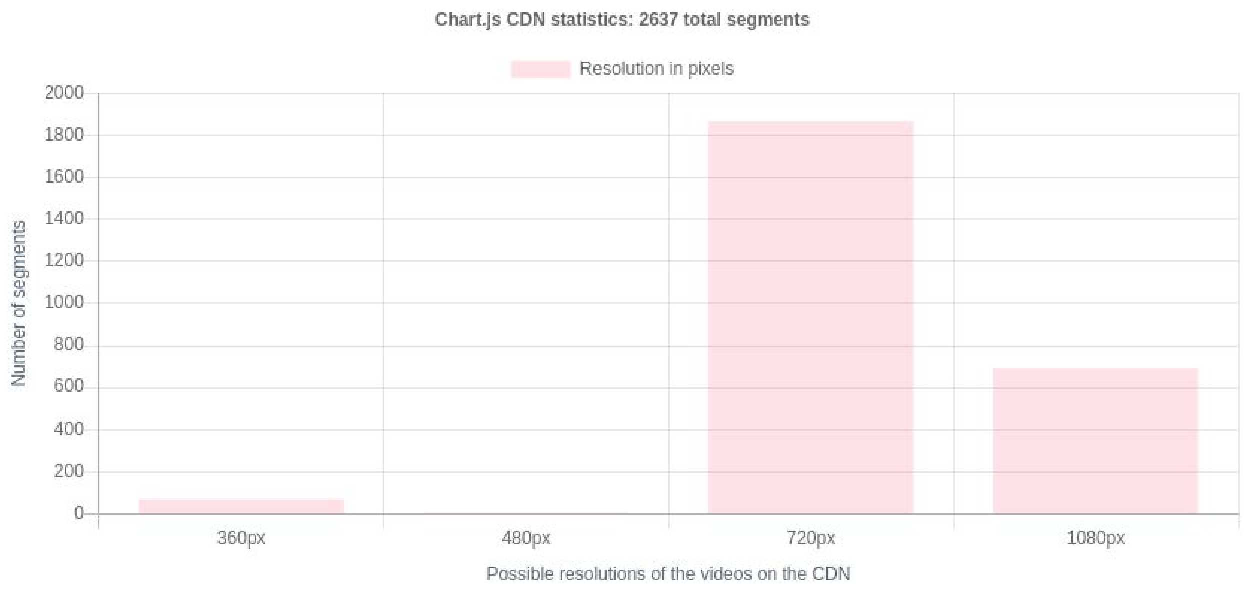
[11] https://fr.wikipedia.org/wiki/Loi_de_Zipf

6 - Statistics

We captured the number and the quality of segments sent by the Edge and Cloud server. We then parsed each segment and add it to a counter for each quality.

To display statistics on both the Edge & the Origin server, we added functions in source code based on chart.js module.

The generated stats are available on "http://<Edge_ip or Cloud_ip>:8000/stats" and look like this :



To generate the stats of what is received by the client, we simply merged the stats files of the two servers, since the client receives video segments either from the Edge or from the Cloud

7 - Final server architecture

We have two virtual machines which are very similar except the virtual volume of their catalog (since the catalog of all n videos is assumed to be stored in the Cloud, while only I<=n videos are assumed to be in the Edge). The operating system installed on both VMs is Ubuntu (18.04), with a 4 Gb RAM and a 20 Gb storage disk. In each of these VMs, we have Docker Engine installed.

On the first VM, we have our Cloud server container running with the following ip address: 172.16.0.13 and listening on port 8000.

On the second VM, we have the Edge server container running with the following ip address: 172.16.0.16 also listening on port 8000.

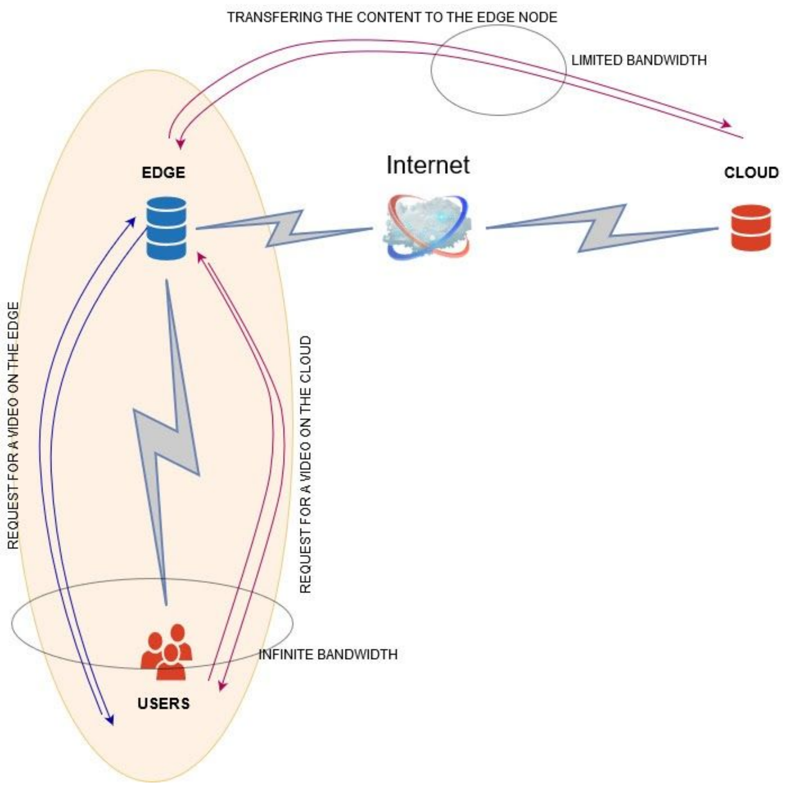
To see each detailed installation step, please refer to our GitHub.

