E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

Characterization and identification of Netflix encrypted traffic

Grado en Ingeniería
en Tecnologías de Telecomunicación

Trabajo Fin de Grado

Pablo Patús Díaz

Dr. Eduardo Magaña Lizarrondo

Pamplona, 23/02/2022

upna
Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Index

1. Introduction 5
2. State of the art 5
3. Objectives 10
4. Streaming of Netflix content 11
4.1. Adaptive pseudo-streaming 11
4.2. MPEG-DASH 11
4.2.1. Periods 11
4.2.2. Adaptation Sets 13
4.2.3. Representations 14
4.2.4. Segments 15
4.2.5. Media Presentation Description file 15
4.2.6. ISO Base Media File Format 16
4.3. Netflix architecture 18
5. Identification of Netflix content 24
5.1. Identification of Netflix Segment sizes in PCAP files 24
5.1.1. Identification of OCAs 25
5.1.2. Identification of TCP connections with OCAs 25
5.1.3. Identification of Segment requests in each connection 26
5.1.4. Segment size calculation 28
5.2. Indexing of Netflix catalog 33
5.2.1. Indexing of Segments within a video 33
5.2.1.1. Manifest request 34
5.2.1.2. SIDX box parsing 36
5.2.1.3. Storage 37
5.2.2. Indexing of video metadata 39
5.2.3. Crawler 42
5.3. Matching 43
5.3.1. Ballot algorithm 43
5.3.2. Deterministic algorithm 45
6. Results 48
6.1. Generation of the set of captures for testing 48
6.2. Ballot algorithm 49
6.3. Deterministic algorithm 53
6.4. Comparison of algorithms 54
6.5. Behavior towards in medias res captures 54
7. Conclusions 56
8. Future work 56
References 57
Appendix 58
A. Intuitive introduction to adaptive pseudo-streaming. 58
B. Digital media storage. Containers. 66

1. Introduction

Streaming entertainment makes for a great fraction of the global Internet traffic. This is true even more so nowadays – due to the pandemic, Comcast (a well-known Internet Service Provider in the USA) registered in 2020 a 32% increase over pre-pandemic levels in peak Internet traffic, with 71% of downstream traffic accounting for video streaming [18].

Netflix is one of the most popular video streaming platforms as of 2022. It should come as no surprise that a great deal of Internet traffic must belong to Netflix. In 2015, Netflix decided to implement TLS (HTTPS) to video streams [19], to adapt to the demands of security in the modern Internet. The purpose of this work is to demonstrate that, despite the encrypted delivery of video, Netflix content can be identified with ease, accuracy and very quickly, questioning the end user's privacy.

This has a very important repercusion because information is power and this leaves the door open to Internet Service Providers to inspect client connections at will, extracting the user's entertainment consumption habits in the process. This also may give rise to traffic identification for similar entertainment platforms such as YouTube, Amazon Prime, HBO or Spotify; which actually raises the question of how compatible are pseudo-streaming protocols with privacy, in the moment of highest demand on security and highest demand on streamed entertainment.

We have implemented an algorithm which just by looking at the TCP traffic with Netflix is able to tell what the user is watching. Our concept improves the existing state of the art by identifying the videos much faster, which is achieved by basing our algorithms in very different foundational concepts compared to previous works. Also, during the process, we have developed techniques to gain access to Netflix catalog database, to Netflix videos themselves and to Netflix internal APIs.

Not even HTTPS is able to protect your Netflix traffic. Any streaming service which makes use of pseudo-streaming protocols as they are conceived nowadays is condemned to privacy vulnerability. We hope that this work about pseudo-streaming and its complicated relationship with security is of your interest.

2. State of the art

This project builds upon two pioneer works in identification of Netflix encrypted content. The first paper [1] by A. Reed and B. Klimkowski implements the mechanics required to identify pseudo-streamed content, focusing on Netflix. It begins explaining why any VBR encoded content delivered via pseudo-streaming can be unequivocally identified by observing the pattern of sizes of downloaded video segments or video chunks. The VBR encoding for Netflix content results in each video having video chunks with sizes that differ from video to video. This set of sizes configures a fingerprint which allows the identification of Netflix content.

The identification process defined in [1] consists of three steps. The first step infers the throughput over a live, wireless interface by observing the received BlockACKs (control frames used by 802.11n WiFi standard which acknowledges multiple data frames at once). They do so by [1]:

1. Assuming that a BlockACK's starting sequence number correlates to the receipt of the data frame with the same sequence number.
2. Inserting missing data frames between the received BlockACKs.

3. Assuming that all data frames carry a full-size packet, which corresponds to 1460 bytes of application layer data.

Let us take Fig. 2.1. and Fig. 2.2. as an example of their approach. Fig. 2.1. represents the captured 802.11n control traffic between a client watching Netflix and the Wireless Access Point (WAP). In this capture, three different BlockACKs have been received. The timespan between the first and the last BlockACK arrivals is 5.1 ms, and all three BlockACKs altogether confirm frames ranging from Sequence Number (SN) 3290 to 3297. As Fig. 2.2. depicts, that is equivalent to receiving 8 full-size packets. Therefore, 11.68 kB have been captured in 5.1 ms.

Time	Source	Dest.	Info
05:42.5028	WAP	Client	RTS
05:42.5028	Client	WAP	CTS
05:42.5028	Client	WAP	BlockACK Start: 3290
05:42.5028	WAP	Client	RTS
05:42.5028	Client	WAP	CTS
05:42.5028	Client	WAP	BlockACK Start: 3295
05:42.5028	WAP	Client	RTS
05:42.5053	Client	WAP	CTS
05:42.5053	WAP	Client	RTS
05:42.5053	Client	WAP	CTS
05:42.5079	Client	WAP	BlockACK Start: 3297

Fig. 2.1. Sample snippet from a wireless capture of Netflix traffic. [1]

Time	SN	App. Layer Data	
05:42.5028	3290	1460	//BlockACK received
05:42.5028	3291	1460	//inferred
05:42.5028	3292	1460	//inferred
05:42.5028	3293	1460	//inferred
05:42.5028	3294	1460	//inferred
05:42.5028	3295	1460	//BlockACK received
05:42.5053	3296	1460	//inferred
05:42.5079	3297	1460	//BlockACK received

Fig. 2.2. Estimating throughput using the capture depicted in Fig. 2.1. [1]

The second step uses the captured information to isolate video chunks. Isolation is important to estimate the size of each of the downloaded chunks individually. This is performed taking into consideration that:

1. As of 2016, Netflix videos were divided in chunks which lasted 4 seconds.
2. As of 2016, Netflix used Microsoft's Silverlight player [16]. This player seeked to fill a 4-minute buffer as quickly as possible.
3. Once the buffer is filled, the player would request a new video chunk each time a segment in the buffer was consumed. This resulted in the arrival of a new segment every 4 seconds approximately.

Consequently, every 4 seconds a burst of BlockACK frames could be isolated, corresponding to the ACKs generated by the download of a new video segment. The chunk size could be calculated for each isolated burst as explained earlier. For example, the capture depicted in Fig. 2.3. isolates two video chunks. The download for the first segment required 390 data frames, which corresponds to a 569.4 kB segment. Analogously, the size of the second segment is 658.46 kB.

```
Time            SN    App. Layer Data
07:32.3821    5821          1460          //BlockACK received
...
...
...
07:32.3962    6210          1460          //BlockACK received

...   //no more BlockACK traffic takes place in 4 seconds

11:33.0059    6211          1460          //BlockACK received
...
...
...
11:33.0172    6661          1460          //BlockACK received

...   //no more BlockACK traffic takes place in 4 seconds
```

Fig. 2.3. Estimating video chunk sizes.

The final step predicts the played Netflix content based on the estimated sizes of the chunks captured. The segment size estimation carried out in step 2 does not return the real size of the downloaded chunk, but rather an approximated value. This loss of accuracy is due to:

1. the last 802.11n frame within a segment download not necessarily having a 1460-byte payload size.
2. not taking into consideration for the segment size calculation any possible headers that the application data may contain.

To be able to match a set of captured video chunks to a certain Netflix video considering these segment size deviations, it was decided to group adjacent video chunks into 30-segments-long sliding windows. It takes two minutes to capture the sufficient video chunks to fill such a window. For each sliding window, a six-dimensional key is calculated. The key for each window is obtained as follows [1]:

- **1st Dimension**: **Total Size**. This is the total amount of data contained in the window, measured in bytes.
- **2nd – 6th Dimensions**: **Data Allocations**. The two minute window is divided into five 24-second slices. Then, the percentage of the entire window's data contained in each slice is calculated (no units; these are numbers between 0.0 to 1.0).

They provide an example of key calculation for a 30-segments-long sliding window captured while watching *Legally Blonde*, reintroduced here as the Fig. 2.4.

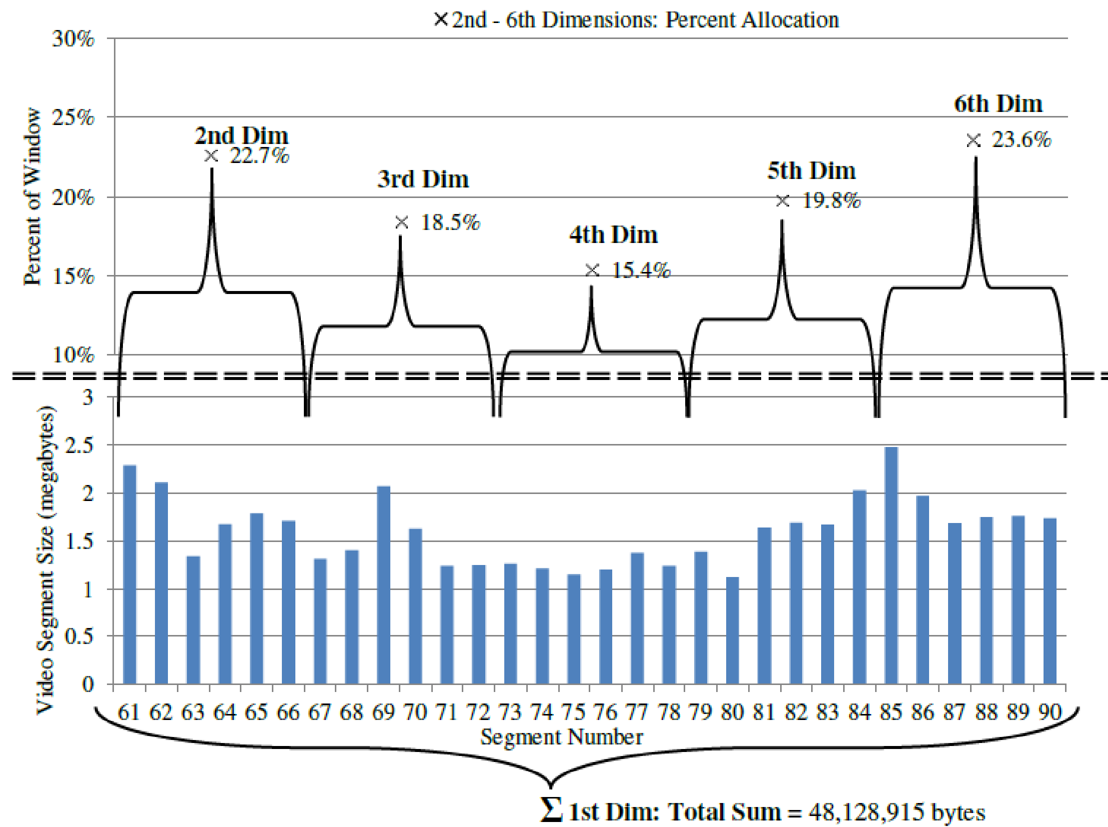


Fig. 2.4 . Key construction for segments 61 through 90 of *Legally Blonde* (3000 kbps encoding). Figure from [1].

This window's 6D key is: `[48128915, 0.227, 0.185, 0.154, 0.198, 0.236]`.

Every possible two-minute sliding window obtainable from a Netflix video was stored in a database and indexed with its 6D key. For the record, *Legally Blonde* was 1440 segments long back in the day (around 96 min), so each encoding resulted in 1411 sliding windows. As this film was encoded in 8 different bitrates, it yielded 11,288 individual windows in total.

In order to obtain a match:

1. They run the calculation of the 6D key of the captured two-minute sliding window.
2. They search in the database for the closest resembling window. To do so, they first obtain candidate windows by looking in the database for all windows with a key whose dimensions were within a predetermined set of thresholds relative to the capture window. Then, for each candidate window, the sizes of the segments of both the candidate and the capture window were plotted as an *x,y* scatter plot and its Pearson's correlation (*r*) was calculated. Any candidate window yielding *r* above a given threshold was considered a match. [1]

Their approach to identify Netflix content has several technical limitations:

- Their algorithm has a 90% accuracy. Furthermore, we must take into account that this hit rate was calculated with a very small dataset (50 movies and a season of a TV show).
- As the estimation of segment sizes is obtained via 802.11n BlockACK control frames, content identification renders impossible in Ethernet setups and in setups using other WiFi standards which do not define BlockACK control frames.

- As 30 segments (two minutes) are required to obtain a searchable key for the database, the identification process takes two minutes at least.

In 2017, the paper [2] was published by A. Reed and M. Kranch, building upon [1]. This time, their approach consisted in analyzing IP and TCP headers to infer segment sizes, rather than deducting them from BlockACK bursts. Thus, Ethernet captures were no longer a problem.

Once again, the sizes of video chunks were not 100% accurately calculated. Application layer payload size has an overhead due to HTTP and TLS headers. They decided to subtract such overhead by considering [2]:

1. HTTP headers add ~520 bytes to each video segment.
2. TLS overhead adds ~0.18% to the combined video content plus HTTP headers.

This works as an approximation, but fails to obtain the exact size of a video segment. Hence, to overcome this lack of accuracy, a technique involving six-dimensional keys for two-minute sliding windows had to be implemented again, as in [1]. Furthermore, the matching algorithm works very similar to its preceding work.

As this work is also based on the two-minute sliding window technique, two minutes are required at least to find a match. In fact, out of 200 video streams used for testing, 51% found a match before 2:26 min. It took 8:09 min to identify 90% of the videos and as long as 13:00 min to identify the whole set with a success ratio of 99.5%.

This paper improves the original work by:

- Enabling Ethernet captures and captures based on any WiFi scheme.
- Implementing a 6D key database for the whole American Netflix catalog (as of 2017). Only 50 movies and an entire season of a TV show were introduced into the dataset in the first work. Out of curiosity, 38,780 episodes of TV shows and 3,247 movies made up the dataset this time. This yielded a total amount of 184,248,110 sliding windows, inserted in the database with their corresponding keys.
- Improving hit rate (from 95% to 99.5%).

3. Objectives

Technology evolves quickly, and Netflix has evolved as well since 2017. Microsoft's Silverlight player has been replaced by Netflix's own player, video segments do not last strictly for 4 seconds anymore, their catalog keeps expanding, and new video codecs and resolutions have come to stay. The techniques described up to now for identification of Netflix content are now outdated.

In this thesis, we are going to demonstrate that the identification of encrypted Netflix videos is still possible nowadays, and we are going to improve the techniques that our preceding works deployed by:

- **Perfecting the accuracy of the calculation of segment sizes.**
  None of the methods proposed until now was able to calculate the exact size that a captured segment has. We have implemented a technique which calculates accurately the size of a downloaded video chunk. As we do not have to tackle the lack of precision anymore, we no longer have to use sliding windows paradigms to find matches.

- **Reducing encrypted video identification from minutes to just a few seconds.**
  As filling two-minutes sliding windows will no longer be required, the identification process speeds up immensely. Now, we will be able to capture a few segments within a few seconds to firmly determine the played Netflix content in an encrypted capture.

- **Improving the technique used to download the entirety of Netflix catalog.**
  A web scraper is used in [2] to determine all movies and TV shows that made up the catalog back then. We are going to use Netflix's internal API to retrieve all Netflix content without parsing any HTML.

- **Improving the technique used to retrieve the sizes of the segments of every video in the catalog.**
  As explained in 4.2., the first bytes of all Netflix videos contain the header which describes the sizes of the subsequent segments that make up a given video. Both [1] and [2] dumped the segment sizes by actively capturing the first seconds of a video stream to obtain the aforementioned header. In order to implement a database with all Netflix videos, as done in [2], a bot had to be programmed to watch every single video for a few seconds to extract the header. This was a very tedious and slow process: it took 4 entire days to retrieve all 42,027 video segments using 4 MacBook Pros simultaneously. We propose a much faster method to extract segment sizes which does not require active captures but makes use of another internat Netflix API.

4. Streaming of Netflix content

4.1. Adaptive pseudo-streaming

Pseudo-streaming is the transfer of digital media from server to client which enables the beginning of the content's playback prior to its complete download. At a technical level, this is done by splitting the content into video chunks or segments. As the user consumes chunks, more are requested to the server. Playback and download happen simultaneously.

In the scope of pseudo-streaming, the term "adaptive" means that the streamed content is able to adapt to the client's changing network conditions. For example, an adaptive pseudo-streamed content can lower playback resolution automatically if a sudden decrease in the user's bandwidth is detected, preventing the playback from interruption. Annex A offers a further intuitive introduction to adaptive pseudo-streaming.

For the correct delivery of streamed content, client and server have to be in constant communication to cope with tasks such as new chunk requests or sudden changes in the network conditions. A protocol is needed to enable this client-server communication. The most common adaptive pseudo-streaming protocols are [4]:

- Apple HLS (HTTP Live Streaming)
- Microsoft Smooth Streaming
- Adobe HDS (HTTP Dynamic Streaming)
- MPEG-DASH (MPEG's Dynamic Adaptive Streaming over HTTP)

The use of HTTP to request chunks is common to all previous protocols. Availing HTTP requests simplifies serving the chunks, as the requests can be processed by standard web servers.

Netflix pseudo-streams its content. In the days when Netflix implemented Microsoft Silverlight as the core player, Microsoft Smooth Streaming protocol was used, as it is the inherent streaming protocol to Silverlight. Nowadays, Netflix implements its own player with a proprietary streaming protocol which is heavily inspired by MPEG-DASH. Before explaining Netflix's architecture and how Netflix provides its content to the end user, MPEG-DASH protocol must be understood first, as it defines multiple concepts and terms later on applied in Netflix's protocol.

4.2. MPEG-DASH

MPEG-DASH [3] is MPEG's adaptive streaming protocol, based on HTTP. MPEG-DASH defines a hierarchy based on periods, adaptation sets, representations and segments. This hierarchy is used to organize the chunks that make up any content according to parameters as their resolution or language.

4.2.1. Periods

A **period** is a temporal subdivision of the content defined by a start timestamp and a duration.

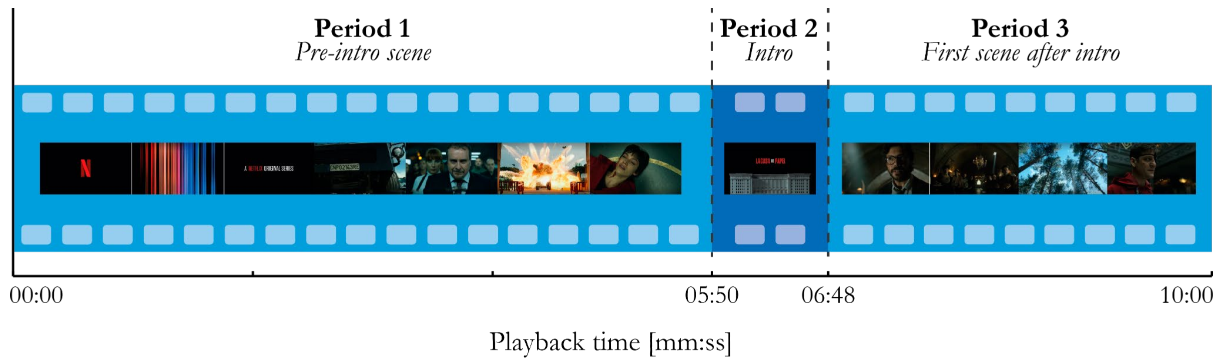


Fig. 4.1. Subdivision in periods for a *Money Heist* episode.

An example of a subdivision of a video into periods is depicted in Fig. 4.1 (only the first 3 periods are shown). The start timestamp and the duration that defines each period are described in Table 4.1.

	Period description	**Start timestamp** [mm:ss]	**Duration** [mm:ss]
1	Pre-intro scene	00:00	05:50
2	Intro	05:50	00:58
3	First scene after intro	06:48	03:12

Table 4.1. Period definition.

The protocol does not require the content to be divided into multiple periods. Thus, media can be classified as single-period or multi-period content:

- Multi-period content is made up of two or more periods. The content exemplified in Fig. 4.1 is clearly multi-period.
- Single-period content is made up of a unique period, as conveyed in Fig. 4.2 and in Table 4.2.

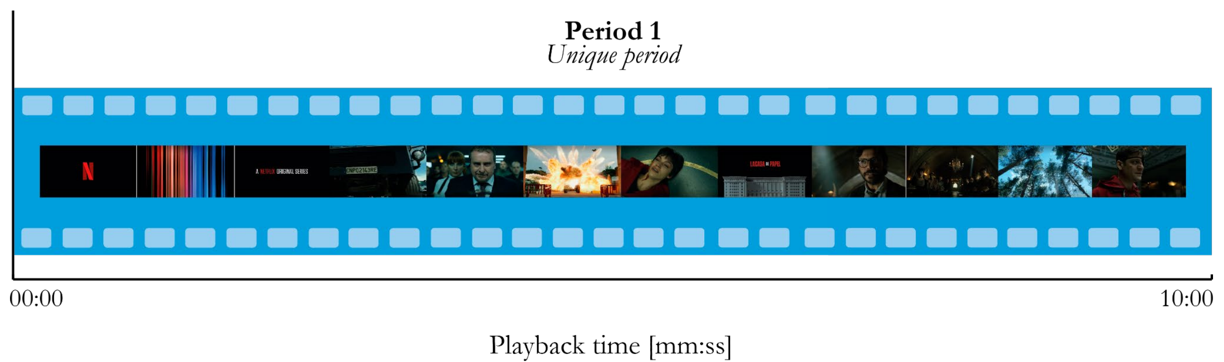


Fig. 4.2. Unique period for the same *Money Heist* episode.

	Period description	**Start timestamp** [mm:ss]	**Duration** [mm:ss]
1	Unique period	00:00	53:55 (entire episode)

Table 4.2. Single-period definition.

4.2.2. Adaptation Sets

Periods are made up of **Adaptation Sets**. Adaptation Sets are the media streams that enable the playback of a Period. For the playback of Netflix content, independent video and audio streams are required. Optionally, a subtitle stream could be introduced as well. In MPEG-DASH terms, any Netflix Period is made up of at least one video Adaptation Set, one audio Adaptation Set and, optionally, one subtitle Adaptation Set (Fig. 4.3).

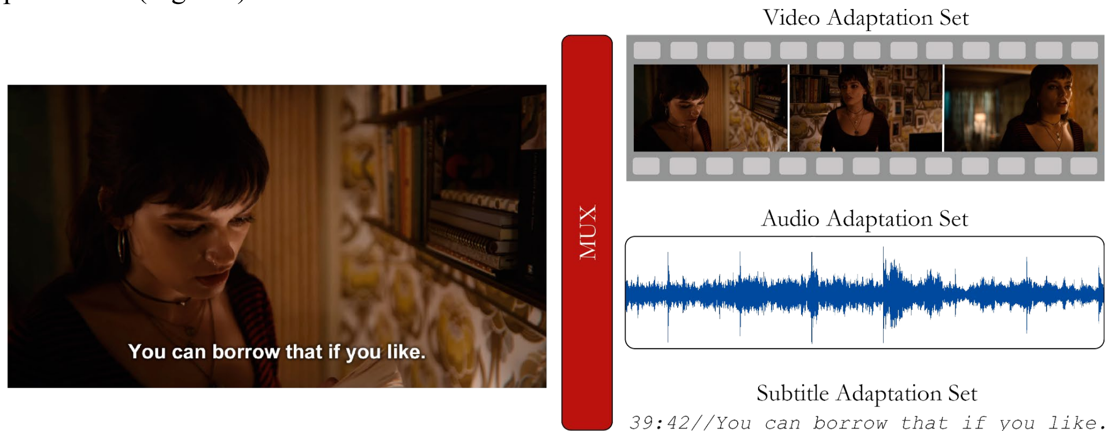


Fig. 4.3. Adaptation Set composition for a Period of *Sex Education*.

Periods can be composed by multiple Adaptation Sets of the same category.

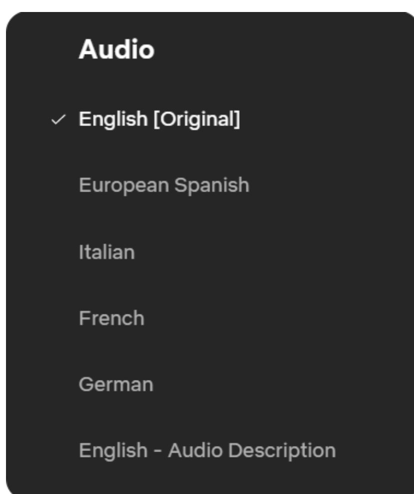


Fig. 4.4. Netflix audio options.

For example, if a certain Netflix video offers the languages shown in Fig. 4.4 for audio playback, such content is made up of 6 audio Adaptation Sets, one per audio option. Switching from one language to another only changes the audio Adaptation Set that is being downloaded; video and subtitle streams remain unaltered. The same applies to switching subtitles from one language to another.

Despite being able to have several audio and subtitle Adaptation Sets, Netflix content typically has a unique video Adaptation Set. This is because a single video stream is generally valid for all users, despite their language choice. However, MPEG-DASH can handle multiple video Adaptation Sets within the same Period. A practical use of including multiple video streams is depicted in Fig. 4.5. Each Adaptation Set offers different translations for the text within the video for the sake of language immersion.

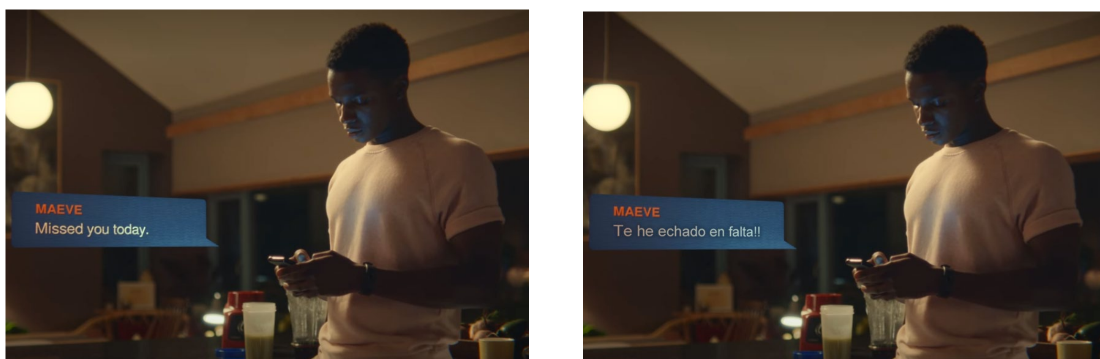


Fig. 4.5. Multiple video Adaptation Sets for language immersion.