V - Analysis

1 - Typical CDN topology

We will consider in our tests that the bandwidth in the access network is limitless, since in future 5G networks access technology are assumed to allow very large bandwidth and already in FTTH deployments the bandwidth on the access network is relatively very large. On the other hand, the bandwidth between the Edge and the Cloud node will be considered limited, because it is shared by unlimited users and because the network operator (i) generally pays transit operators to transmit/receive traffic to the rest of the Internet or (ii) has peering agreements with other operators in which traffic from/to the Internet is exchanged for free, but cannot exceed certain limits, specified in the agreements.

A real CDN architecture would work as follows:

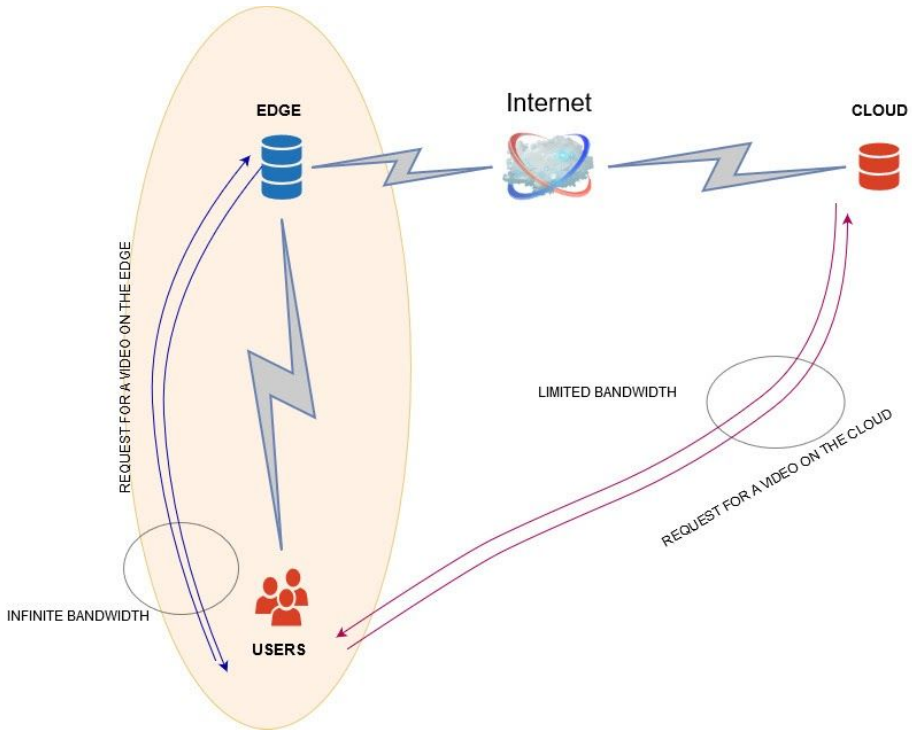

TRANSFERING THE CONTENT TO THE EDGE NODE
LIMITED BANDWIDTH
EDGE
Internet
CLOUD
REQUEST FOR A VIDEO ON THE EDGE
REQUEST FOR A VIDEO ON THE CLOUD
INFINITE BANDWIDTH
USERS

When a user makes a request, it is sent to the Edge server. If the requested content is present in the Edge catalogue, it will send it to the user. Otherwise, the Edge server will request the file from the Cloud server.

What's important here is that the users would never dialogue directly with the Cloud server. This is just a solution. Another solution would instead be that the user directly talks with the Cloud, by means of DNS redirection or similar techniques.

2 - Our CDN topology

Since the focus of our work is not on redirection mechanisms, we allowed the client to request the video to the Cloud server directly, if the video is not in the Edge (i>I). To emulate the limited bandwidth between the Cloud and Edge network, we used a Linux tool called **wondershaper**, which allowed us to limit the outgoing Cloud interface speed.


EDGE
Internet
CLOUD
REQUEST FOR A VIDEO ON THE EDGE
LIMITED BANDWIDTH
REQUEST FOR A VIDEO ON THE CLOUD
INFINITE BANDWIDTH
USERS

What is interesting to note is the variance of the quality resolution when we have a limited bandwidth between the client and the Cloud server. We can anticipate the fact that, if the Edge cache size I is sufficiently high, the majority of the requests will be directed to the Edge server, which is hosting the most viewed content.

Before analyzing data, we should mention that our VM hosting the Cloud has a 1 Gbps network interface; a real CDN would have much higher capabilities. Furthermore, we would have needed more clients on different machines to emulate a real traffic on a real access network. We instead launched 50 simultaneous browsers on a single client machine requesting videos. We also used a playlist of a single 36 Mo video; in a real implementation, we would have plenty of files, having different sizes. Therefore, our work can be considered a first step toward large scale experiments for Edge computing, which will be pursued by the THD team in their future activities.