4.2.3. Representations

An Adaptation Set is subdivided into different **Representations**. A Representation is a variant of a given media stream. The differences between variants are aspects such as playback resolution, target bitrate or the required codec. Fig. 4.6 offers an example of possible Representations that conform video, audio and subtitle Adaptation Sets.

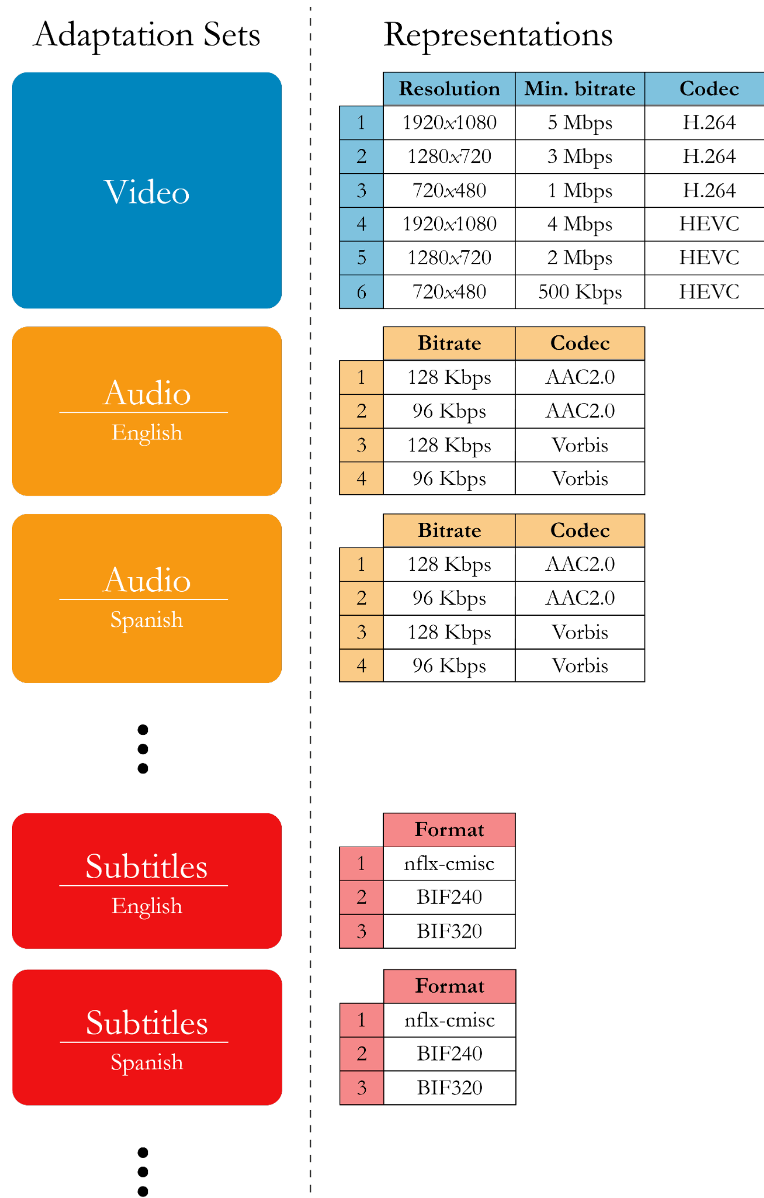


Fig. 4.6. Breakdown of Representations included within Adaptation Sets.

The criteria used to pick a Representation among all available within an Adaptation Set depend on the client-side implementation. The chosen Representations often depend entirely on aspects foreign to the end user, such as the codecs supported by the end host. In some implementations, the user has some control over which Representations to obtain. This is the case of YouTube, as the user can freely choose the playback quality, influencing directly on the election of Representations. In other implementations, such as in Netflix, the user cannot pick the resolution; the chosen quality relies entirely on the detected user's bandwidth.

4.2.4. Segments

A Representation is split into **Segments**. A Segment is a fragment or chunk of a particular media. Segments are downloaded by the client and played back-to-back giving the desired streaming feeling. They comprise the smallest unit in which any media can be divided and they are the elements on which adaptive pseudo-streaming is based. An example of a subdivision of Representations into Segments is represented in Fig. 4.7.

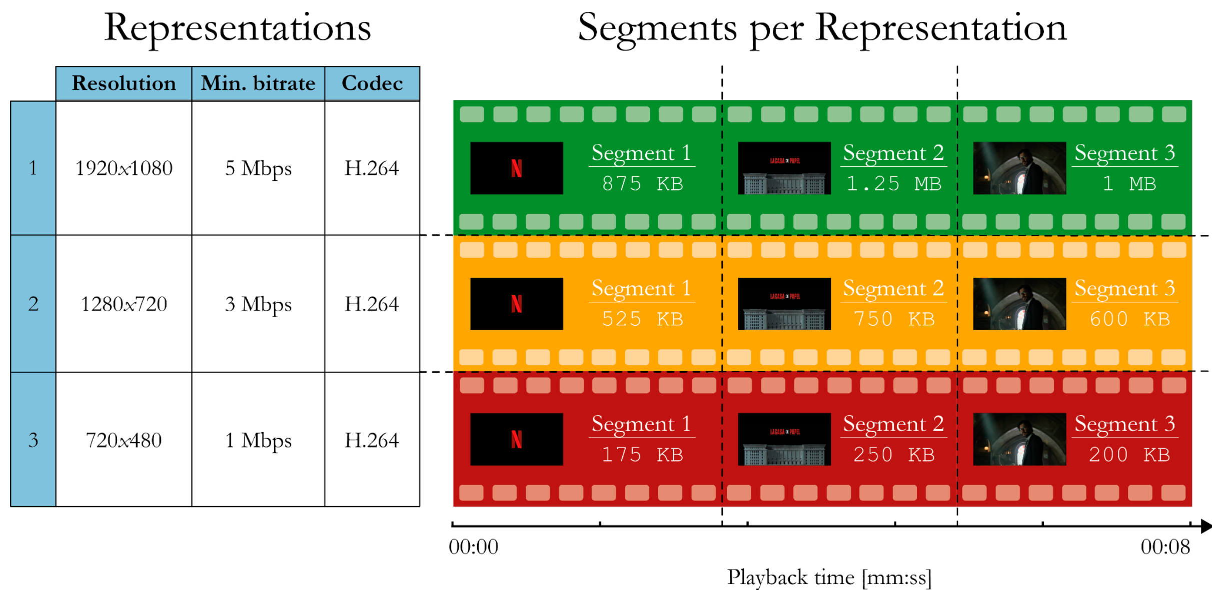


Fig. 4.7. Breakdown of Segments within video Representations, up to 8 seconds of playback.

As depicted in Fig. 4.7, the *n*th Segment in each Representation must last the same timelapse. In this way, the client can switch to another Representation (i.e.: change playback quality) without altering the chronological continuity of the playback.

Netflix uses the client's bandwidth to choose the optimal Representation for playback. When downloading a Segment, Netflix keeps track of each chunk's length and download time to estimate the bandwidth at a given point in time. Knowing the instantaneous bandwidth, the Netflix player will request the subsequent Segments from the most suitable Representation, in order to adapt to the changing network conditions and to never incur in an interruption.

4.2.5. Media Presentation Description file

MPEG-DASH hierarchy is synthesized in Fig. 4.8. A DASH client which intends to receive pseudo-streamed content via MPEG-DASH must know the hierarchy for that particular content in order to request its Segments. The hierarchical subdivision into Periods, Adaptation Sets, Representations and Segments, for any given content, is represented in a Media Presentation Description (MPD) file [5]. The MPD file is an XML document which describes the hierarchy for a particular content that can be streamed. With this hierarchy, a DASH client can construct `HTTP GET` requests to download Segments from the corresponding Representations and Adaptation Sets, providing the streaming service to the end user.

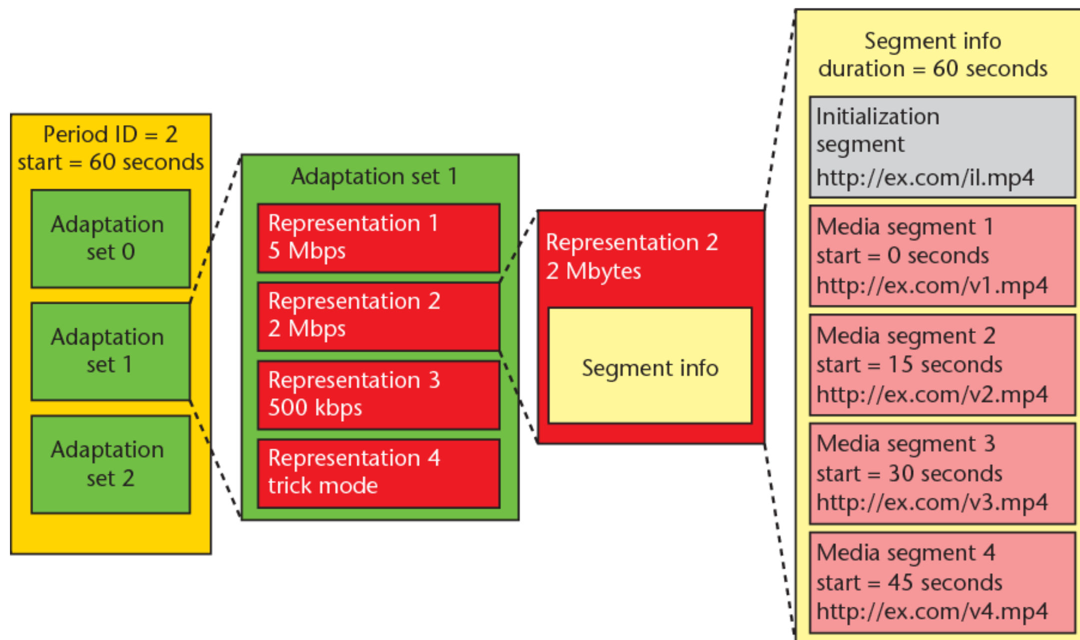


Fig. 4.8. MPEG-DASH hierarchy.

4.2.6. ISO Base Media File Format

All the Segments that make up a video are stored in servers. Each Segment constitutes its own video file (e.g.: `segment_1.mp4`, segment_2.mp4, …) and, as all video files, they are stored on disk following the binary syntax defined by a video container.

A video container wraps into a single file multiple streams (video, audio, subtitles) together with the metadata required for its synchronized playback. Containers define the binary syntax that video files must follow to multiplex the different streams that make up such video. For a further insight about video storage on computers, refer to Annex B.

MPEG-DASH supports Segments with formats based on the following MPEG containers:

- ISO Base Media File Format, defined in ISO/IEC 14496-12 – MPEG-4 Part 12.
- MPEG-2 Transport Stream, defined in ISO/IEC 13818-1.

These two containers are natively supported by MPEG-DASH because they both regard the subfragmentation of media, which is the most basic principle required for pseudo-streaming. Netflix in particular uses ISO Base Media File Format to store all its Segments. For this reason, we will now proceed to explain such container.

ISO Base Media File Format (ISOBMFF) container is defined in MPEG-4 Part 12 [7]. It is used as the basis for more specific containers such as MP4 (i.e.: MP4 container is a particularization of ISOBMFF container). Any file compliant with ISOBMFF is known as an 'ISO Base Media File'.

The binary syntax that ISOBMFF defines is based on a hierarchy of boxes. Boxes are the building blocks of ISO Base Media Files; all data within these files (i.e.: media streams and metadata) is contained in various types of boxes. Fig. 4.9 represents the succession of boxes that an ISO Base Media File takes when the video it wraps is split into different segments. All the information related to a

segment is contained within a pair made of `moof` box + `mdat` box. While the `mdat` box contains the multiplexed samples of the different streams that make up the segment, the `moof` box contains the metadata necessary to retrieve the samples in the `mdat` box.

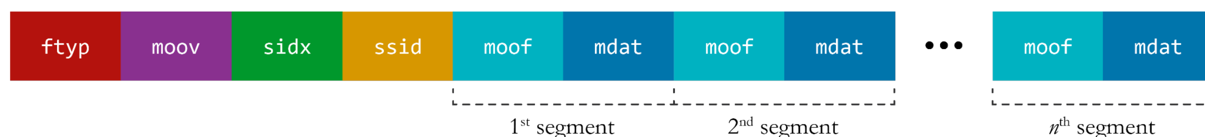


Fig. 4.9. Structure of ISOBMFF-based subfragmented files.

In subfragmented videos, the `sidx` box gathers the metadata related to the subdivision of the entire video into segments. The structure of `sidx` boxes is represented in Fig. 4.9.