For all those reasons, our results may not reflect a real world usage accurately but would provide general tendencies.

3 - Test results

For our experiments, we would change those parameters to see their significance:

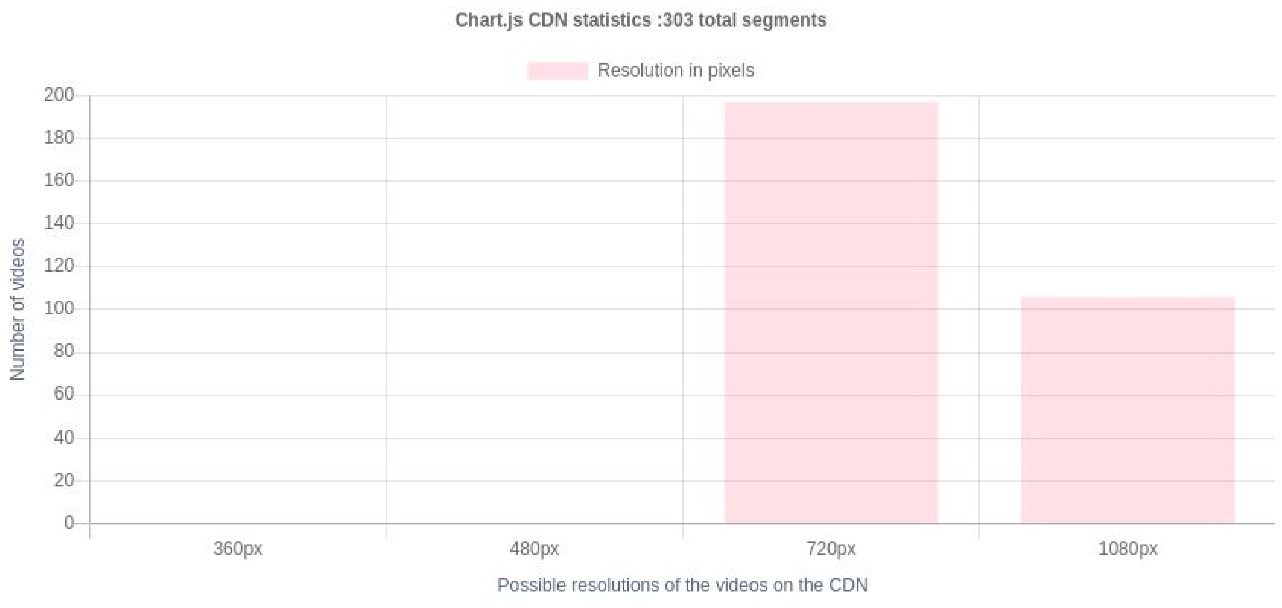
- **α** (or **s**), the exponent of the Zipf law,
- **n**, the total number of videos in the catalog (all present in the Cloud server),
- **l**, edge cache size, i.e., the number of videos on the Edge server,
- the outgoing interface speed for the Cloud server.

a - Influence of the Cloud bandwidth

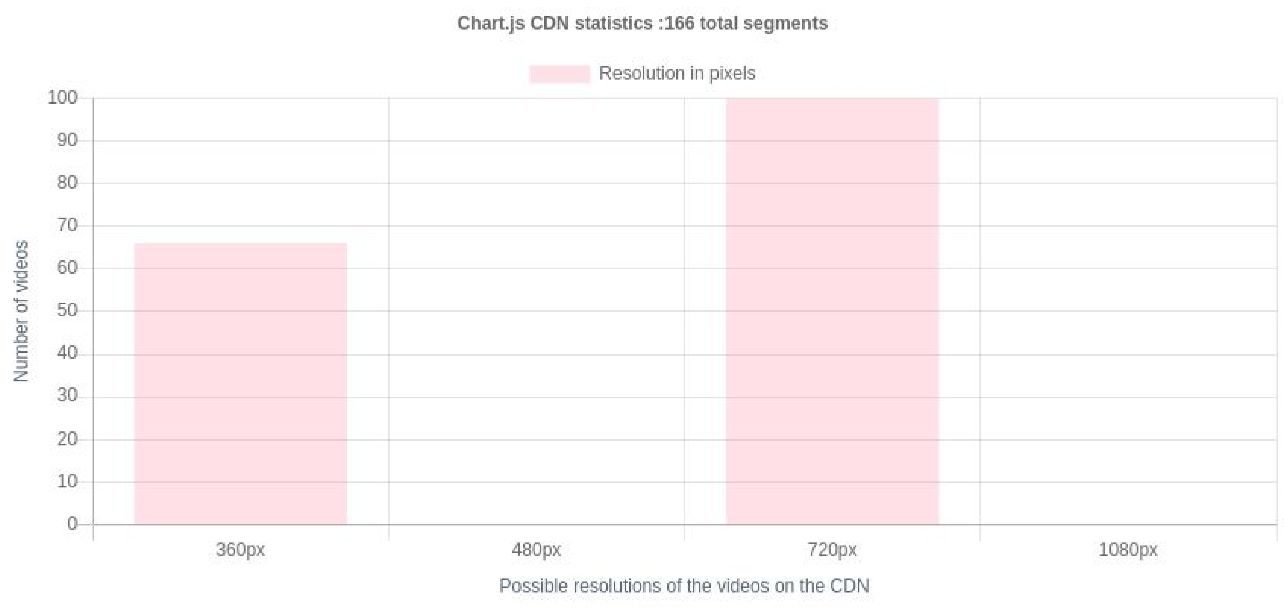
We started with **α**=1, **n** = 100 and **l** = 50. We first wanted to see the noticeable differences induced by modifying the Cloud bandwidth.