0 1 2 3 4
reference_ID
timescale
earliest_presentation_time
first_offset
reserved | reference_count
1: r | referenced_size | subsegment_duration
SAP_type | SAP_delta_time
2: r | referenced_size | subsegment_duration
SAP_type | SAP_delta_time
⋮
reference_count: r | referenced_size | subsegment_duration
SAP_type | SAP_delta_time

Fig. 4.10. Segment Index Box definition in the ISO Base Media File Format [7].

Among all fields defined within a `sidx` box, the most important ones in order to understand Netflix architecture later on are the following:

- `timescale` determines the timescale that must be used by other fields which are in some way related to time. It is measured in samples per second. For example, `timescale=8000` implies that 1 second of playback corresponds to 8000 samples.
- `reference_count` provides the total number of segments which make up the content. For example, `reference_count=1250` implies that the video is split into 1250 segments. Each of the 1250 segments is described by the fields `r`, `referenced_size`, `subsegment_duration`, `SAP_type` and `SAP_delta_time`. This group of fields is inserted as many times as total segments available.
- `referenced_size` provides the size (in bytes) of the segment that is being described.
- `subsegment_duration` provides the duration of the segment that is being described, measured in samples per second. For example, if `subsegment_duration=40000` (considering `timescale=8000`), then the referenced segment lasts for 5 seconds.

Netflix stores all its content as subfragmented videos using ISOBMFF containers. This implies that the binary syntax of Netflix video files is the one depicted in Fig. 4.9, which means that every file will contain a `sidx` box describing the segments that make up the corresponding video.

4.3. Netflix architecture

Netflix does not use MPEG-DASH as its pseudo-streaming protocol, but it does implement a proprietary protocol which is heavily inspired in MPEG-DASH concepts and which also uses ISOBMFF containers for video storage. We can state this because when we analyzed requests performed by web browser Netflix clients in search of MPEG-DASH MPD files we ended up finding that Netflix sent the description of the Segmentation of its content in its own file, rather than using MPD files.

All Netflix content is identified by an ID. Every video (i.e.: every movie and every episode of a TV show) has its own, unique ID. The URL uses that ID to trigger the playback of the corresponding video. For example, if we wanted to watch in a web browser the first episode of the first season of *Stranger Things* (whose video ID is `80077368`), we would visit `https://www.netflix.com/watch/80077368`.

All the content is stored in Netflix's own CDN [8]. Such CDN consists of 60+ global data centers entirely run by Netflix, together with all the servers deployed at no cost in the data centers of ISPs which partner with Netflix. Netflix refers to the media servers spread across its own CDN as "Open Connect Appliances" (OCA for short). Deploying embedded OCAs within an ISP's data center is beneficial for both Netflix and the ISP itself: Netflix shortens distances with the end user to its fullest, and the ISP offloads substantial Netflix traffic towards the Internet. The most common Netflix content distribution paradigm is depicted in Fig. 4.11. On one hand, the end user connects to an OCA found in its ISP's data center to download the content to play. On the other hand, OCAs within ISPs can connect via a VPN to OCAs in Netflix's own data centers to provide additional resiliency and to enable nightly content fill and update. Other paradigms involve clients with an ISP which does not have Netflix OCAs. In those cases, the client will connect directly to an OCA in one of Netflix's own data centers to download the desired content.

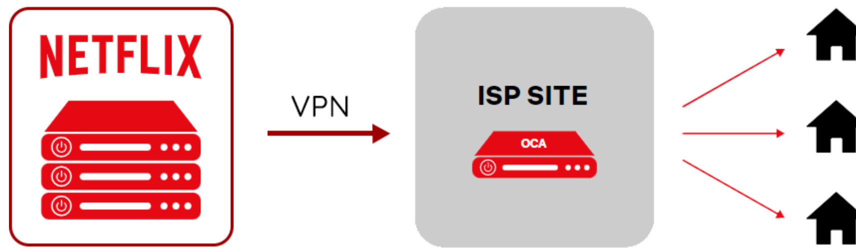


Fig. 4.11. The most common Netflix content distribution paradigm [8].

The OCAs store the entirety of Netflix's catalog. Each Representation that makes up a given Netflix video is stored as a separate file. To illustrate this concept better, Fig. 4.12 shows a depiction of how *Stranger Things* S1-E1 (i.e.: `80077368`) Representations could be stored.

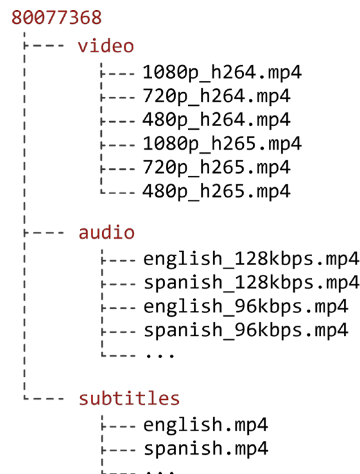


Fig. 4.12. Different Representations being stored in separate files.

Netflix stores its media using containers based on ISOBMFF (4.2.6.) following the structure previously defined in Fig. 4.9.

In order to start the playback of a video, the client must first know the Adaptation Sets, Representations and Segments available for that content. MPEG-DASH would deliver this information to the client via a MPD file, but Netflix uses its own document with its own structure, yet embracing the concepts defined by MPEG-DASH. This is the only major difference between Netflix and MPEG-DASH clients. From this point onward, we will be referring to Netflix's MPD-equivalent file as "Netflix Manifest" or simply as "Manifest".

The Netflix client retrieves the Manifest for a particular video by making a request to a specific endpoint:

```
https://www.netflix.com/nq/msl_v1/cadmium/pbo_manifests/%5E1.0.0/router
```

In order to return a valid Manifest, such endpoint must know, among other information, the ID of the video that the user wants to download. All the required information is passed to the server via a `POST` request, only after ciphering all that information via MSL (Message Security Layer) [9], which is a flexible secure messaging framework implemented over HTTPS. This implies that a sniffer which has access to the decrypted HTTPS traffic (e.g.: debugging tools within browsers) will not have access to the plain text Manifest, as both its request and reply are implemented on-top of MSL.

Thus, the next step that the Netflix client performs is the decryption of the MSL message to retrieve the Manifest. In the particular scenario of browser clients, the player is based on JavaScript (`cadmium-playercore.js`). This script will be in charge of handling all Segment requests, only after decrypting the received Manifest. The techniques used throughout the development of this work to obtain deciphered Netflix Manifests will be discussed in 5.2.1.1.

Netflix Manifests use JSON format. The structure used in Manifests, alongside some annotations which describe the equivalences with MPEG-DASH MPD files, are represented in Fig. 4.13.

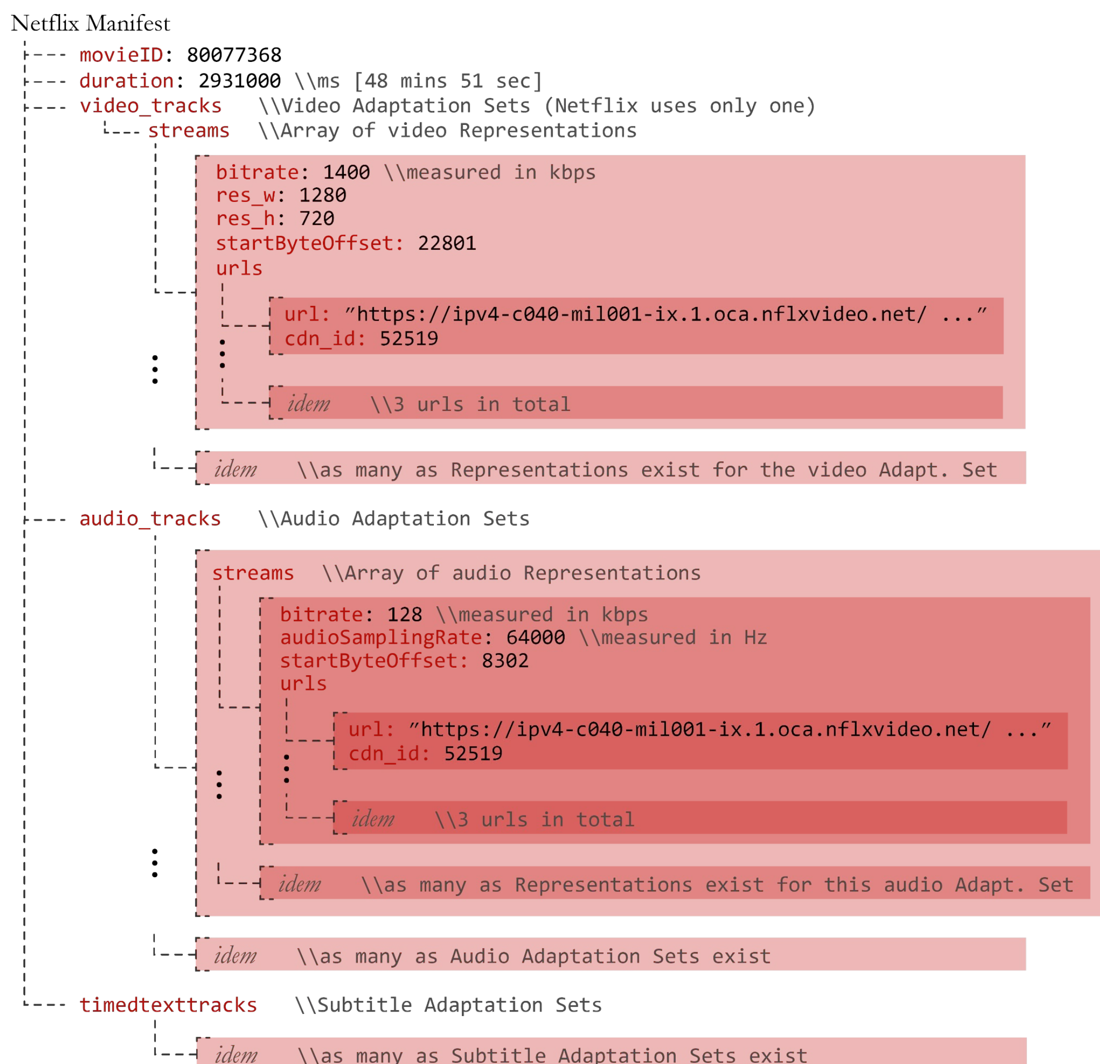


Fig. 4.13. Snippet of a simplified Netflix Manifest.

The Manifest shown in Fig. 4.13 is simplified. The actual Manifests have more fields which have not been included in the Figure for the sake of readability, but the ones depicted are those which represent the breakdown of the video into Adaptation Sets and Representations. The Manifest does not convey any subdivision into Periods because Netflix content is single-period (4.2.1.).

The `audio_tracks` and `timedtexttracks` arrays contain multiple Adaptation Sets (i.e.: as many as different language options exist for audio and subtitle playback, respectively). The `video_tracks` array contains a single video Adaptation Set because Netflix does not implement different video variants depending on the language or any other possible factors (4.2.2.).

Within each Adaptation Set, the `streams` array wraps all its available Representations. The parameter `bitrate` within each Representation helps the Netflix client decide which Representation is the most suitable to download based on the current user's network conditions.

The `urls` array contains different endpoints which serve the Segments that make up the corresponding Representation. This array always includes 3 different endpoints, defined by their URL and by an ID which identifies the OCA. Taking into account Netflix's CDN architecture, the endpoints offered to the user will vary depending on its ISP. These URLs are sorted from most to least preferred endpoint (i.e.: the client must first attempt to request the Segments to the first OCA, and if connection issues arise, the client can backup to the second OCA, and so on). The offered URLs include `GET` variables, which are used to internally pinpoint the corresponding Representation file. The structure of OCA's URLs is as follows:

```
https://ipv4-c040-mil001-ix.1.oca.nflxvideo.net/?o=1&v=70&e=1638394808&t=VEclzczW0n…
```

Once the client chooses a Representation to download Segments from, it uses its URLs to build the Segments requests. For example, let us suppose that the client wants to download Segments from the 1080p H.264 Representation of the video Adaptation Set of *Stranger Things* S1-E1. Let the endpoint chosen to request the Segments be `https://nflx.oca.foo/?[identifier_GET_parameters]`, where `[identifier_GET_parameters]` represents the `GET` variables used to pinpoint the 1080p H.264 Representation file If the client wanted to download the byte range 30000-75000 from the targeted file (because that corresponds to a desired Segment), then the URL that the player would have to generate to request that Segment would be:

```
https://nflx.oca.foo/range/30000-75000?[identifier_GET_parameters]
```

The client appends `/range/{firstByte}-{endByte}` in between the OCAs domain name and the `GET` parameters. The web server that processes this URL uses the `GET` variables to target the file (i.e.: the file for the 1080p H.264 video variant of *Stranger Things* S1-E1) and serves to the client the specified byte range from such file, serving the Segment as a result. But in order to start requesting Segments, the client must first know which Segments (i.e.: which byte ranges) are available for a given Representation (i.e.: which Segments make up the 1080p H.264 video variant of *Stranger Things* S1-E1) .

For that reason, prior to start requesting any further Segment, the client first asks for the SIDX box of the targeted file. Recall that the SIDX box of ISOBMFF containers describes the subdivision of the content into Segments. The `{firstByte}` and `{endByte}` variables in URLs are processed by the web server by taking the first byte of the SIDX box as reference – that is, the web server maps the byte `0` to the first byte of the SIDX box of a file, as depicted in Fig. 4.14.