By reducing the bandwidth of the Cloud server, the segments quality sent by the Cloud server **deteriorated** as expected:

- with an interface speed of 1 Gbps, the segment quality sent by the Cloud varies between 720p and 1080p;

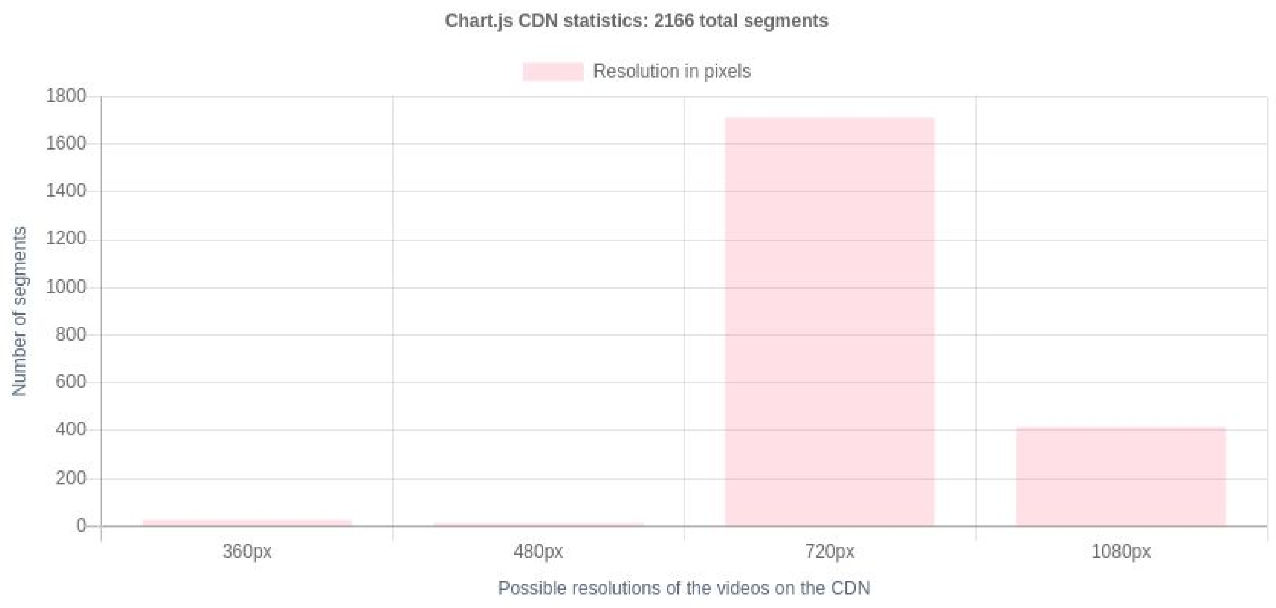


- with an interface speed of 500 Kbps, the segment quality varies between 360p and 720p:

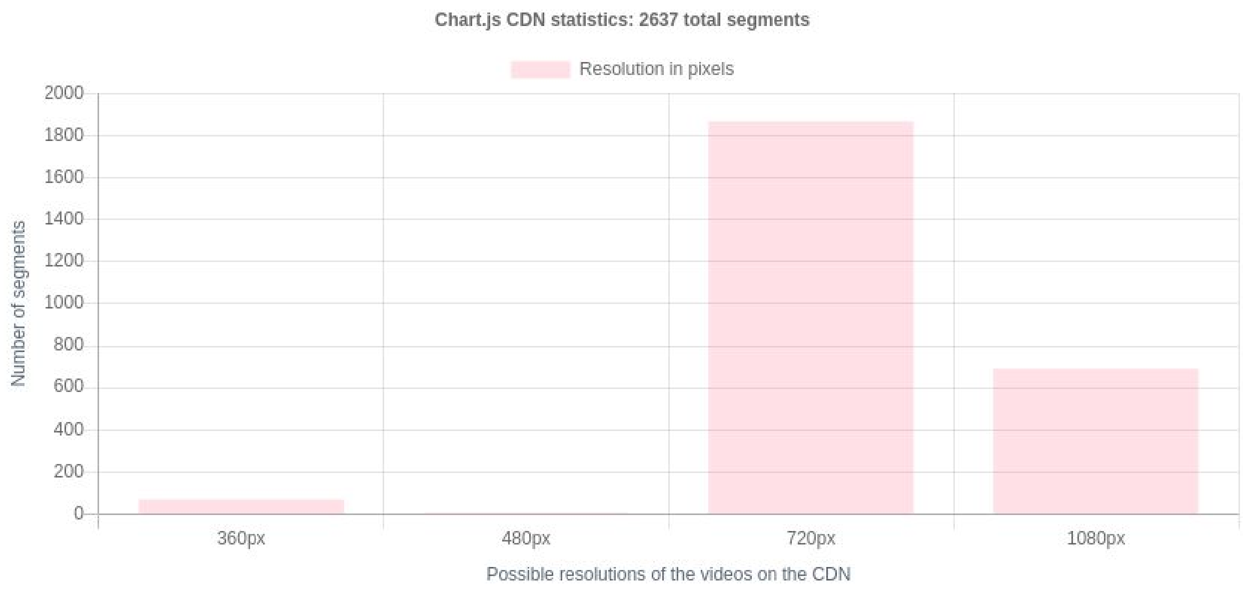


But the overall quality of the segments received by the client stays rather unchanged, because only approximately 225 segments are sent by the Cloud against 1800 segments for the Edge:

- quality received by the client with 1 Gbps for the Cloud speed:



- quality received by the client with 500 Kbps for the Cloud speed:

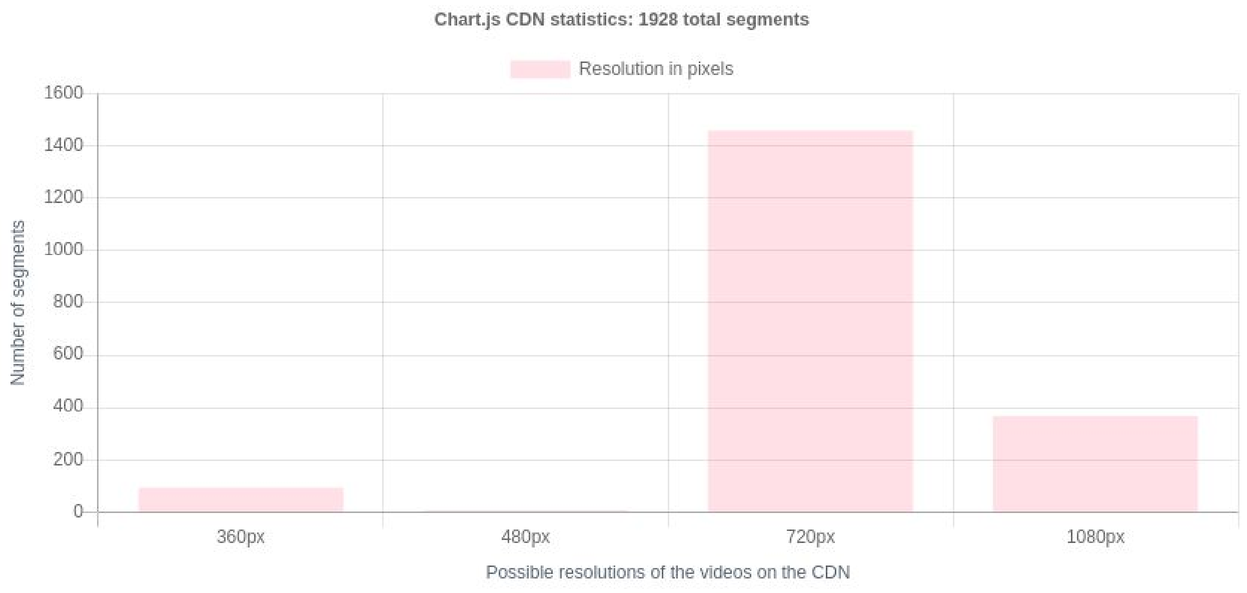


There is still the double of 360 segments with the 500 Kbps case, but the average segment quality still is of 720p. What was not expected was to find very few 480p segments compared to the 360p segments.

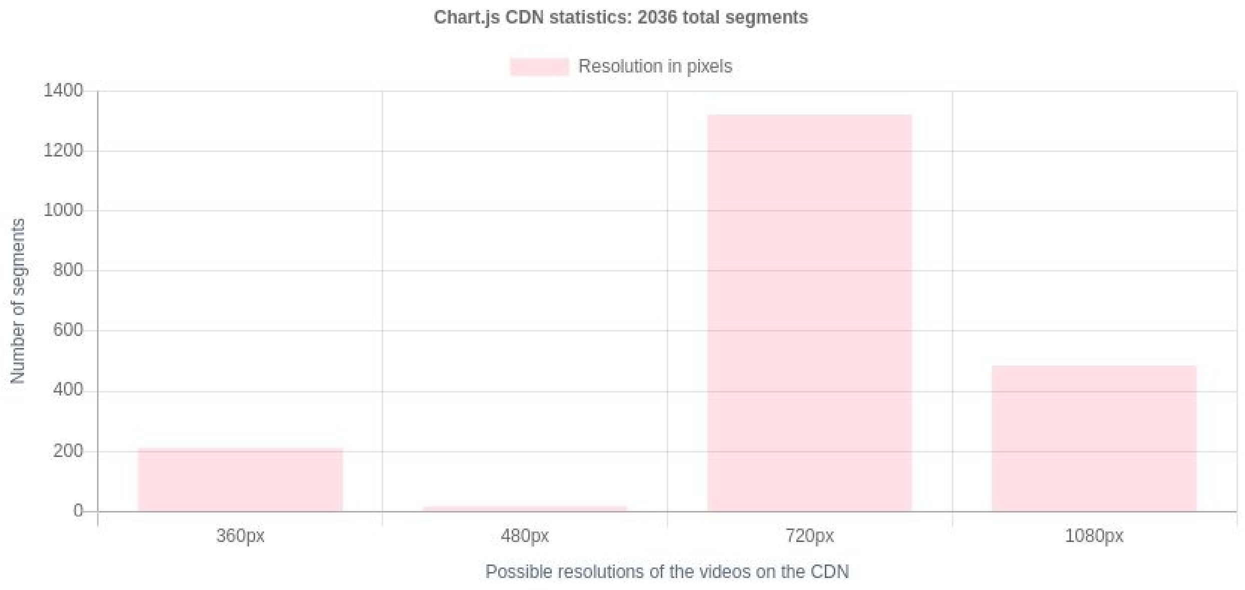
b - Influence of the Zipf curve steepness