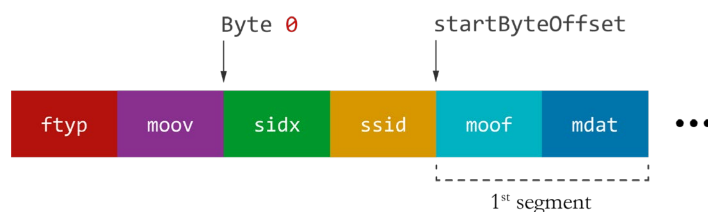


Fig. 4.14. Reference bytes for the web server to process `/range/{firstByte}-{endByte}`.

Thus, when the client requests the SIDX box of a Representation, it builds an URL as follows:

```
https://nflx.oca.foo/range/0-{endByte}?[identifier_GET_parameters]
```

Web browser Netflix clients are programmed to perform their SIDX requests setting `{endByte}` to the value determined by the following formula (extracted from `cadmium-playercore.js` player):

$$\texttt{\{endByte\}} = ceil(\frac{video\ duration\ [milliseconds]}{2000}) \cdot 12 + 2292$$

The duration of *Stranger Things* S1-E1, as stated in its Manifest (Fig. 4.13), is 2931000 ms (48 mins 51 seconds); therefore, if a Netflix client wanted to request any of its SIDX boxes, `{endByte}` would be `19884`. Thus, the resulting URL built for SIDX box requests would be:

`https://nflx.oca.foo/range/0-19884?[identifier_GET_parameters]`

Once the client has access to a SIDX box, it parses it (based on the binary structure defined on Fig 4.10.) and gains access to all the Segments that make up that particular Representation. The Table 4.3 represents the Segment composition for the 1080p H.264 video variant for *Stranger Things* S1-E1. All its information was obtained from parsing the Representation's SIDX box.

Segment number	**Size** [bytes]	**Duration** [seconds]	**Byte range to request**
1	58833	3.670	`/range/22801-81633`
2	122763	1.335	`/range/81634-204396`
3	131766	5.756	`/range/204397-336162`
4	228934	5.756	`/range/336163-565096`
…			
864	172922	10.010	`/range/421528418-421701339`
865	187329	10.010	`/range/421701340-421888668`
866	140097	8.592	`/range/421888669-422028765`

Table 4.3. Segment breakdown for *Stranger Things* S1-E1 (1080p H.264), extracted from the Representation's SIDX box.

The parameter `startByteOffset`, defined for each Representation independently (Fig. 4.13), pinpoints the first byte of media content (see Fig. 4.14). The `startByteOffset` parameter, together with the sizes of each Segment, help build the byte ranges necessary for Segment requests. The byte ranges associated with each Segment of *Stranger Things* S1-E1 (1080p H.264) are also represented in Table 4.3. For this particular Representation, as `startByteOffset=22801` (defined in Fig. 4.13) and the size of the first Segment is 58833 bytes, the byte range linked to the first Segment is `/range/22801-81633`. The subsequent byte ranges are calculated taking the immediately preceding range as reference.

The player must download the SIDX box of all Representations it wants to have access to. Typically, 2 SIDX boxes are downloaded when playback starts (as shown in Fig. 4.15, which captures the sequence of downloaded resources in a browser Netflix client). One SIDX will belong to the video Adaptation Set, and the other will belong to the audio Adaptation Set of choice. If the user changes language at some point during playback, or if the client detects major variations in bandwidth and decides to switch

to another resolution, the player will request more SIDX boxes to have access to the Segments of the most suitable Representations.

Name	Size	Time	Waterfall
0-19883?o=1&v=21&e=1642660713&t=lYagAaCW_eBd...	20.3 kB	74 ms	
0-19883?o=1&v=21&e=1642660713&t=9eS0a1PMt8iwi...	20.3 kB	73 ms	
router?reqAttempt=1&reqName=bind&clienttype=akira...	3.8 kB	89 ms	
97093-129869?o=1&v=21&e=1642660713&t=9eS0a1P...	33.2 kB	7 ms	
70118-163186?o=1&v=21&e=1642660713&t=lYagAaC...	93.5 kB	30 ms	
129870-391560?o=1&v=21&e=1642660713&t=9eS0a1...	262 kB	41 ms	
163187-259788?o=1&v=21&e=1642660713&t=lYagAaC...	97.1 kB	40 ms	
259789-415920?o=1&v=21&e=1642660713&t=lYagAaC...	157 kB	34 ms	
415921-535101?o=1&v=21&e=1642660713&t=lYagAaC...	120 kB	14 ms	
391561-654073?o=1&v=21&e=1642660713&t=9eS0a1...	263 kB	17 ms	
535102-695654?o=1&v=21&e=1642660713&t=lYagAaC...	161 kB	14 ms	

Fig. 4.15. Sequence of downloaded resources on a browser Netflix client when starting playback.

As the sizes of Segments are static, playing a certain video will always trigger the same Segment download sequence, which enables the identification of Netflix playback traffic despite being encrypted.

5. Identification of Netflix content

This work explains the process carried out to identify Netflix encrypted content based on the architecture that Netflix implements as of February 2022. The work is subdivided in 3 major tasks:

1. **Identification of Netflix Segment sizes in PCAP files.**
   This task tackles the processing of PCAP files to:
   - identify which TCP packets belong to Netflix Segment downloads.
   - calculate the size of all downloaded Segments.
2. **Indexing of Netflix catalog.**
   This task entails the creation of databases which map all Netflix Segment sizes to video IDs, and store video metadata.
3. **Matching.**
   The matching stage determines precisely which Netflix video was being watched based on the Netflix Segments spotted in a PCAP file. This task involves the algorithms developed to correlate all Segment sizes identified in a PCAP file to the videos indexed in the databases.

While the task ***2)*** *Indexing of Netflix catalog* constitutes its own process, the tasks ***1)*** *Identification of Netflix Segment sizes* and ***3)*** *Matching* are performed back to back in the process of identification of a video.

5.1. Identification of Netflix Segment sizes in PCAP files

In order to be able to tell which Netflix video was being watched based on a PCAP file, such capture file must be analyzed first to determine which Netflix Segments were downloaded while capturing. Thus, we implemented a program which takes a PCAP file as input and returns as output the sizes of all Netflix Segments downloaded within the capture. The workflow of the PCAP analyzer is synthesized as a black box diagram in Fig. 5.1.

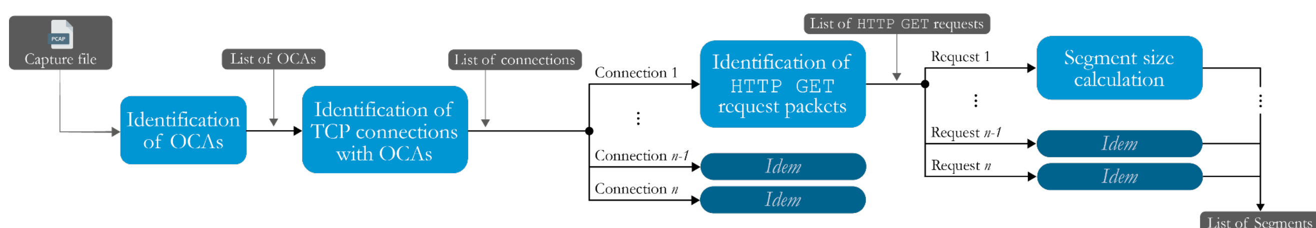


Fig. 5.1. PCAP analyzer workflow.

Briefly, the PCAP analyzer takes a capture file and lists all the TCP connections established with OCAs. It later identifies all the HTTP requests within each of those TCP connections as they must belong to Segment requests, and it finally processes the reply to each of those requests to extract the size of each downloaded Segment. The subsequent sections offer a further explanation of each of the stages that make up the analyzer.

5.1.1. Identification of OCAs

The first task that the PCAP analyzer must do is filter all traffic that does not imply communication with a Netflix OCA. Thus, the program must first know which endpoints, among all registered in the capture, correspond to an OCA.

The script lists all IPs involved in TCP connections and requests their domain names individually (reverse DNS resolution). An OCA can be easily identified by its domain name because it contains "`.oca.`". These domain name mappings do not change overtime in the timescale we were using. Therefore, all IPs whose domain name contains "`.oca.`" will be considered Netflix OCAs. All Netflix Segments available within the capture must belong to TCP connections established with OCAs.

5.1.2. Identification of TCP connections with OCAs

Once the program knows which endpoints correspond to OCAs, it will now isolate each TCP connection with an OCA to later on look for Segments in each connection individually. This process is done via `tcpdump`.

As an example, let us imagine a PCAP file which captured traffic with 5 different OCAs (e.g.: `23.246.50.158`, `23.246.50.174`, `23.246.51.146`, `23.246.51.173`, `23.246.51.175`). By requesting `tcpdump` to output all TCP packets addressed to the HTTPS port of an OCA, and by parsing such output, the program can easily list all the TCP connections established with all OCAs. A snippet of the lines that `tcpdump` outputs in this stage is represented in Fig. 5.2.

```
17:39:33.639240 IP 10.20.1.192.58806 > 23.246.50.158.443: tcp 0
17:39:35.423383 IP 10.20.1.192.48800 > 23.246.51.173.443: tcp 0
17:39:35.455741 IP 10.20.1.192.48800 > 23.246.51.173.443: tcp 0
17:39:35.455936 IP 10.20.1.192.48800 > 23.246.51.173.443: tcp 517
17:39:35.516662 IP 10.20.1.192.47732 > 23.246.51.146.443: tcp 0
17:39:35.516693 IP 10.20.1.192.47734 > 23.246.51.146.443: tcp 0
17:39:35.531063 IP 10.20.1.192.48800 > 23.246.51.173.443: tcp 0
17:39:35.531182 IP 10.20.1.192.48800 > 23.246.51.173.443: tcp 0
17:39:35.531275 IP 10.20.1.192.48800 > 23.246.51.173.443: tcp 0
17:39:35.533945 IP 10.20.1.192.48800 > 23.246.51.173.443: tcp 64
...
```

Fig. 5.2. Outgoing packets of TCP connections with OCAs (`tcpdump`).

The Table 5.1 presents all the TCP connections established with OCAs, obtained after parsing the entire output of `tcpdump` in the context of the aforementioned example. All Netflix Segments available within the capture must have been downloaded from these TCP connections:

OCA		Client Port
IP	Name	
23.246.50.158:HTTPS	ipv4-c056-mil001-ix.1.oca.nflxvideo.net:HTTPS	58800
		58802
		58804
		58806
23.246.50.174:HTTPS	ipv4-c088-mil001-ix.1.oca.nflxvideo.net:HTTPS	56024
23.246.51.146:HTTPS	ipv4-c034-mil001-ix.1.oca.nflxvideo.net:HTTPS	47732
		47734
		47744
		47746
23.246.51.173:HTTPS	ipv4-c093-mil001-ix.1.oca.nflxvideo.net:HTTPS	48800
23.246.51.175:HTTPS	ipv4-c095-mil001-ix.1.oca.nflxvideo.net:HTTPS	47194

Table 5.1. TCP connections with OCAs.

5.1.3. Identification of Segment requests in each connection

The purpose of this stage is to identify all TCP packets involved in the download of a Segment. The workflow of this stage starts by identifying all `HTTP GET` packets within a TCP connection. The identification of HTTP packets will be explained in the following paragraphs. With all `HTTP GET` requests identified, all the downstream TCP packets associated with each request are grouped independently. These groups are parametrized by the SN range that the TCP packets cover. Grouping all downstream TCP packets by its originating `HTTP GET` request will help the following stage in the Analyzer calculate the exact size of each downloaded Segment independently. A depiction of the aim of this stage is represented in Fig. 5.3.

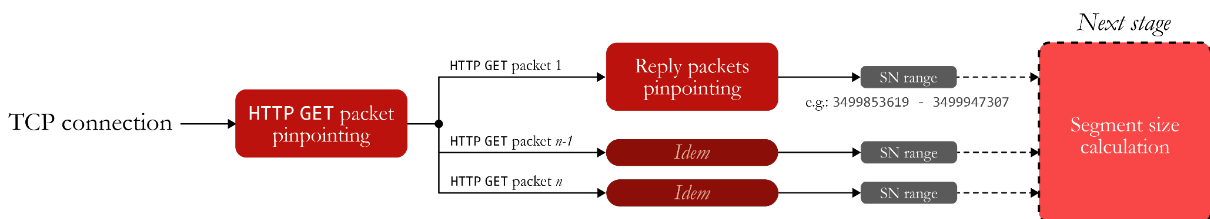


Fig. 5.3. Pinpointing of all TCP packets involved in the download of a Segment.

For example, the SN range `3499853619 - 3499947307` covers all the TCP packets involved in the download of the Segment triggered by the first `HTTP GET` request. It is important to remark that this SN range does not reveal the size of the downloaded Segment – that is, the size of this first Segment is not 93689 bytes (where 93689 = `3499947307` – `3499853619` + 1). This is because the downloaded payload includes HTTP and TLS headers, which should be ignored for the calculation of the actual size of the Segment. The SN ranges are used by the next stage to access the corresponding TCP packets in order to process their payload and obtain the exact size of the Segment.

In order to prevent prompting the calculation of the payload size for a requested resource which is not a Segment, it is important that all `HTTP GET` packets within the TCP connection correspond to Segment requests. If there exists any HTTP request for a resource other than a Segment, it must be ignored. It is also important to remark that Netflix does not implement HTTP pipelining in its Segment requests: only

one Segment can be requested at a time per connection. Therefore, in terms of HTTP packets, a new `HTTP GET` packet will not be captured within a TCP connection until the Segment requested in the preceding `HTTP GET` request is fully downloaded. In other words, in between two consecutive `HTTP GET` packets there will only be TCP packets corresponding to the download of a Segment. The PCAP analyzer takes advantage of this to isolate all TCP packets in between two consecutive `HTTP GET` requests to later on reassemble the entire TCP payload, which is a necessary step for the calculation of the exact downloaded Segment size.

Performing Deep Packet Inspection (DPI) to pinpoint all `HTTP GET` packets among all TCP frames is not an option because they are encrypted (HTTPS). Considering that a TCP connection with an OCA is only used to download Segments, all the traffic that we can expect from this sort of connections are:

- **`Client –> OCA:443.`** `HTTP GET` packets (requesting Netflix Segments).
- **`Client <– OCA:443.`** TCP packets transporting a Segment, as a reply to a `HTTP GET` packet.
- **`Client –> OCA:443.`** ACKs.

The only data that the client ever sends are the HTTP requests for Segments. The rest of the packets that the client transmits are ACKs, which are very short in size since they do not contain any payload. When it comes to the `HTTP GET` packets, their size (physical level) averages around 900 bytes in the set of captures carried out throughout the development of this work. The program is able to pinpoint all `HTTP GET` packets within a TCP connection by isolating packets which comply with the following requirements:

- The packet is sent to the HTTPS port of an OCA (**`Client –> OCA:443`**).
- The size of the packet is greater than 890 bytes.
  Fig. 5.4 overlaps the distributions of the sizes of the following packets within a TCP connection with an OCA:
  1. TLS `client_hello` packets (the largest packets of the entire TLS handshake process).
  2. `HTTP GET` requests for Segments.
  3. `HTTP GET` requests for resources other than Segments.

  Attending to such figure, it is safe to assume that any packet greater than 890 will be an `HTTP GET` request for a Segment.

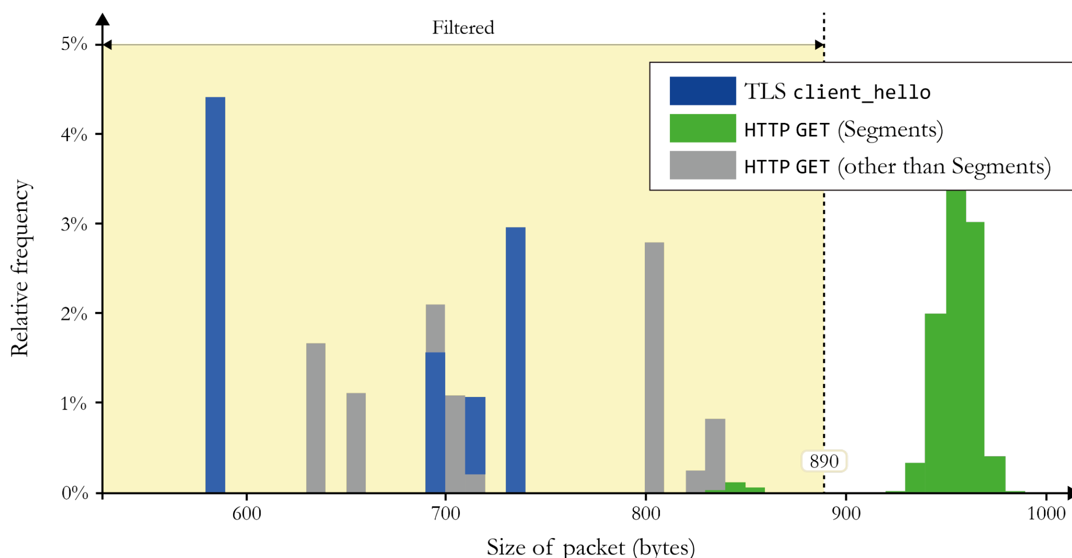


Fig. 5.4. Packet size distributions for TLS `client_hello` packets and HTTP requests.

After removing all packets which do not comply with the previous criteria, the remaining frames must correspond to HTTP GET packets. Continuing with the example and filtering with tcpdump (following the previous criteria) the TCP connection with Client Port = 56024 and OCA = 23.246.50.174:HTTPS (Table 5.1), the following packets remain (which must be HTTP GET packets):

```
15:45:02.683874 IP 10.20.1.192.56024 > 23.246.50.174.443: Flags [P.], seq 3084383485:3084384392, ack 3499853619 ...

15:45:03.037701 IP 10.20.1.192.56024 > 23.246.50.174.443: Flags [P.], seq 3084384392:3084385315, ack 3499947308 ...

15:45:03.675901 IP 10.20.1.192.56024 > 23.246.50.174.443: Flags [P.], seq 3084385315:3084386238, ack 3500033083 ...

15:45:03.818196 IP 10.20.1.192.56024 > 23.246.50.174.443: Flags [P.], seq 3084386238:3084387163, ack 3500166719 ...

...
```

Fig. 5.5. HTTP GET packets in TCP connection with OCA = 23.246.50.174:HTTPS and Client Port = 56024 (tcpdump).

Since the first HTTP GET packet has ack = 3499853619, the SN of the first byte of the server's reply to that request will be 3499853619. Analogously, the SN of the first byte of the reply to the second HTTP request will be 3499947308, which implies that the byte with SN 3499947307 is the last byte of the first Segment. Consequently, the download for the first Segment is located between the SN range 3499853619 - 3499947307. Table 5.2 collects the SN ranges that can be derived from Fig. 5.5.

HTTP GET request num.	SN range of reply
1	3499853619 - 3499947307
2	3499947308 - 3500033082
3	3500033083 - 3500166718
...	

Table 5.2. SN ranges for each Segment download.

Knowing the SN range that covers the download of a particular Segment helps the program analyze the downstream TCP payload lying in that SN range to calculate precisely the size of the downloaded Segment in the next stage.

5.1.4. Segment size calculation

This stage performs the calculation of the exact size of a Segment using tcpflow, which is a UNIX command line program that extracts the payload of all TCP packets involved in the TCP connections described by a Wireshark *DisplayFilter*.

Let us take the example of the first `HTTP GET` request described in Table 5.2, whose reply covers SN range `3499853619` - `3499947307`. To extract the payload of all TCP packets involved in the first Segment download, the PCAP analyzer calls `tcpflow` using the following filter:

```
(src 23.246.50.174 and src port 443) and (dst 10.20.1.192 and dst port 56024) and
tcp[4:4]>=3499853619 and tcp[4:4]<3499947308
```

`tcp[4:4]` is the expression used by Wireshark's *DisplayFilters* to refer to the SN of a TCP packet. The previous filter is referencing all TCP packets sent **`Client:56024 <- OCA:443`** (where `OCA` = `23.246.50.174`) which comply with $3499853619 \leq SN < 3499947308$ – that is, the filter is pinpointing all downstream TCP packets involved in the download of the first Segment found in the **`Client:56024 <-> OCA:443`** TCP connection.

The construction of the TCP packets involved in Segment downloads (whose payload will be extracted by `tcpflow`) is depicted in Fig. 5.6.

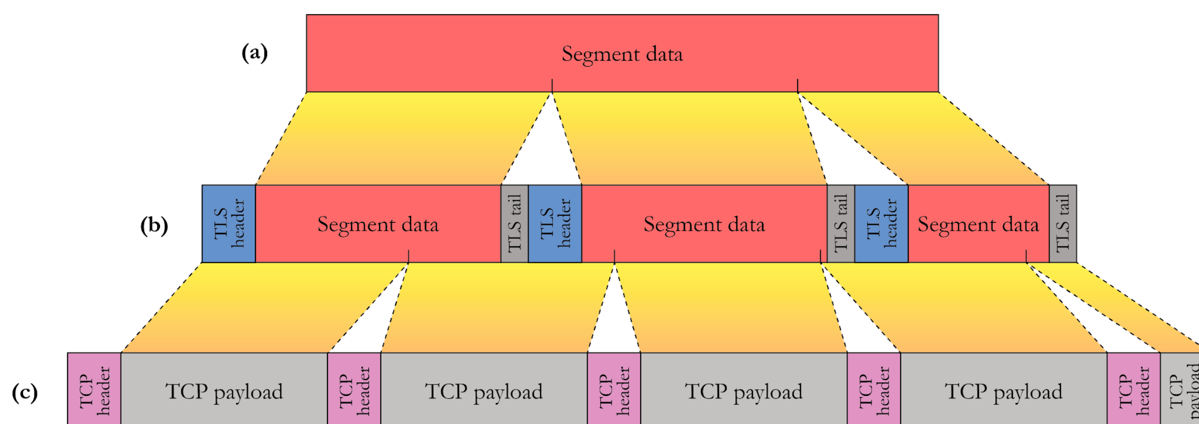


Fig. 5.6. Construction of TCP packets involved in a Segment download.

The application data stream **(a)** (i.e.: the Segment) is split and introduced (encrypted) into several consecutive TLS frames, forming the TLS stream **(b)**. TLS frames are packets of the *Transport Layer Security* cryptographic protocol, used to provide secure communications over a network. HTTPS is implemented over TLS.

The frame that yields after prepending a TLS header to the encrypted TLS payload (TLS payload = application payload + TLS tail) is denoted as a "TLS Record". The communication with Netflix servers is implemented over `TLSv1.3`. The structure of a `TLSv1.3` Record is depicted in Fig. 5.7.

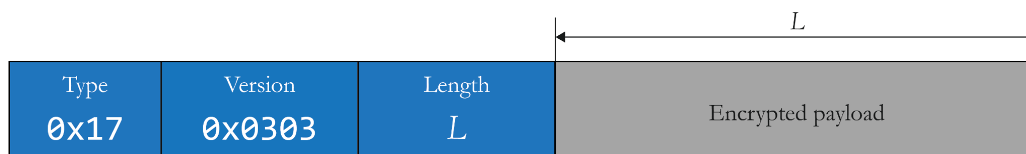


Fig. 5.7. Structure of a `TLSv1.3` Record used for Application Data transportation.

The TLS Record Header is made up of the following fields:

- The field `Type` determines what sort of data is being transported on the encrypted payload. The value `0x17` is used for "`Application Data`". The HTTP replies (`200 OK` + Segment) are considered Application Data . Other values are used during the TLS handshake process.

- The field `Version` specifies the TLS protocol version.
- The field `Length` determines the size (in bytes) of the encrypted data.

The structure of the encrypted payload within `TLSv1.3` Records is represented in Fig. 5.8:

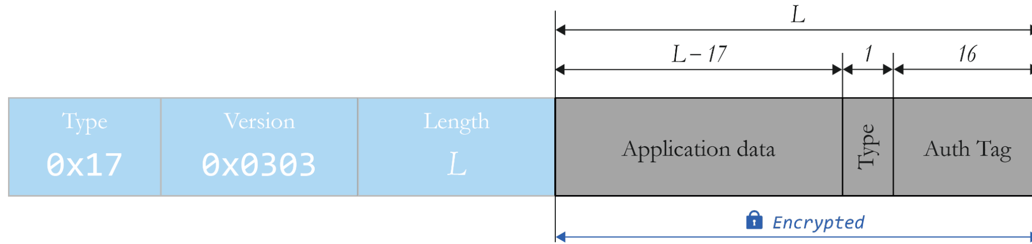


Fig. 5.8. Structure of the encrypted data in `TLSv1.3` Records.

The encrypted data in `TLSv1.3` Records [11] arises from adding a 17-byte tail to the actual application data, and then ciphering it. The application data introduced within TLS Records can have a maximum size of 16KB (2^{14} bytes = 16384 bytes) [10]. Since the encryption operation does not affect the overall payload size (e.g.: 500 bytes of encrypted content implies that the decrypted content was 500 bytes long as well), the size of the actual deciphered application data sent over a `TLSv1.3` Record can be calculated using the `Length` field as follows:

Deciphered application data size [bytes] = `Length` [bytes] – 17

This formula is also represented in Fig. 5.8. Calculating and adding up the sizes of the "`Application Data`" field among all the TLS Records that make up the download of a Segment, results in the exact size of said decrypted Segment. Fig. 5.9 represents this summation process with a Segment which takes 4 TLS Records to download.

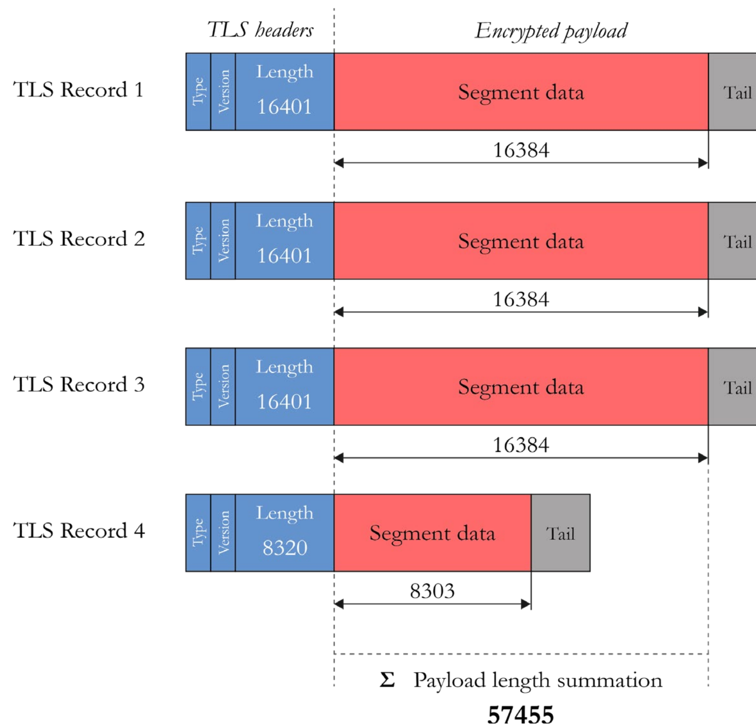


Fig. 5.9. Summation process resulting in the exact size of a downloaded Segment.