We want to study next the effect of increasing the steepness of the Zipf curve by using **α**= 1.2. In theory, this should mean that more segments are sent by the Edge. With **n** = 100, **l** = 50 and an interface speed of 10 Mbps on the Cloud server, we received those results:

- **α**= 1.0, the client segment quality distribution is as follows;



- α= 1.2;



There’s an increase of 1080p segments, more than 25%, since the Edge node delivers more requests, around 200 segments more than when α=1.0 and the Cloud Edge delivers 150 segments less than when α=1.0. The increase of 360p segments, which doubled too, is due to the Edge having to handle most requests.

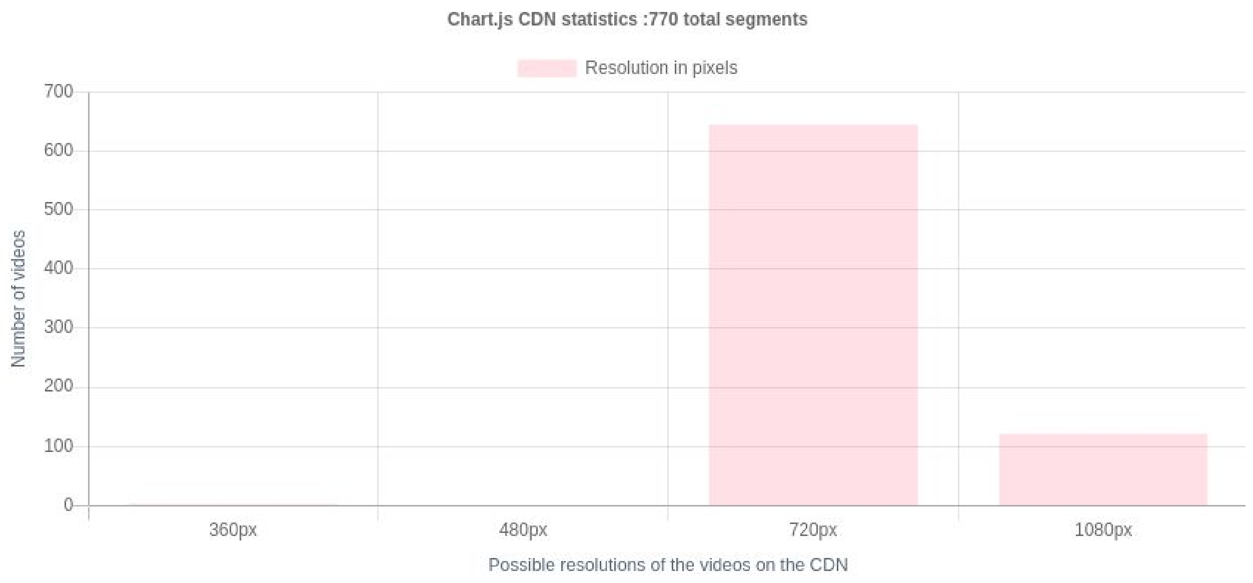
This experiment follows the theory as expected and shows that the advantages of Edge CDNs are more evident with steep catalogs, since more high quality chunks can be provided.

c - Influence of the cache to catalogue ratio

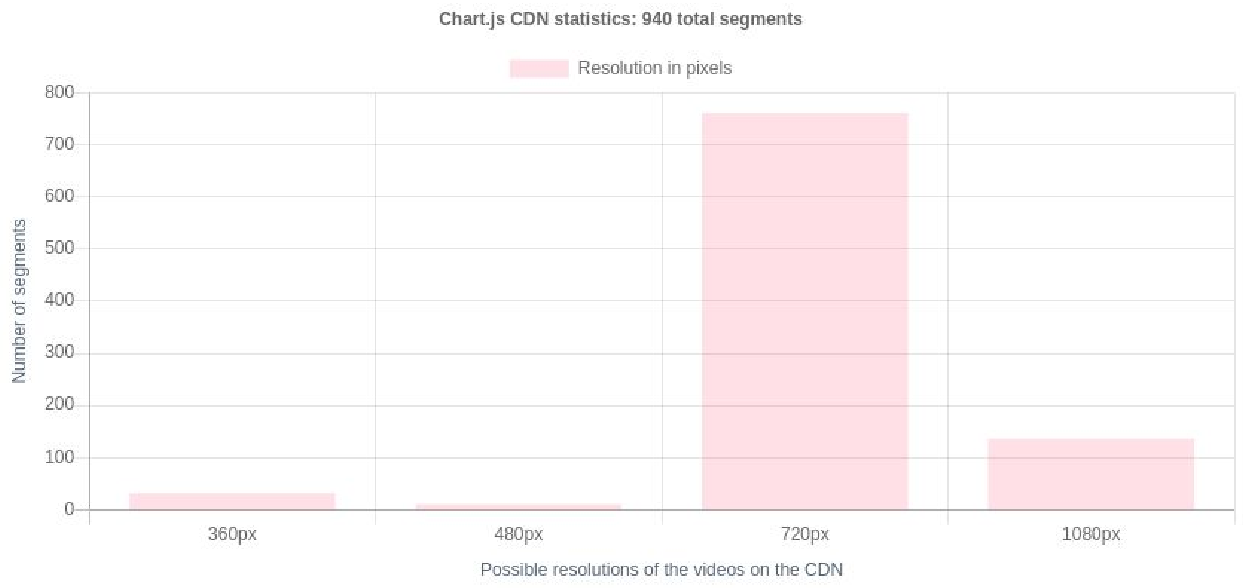
On a real CDN infrastructure, the ratio between the number of videos cached on the Edge and that on the Cloud server would be different than just 50 out of 100.

We considered taking a $\frac{10}{10\,000}$ ratio, that is to say, **l**=10 videos on the Edge and **n**=10 000 videos on the Cloud. The Cloud bandwidth is set to 100 Mbps and α=1.2.

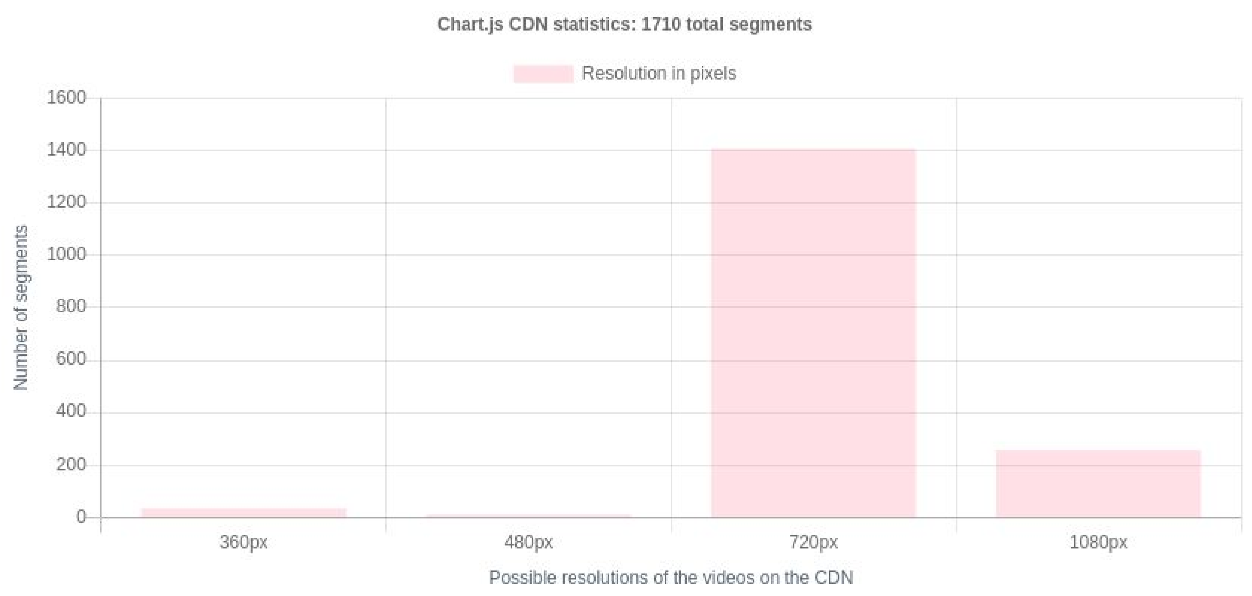
- for the Cloud server:



- for the Edge server:

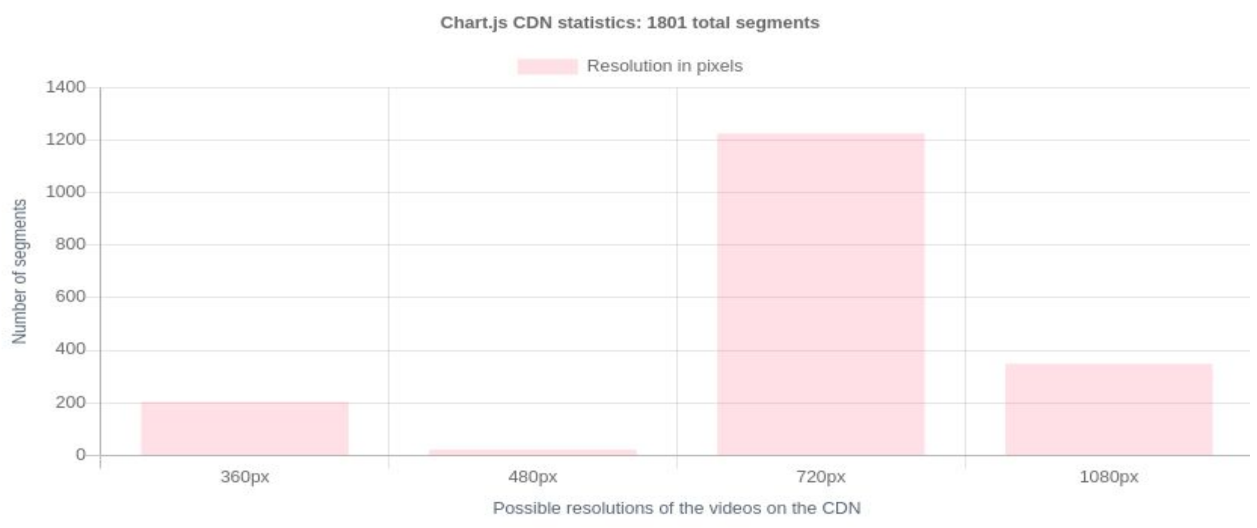


- for the client:



As in this experiment the Edge and Cloud servers have the same bandwidth of 1 Gbps, they have **similar distribution**. What's really interesting here is the **homogeneous repartition of the requests** between the two servers, around 800 segments for each. In this case, the bandwidth allocated to the Cloud server becomes very critical because of the volumes delivered.

If we increase the cache to catalogue ratio by increasing the number of cached videos to a $\frac{1000}{10000}$ ratio, there's a 50% growth of 1080p segments received by the Client:



We can also notice a great increase of 360p segments that might be due to the Edge server handling more requests.

Conclusion

We achieved a functional basic CDN architecture, although it would need some tweaks to be closer to a real CDN architecture, with dockerized Cloud and Edge servers, which deliver VOD service through HLS.

We also studies the quality distribution of those segments to the user and verified that the quality distribution was similar to the results expected from theory.

What did we learn ?

We have acquired skills in different areas :

- What is a container and what differentiates a container from a virtual machine (*general knowledge*) ?,
- How to use Docker, how to run a container and to customize our own docker image (*technical knowledge*) ?,
- New trends in networks, as Edge Computing (*general knowledge*)
- The problems and models involved in distributing multimedia content on the Internet at high scale (CDNs, caching, Zipf model)
- Asynchronous programming with Node.js (*technological knowledge*).
- Video js framework and it functions (*technological knowledge*).
- Scripting for automating experiments in a real testbed.
- Releasing open source code.

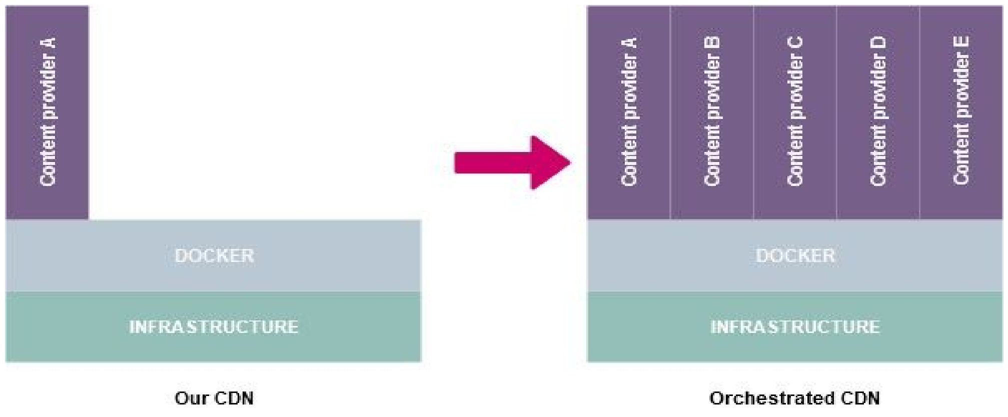
These proficiencies will most likely be applied in our soon-to-be career, as containerization and development will eventually be a crucial part of IT and Networks.

What's next ?

This project could be improved on several aspects:

- Although we managed a workaround, all requests might be directed to the Edge server, which would then ask the Cloud server if it doesn't have the content cached.
- In a production context, you would want to mount a volume in a Dockerfile to access your own catalog of videos and, instead of having the requests always hitting the same playlist as we did, it should request the specified file on the correct server. This would also mean we could have more accurate results, since the files wouldn't have the same weight.

- We only worked with an unorchestrated architecture. The implementation of an orchestrator, like Kubernetes, should provide more detailed data on the resources management the ISP should do on the server running multiple VOD servers. An orchestrated CDN architecture would allow to test such scenario :



- Transcoding on-the-fly could be added as an additional function to our CDN.
- Multiple clients from multiple machines should be launched to better emulate real load.

This project is a first step to an Edge computing analysis and provide the necessary tools to later study resources allocation in a physical infrastructure shared between many Content Providers.

Resources

All resources used in our project are their author propriety:

- Docker's Official Website
- Node.js
- Video.js
- VOD tutorial
- Our GitHub
- Zipf law (Wikipedia)
- Prob.js
- create-vod-hls.sh
- Transcoding tutorial using ffmpeg
- CDN explained
- Kubernetes
- Millions of base stations in China
- Cisco: Internet traffic forecast
- Andrea Araldo’s work about Edge Computing (Telecom SudParis)
- Jellyfish video