Performing this summation process for all requests found among all TCP connections with OCAs yields a list with the sizes of all the Segments downloaded within the capture.

However, prior to any calculation, the algorithm must discard the first TLS header, as it always belongs to an independent TLS Record which contains exclusively the `HTTP OK` heading message (see Fig. 5.10), which is not relevant for the calculation of the size of the Segment itself.

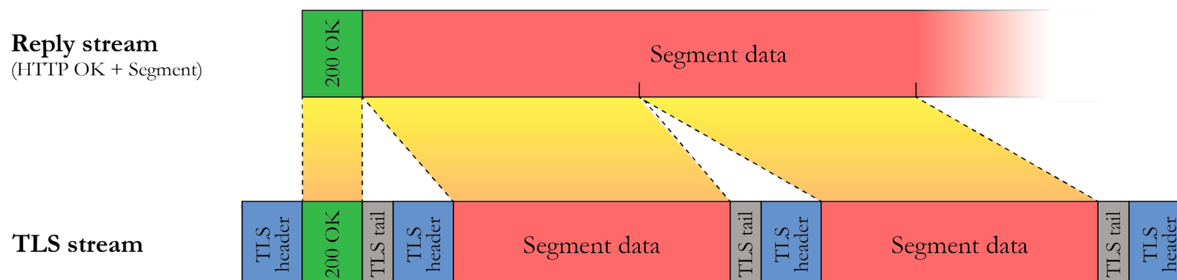


Fig. 5.10. `HTTP OK` header forming its own independent TLS Record.

Once the first TLS Record is discarded, the remaining Records contain Segment Data. At this point, the first byte in the stream must correspond to the first byte of a TLS header (that of the TLS Record after the `HTTP OK` Record). The processing that this summation algorithm performs to scan the TLS Record headers across the reconstructed payload stream is depicted in Fig. 5.11.

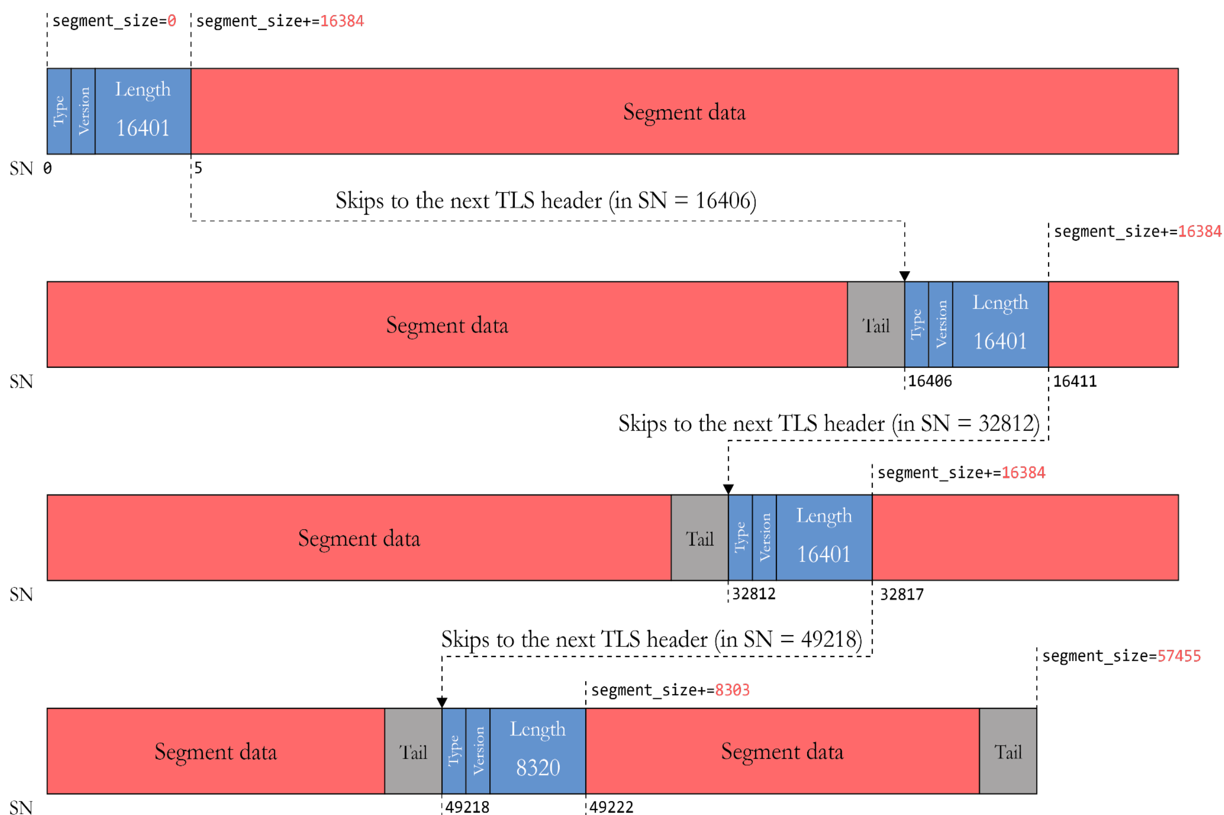


Fig. 5.11. Processing of TLS Record Headers.

The processing of the `Length` field of the first TLS header adds 16384 bytes to the Segment size summation, but also contributes to skipping the entire TLS payload to the next TLS header (which starts in SN $16406 = 5 + 16401$). This process is held iteratively until the end of the stream, concluding with an accurate size for the downloaded Segment.

As the calculation of Segment sizes considers exclusively the `Length` field of TLS Record Headers, TCP packets which happen to not contain any TLS Header are irrelevant in this process. For this reason, this summation process is resistant (to a large degree) to missing TCP packets which, for whatever reason, have not been captured. Fig. 5.12 highlights all special casuistry that can arise when processing TLS Record Headers.

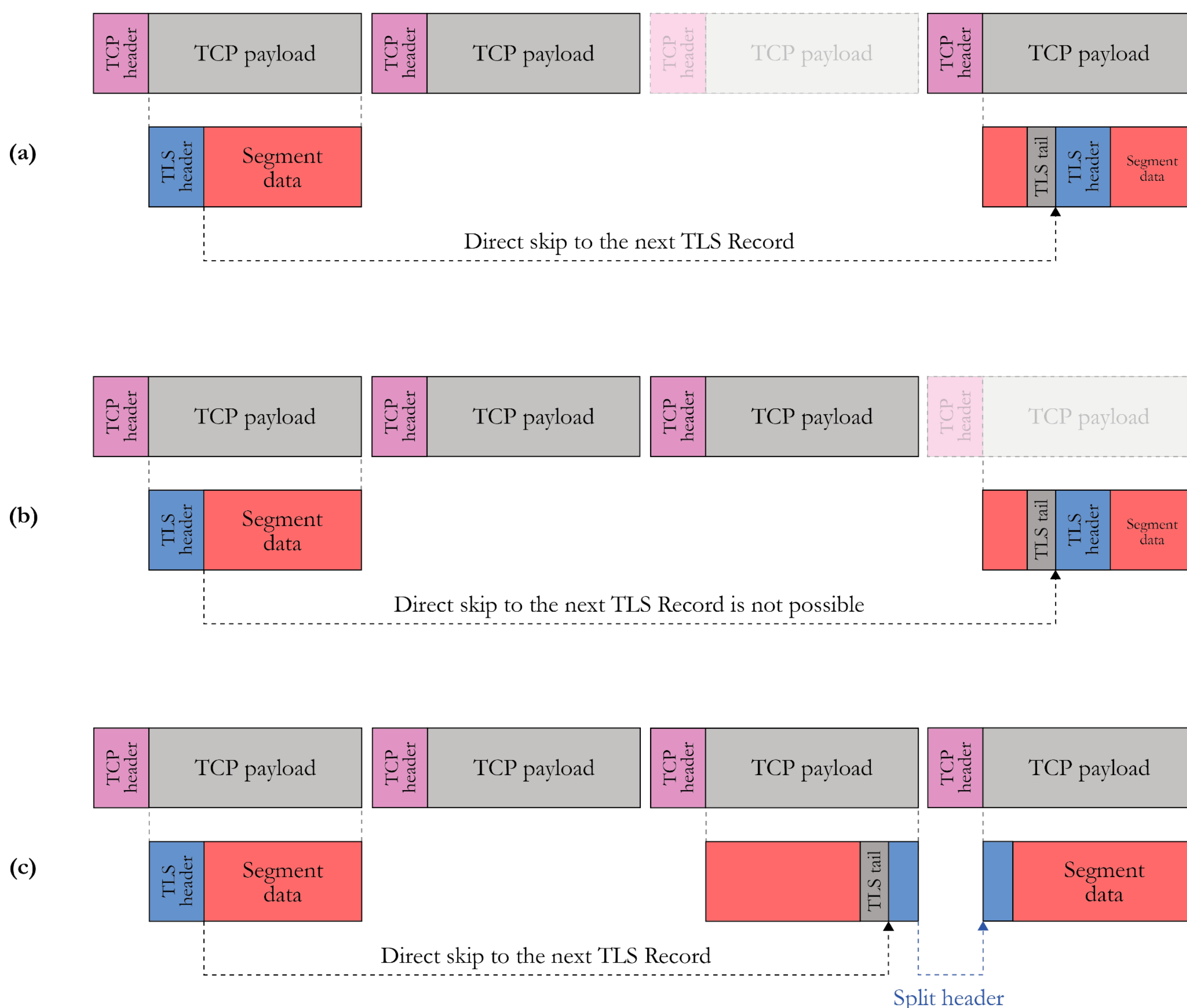


Fig. 5.12. Special scenarios in TLS Record Header processing.

In the scenario **(a)**, the 3rd TCP packet was not captured. However, that is not a problem for the algorithm because the `Length` field of the TLS header in the first TCP packet makes the program skip the calculation directly to the 4th TCP packet, where the next TLS header is.

However, in **(b)** the 4th TCP packet is missing. This is fatal, because the algorithm is told to skip the calculation to a byte which was not captured. In this case, the summation process aborts and the program is unable to determine the exact size of the Segment.

Finally, **(c)** demonstrates that the algorithm recognises when a TLS header happens to be split into 2 consecutive TCP packets. The program reassembles the header and the summation process continues.

5.2. Indexing of Netflix catalog

Now that the PCAP analyzer is able to tell which Netflix Segments were downloaded within a capture with encrypted traffic, the program needs to correlate such Segments to a Netflix video. We must store all the sizes of all the Segments that make up every single video in Netflix catalog to be able to perform a match down the line. This task entails the implementation of two different databases to store:

1. Segment sizes, and their relationship to a Netflix video.
2. Video metadata (e.g.: movie or TV show title, season and episode number, episode name, etc.)

5.2.1. Indexing of Segments within a video

As a brief summary, let us recall that all Netflix videos have a unique ID and are made up of Adaptation Sets (i.e.: different media streams) and Representations (i.e.: different variants in terms of resolution, bitrate or codec), which are split into Segments. The hierarchy of Adaptation Sets and Representations for a particular Netflix video is represented in a Netflix Manifest (4.3.), which the client requests to start performing Segment downloads. Prior to downloading any Segment, the client first retrieves a SIDX box (4.2.6.), which describes how the subdivision into Segments was performed for a given Representation. The client utilizes the SIDX box to build the HTTP requests for Segments (4.3.). The playback of the same Representation of the same Netflix video always triggers the same Segment download sequence, and this is why Netflix traffic is identifiable.

Table 4.3 represented the sizes of the Segments that make up the 1080p H.264 video variant for *Stranger Things* S1-E1, which was obtained from just parsing its corresponding SIDX box. Analogously, if we parse the SIDX box of each video Representation available for all videos in the entire Netflix catalog, we obtain the complete mapping between Segment sizes and Netflix videos (i.e.: which Segment sizes correspond to which Netflix videos).

The process that we have developed to retrieve the Segment sizes for a certain video is described in Fig. 5.13.

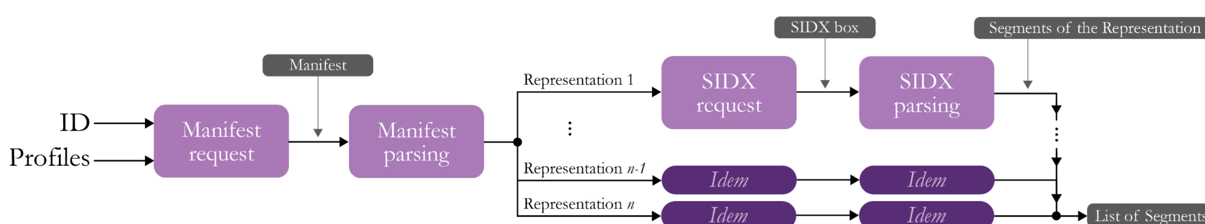


Fig. 5.13. Segment extraction workflow.

First of all, the Manifest for the desired video is downloaded. Parsing the Manifest (which has a JSON structure) grants access to all the Representations that make up that video. Each Representation offers a list of URLs from where that Representation can be downloaded. The program uses the URLs of a Representation to trigger the download of its SIDX box, which later parses to extract all its Segment sizes. The subsequent sections offer a further explanation of each of the stages that make up the Segment extraction process.

5.2.1.1. Manifest request

Prior to any other step, the program needs to obtain the Manifest of the video whose Segments we want to extract. As it was explained in 4.3., all the Manifest requests are processed by `https://www.netflix.com/nq/msl_v1/cadmium/pbo_manifests/%5E1.0.0/router`, and are sent over Netflix's own Security Layer protocol (MSL).

Throughout the development of this work, Manifest requests were handled with `pymsl` [12], which is a Python library that requests Netflix Manifests via MSL's API. A `pymsl` client is initialized by authenticating as a real Netflix user. This client accepts multiple authentication methods, including the email and password for your Netflix account. An instance of this client implements the `load_manifest` method, which takes the ID of a Netflix video and returns its corresponding Manifest.

A `pymsl` client accepts extra parameters (to be passed when initializing the client) which set specific configurations for the returning Manifest. One of these extra parameters is "`profiles`", which is an array that includes the codecs supported by the user. This list is processed by the server to build a customized Manifest which includes Representations encoded with codecs that the client supports. Fig. 5.14 includes the snippet of Python code necessary to initialize a `pymsl` client (setting specific profiles) and to download the Manifest for `80092521`.

```
p = ['playready-h264mpl30-dash', 'playready-h264mpl31-dash'...]
client = pymsl.MslClient(user_auth_data,profiles=p)
manifest = client.load_manifest(80092521)
```

Fig. 5.14. Python code snippet to request the Manifest for `80092521`.

The JavaScript player used in browser clients (`cadmium-playercore.js`) includes a hardcoded list with all video, audio and subtitles codecs available for playback. To request a Manifest, the client first constructs a "`profiles`" array appending all the codecs it supports, among all defined in the hardcoded list. All the video codecs defined in "`profiles`" are gathered in Table 5.3. The nametags used to identify each codec follow, generally, the following syntax:

`[codec][profile][level][variant]`

For example, the nametag "`playready-h264mpl30-dash`" refers to the H.264 codec using the "Main Profile" (`mp`) and the level `3.0`. For the sake of understanding what this nomenclature refers to, in the scope of codecs a profile defines a set of preconfigured capabilities enabled for the encoding process; while a level is a specified set of constraints that indicate a degree of required decoder performance for a profile. For instance, the H.264 codec defines 3 different profiles: Baseline Profile (`bp`), Main Profile (`mp`) and High Profile (`hp`). The difference between profiles (within the same codec) lies in the set of algorithms and features used during the encoding process (i.e.: higher profiles lead to higher compression efficiencies at the expense of a higher computational cost). H.264 also defines a wide set of levels, which range from L1 to L6.2. In particular, Level `3.0` aims for a 720x576@30fps output video, regardless of the chosen profile. The higher the profile and the higher the level, the better the quality of the output video. Each codec defines its own set of profiles and levels.

As every codec defined within "`profiles`" leads to very different compressions, a same video, when compressed with the different codecs, leads to different Segment sequences. This behavior is depicted in Fig. 5.15.

Codec family	Profiles	
`playready-h264-dash`	playready-h264mpl30-dash playready-h264mpl31-dash playready-h264mpl40-dash playready-h264hpl22-dash	playready-h264hpl30-dash playready-h264hpl31-dash playready-h264hpl40-dash
`vp9-dash-cenc`	vp9-profile0-L21-dash-cenc vp9-profile0-L30-dash-cenc	vp9-profile0-L31-dash-cenc vp9-profile0-L40-dash-cenc
`hevc-dash-cenc`	hevc-main10-L30-dash-cenc hevc-main10-L31-dash-cenc hevc-main10-L40-dash-cenc	hevc-main10-L41-dash-cenc hevc-main10-L50-dash-cenc hevc-main10-L51-dash-cenc
`hevc-dash-cenc-prk`	hevc-main10-L30-dash-cenc-prk hevc-main10-L31-dash-cenc-prk	hevc-main10-L40-dash-cenc-prk hevc-main10-L41-dash-cenc-prk
`hevc-dash-cenc-prk-do`	hevc-main10-L30-dash-cenc-prk-do hevc-main10-L31-dash-cenc-prk-do hevc-main10-L40-dash-cenc-prk-do	hevc-main10-L41-dash-cenc-prk-do hevc-main10-L50-dash-cenc-prk-do hevc-main10-L51-dash-cenc-prk-do
`hevc-dv5-dash-cenc-prk`	hevc-dv5-main10-L30-dash-cenc-prk hevc-dv5-main10-L31-dash-cenc-prk hevc-dv5-main10-L40-dash-cenc-prk	hevc-dv5-main10-L41-dash-cenc-prk hevc-dv5-main10-L50-dash-cenc-prk hevc-dv5-main10-L51-dash-cenc-prk
`hevc-dv5-dash-cenc-prk-do`	hevc-dv5-main10-L30-dash-cenc-prk-do hevc-dv5-main10-L31-dash-cenc-prk-do hevc-dv5-main10-L40-dash-cenc-prk-do	hevc-dv5-main10-L41-dash-cenc-prk-do hevc-dv5-main10-L50-dash-cenc-prk-do hevc-dv5-main10-L51-dash-cenc-prk-do
`hevc-hdr-dash-cenc`	hevc-hdr-main10-L30-dash-cenc hevc-hdr-main10-L31-dash-cenc hevc-hdr-main10-L40-dash-cenc	hevc-hdr-main10-L41-dash-cenc hevc-hdr-main10-L50-dash-cenc hevc-hdr-main10-L51-dash-cenc
`hevc-hdr-dash-cenc-prk`	hevc-hdr-main10-L30-dash-cenc-prk hevc-hdr-main10-L31-dash-cenc-prk hevc-hdr-main10-L40-dash-cenc-prk	hevc-hdr-main10-L41-dash-cenc-prk hevc-hdr-main10-L50-dash-cenc-prk hevc-hdr-main10-L51-dash-cenc-prk
`hevc-hdr-dash-cenc-prk-do`	hevc-hdr-main10-L30-dash-cenc-prk-do hevc-hdr-main10-L31-dash-cenc-prk-do hevc-hdr-main10-L40-dash-cenc-prk-do	hevc-hdr-main10-L41-dash-cenc-prk-do hevc-hdr-main10-L50-dash-cenc-prk-do hevc-hdr-main10-L51-dash-cenc-prk-do
`av1-dash-cbcs-prk`	av1-main-L20-dash-cbcs-prk av1-main-L21-dash-cbcs-prk av1-main-L30-dash-cbcs-prk av1-main-L31-dash-cbcs-prk	av1-main-L40-dash-cbcs-prk av1-main-L41-dash-cbcs-prk av1-main-L50-dash-cbcs-prk av1-main-L51-dash-cbcs-prk

Table 5.3. Codecs and profiles supported by Netflix.

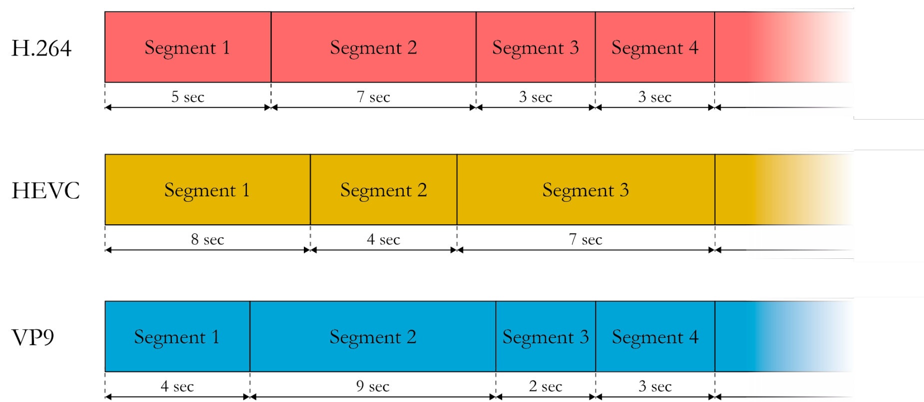


Fig. 5.15. Different Segment subdivision depending on the codec.

To obtain the totality of Segments that make up all qualities for all the codifications for a certain video, we must configure the `pymsl` client to accept all codec variants. This, however, is not as easy as setting "`profiles`" to have all the variants declared in Table 5.3. This is because the Manifest server will build for the client a custom Manifest which exclusively points to videos encoded with the most favorable profiles among all profiles declared by the client. For example, if a client declares that it supports all H.264 and VP9 profiles, the server will only include the VP9-encoded videos in the returning Manifest, as the server internally considers the VP9 codec to be preferable for playback over H.264.

In fact, H.264 is the least preferred codec, yet the Manifest server requires the "`profiles`" array to include the following H.264 profiles (`"playready-h264mpl30-dash"`, `"playready-h264mpl31-dash"`, `"playready-h264hpl30-dash"`, `"playready-h264hpl31-dash"`), otherwise the Manifest request fails. Crafting a "`profiles`" array which includes the compulsory H.264 profiles, together with the profiles whose Segments we actually want to extract, leads (on paper) to a Manifest pointing to the codec-specific Representations. The theoretical strategy defined to extract all Segments (for all qualities for all the codifications) for a certain video is depicted in Fig. 5.16.

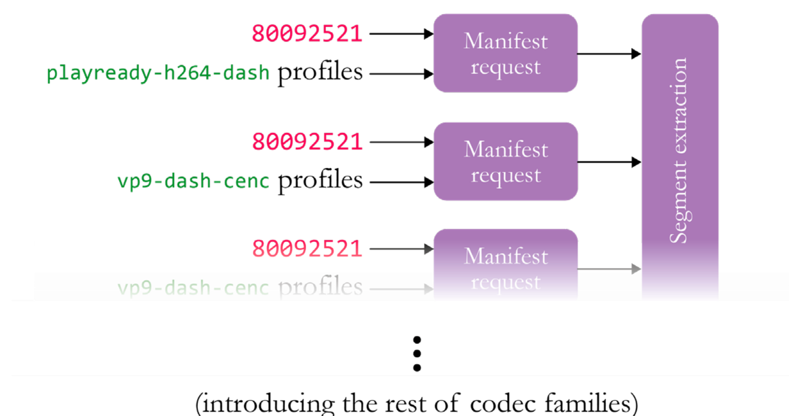


Fig. 5.16. Strategy for the extraction of all Segments of `80092521`.

The previous strategy is only theoretical, as on practice we discovered that there are multiple factors which affect the profiles that the server includes in its Manifests (e.g.: user agent, user device, etc.). The server needs to verify a very particular set of conditions in order to correctly build a Manifest with the codecs described in "`profiles`". If the conditions are not met (e.g: if the declared user agent is not what the server would expect for the codecs declared) the server limits to construct a Manifest with the 4 compulsory `playready-h264-dash` codecs, ignoring any preference set in "`profiles`". Throughout the development of this work we only managed to extract Segments from `playready-h264-dash` and `vp9-dash-cenc` profile families. Extraction for the rest of families is pending investigation.

5.2.1.2. SIDX box parsing

Once a Manifest has been downloaded, it is time to parse it to access the URLs leading to the SIDX boxes. Considering the Manifest's structure defined in Fig. 4.13, the parser must navigate to the `urls` array within each video Representation. Having the URL for each Representation pinpointed, we can request the SIDX box for each Representation as explained in 4.3.. Then, the program will parse each SIDX box individually to extract the Segment sizes for each video Representation.

The SIDX box parser we developed takes a SIDX box as input and returns a list of Segment sizes as output. The extraction of the Segments (and the parsing of the box in general) is performed taking into consideration the binary structure of SIDX boxes, defined in Fig. 4.10.

Once the SIDX box for each video Representation has been parsed, the program has a list with all the Segment sizes that make up that video (for the codification included in the Manifest).

5.2.1.3. Storage

The previous stage yields a list of Segment sizes (which depend on the codec) associated with a video ID. These relationships must be stored in databases to enable the matching later on.

To store the relationships between Segment sizes and videos, mongoDB [13] was chosen. Its non-relational model and its orientation to documents is optimal for the sort of data that we want to store. The philosophy guiding the structure of our mongoDB database consists in having one file per Segment size and appending in that document all video IDs that have that Segment size within their Segments. Each file within the database is defined by the parameter "`_id`", which takes the value of a Segment size, and is used internally by mongoDB's searching algorithm. Alongside the "`_id`", each file also defines the parameter "`list`", which is an array that includes the IDs of all videos in the Netflix catalog which have a Segment whose size is the one defined in "`_id`". A representation of this philosophy is defined in Fig. 5.17.

Fig. 5.17. Structure of Segment sizes database (mongoDB).

Fig. 5.17 exemplifies that videos `80166369`, `70208599`, `541018` and `70281532` (among other IDs which have not been represented) have a Segment, somewhere in its Representations, whose size is 50376 bytes. The same reasoning can be applied for the Segment sizes `98589` and `112632`. The database adopts this structure for the sake of efficiency for the matching algorithms.

If the previous dataset was to be introduced in a SQL-based database (e.g.: MySQL), the Segment ↔ ID mapping would have to follow the structure depicted in Fig. 5.18, where both the "`segment`" and "`video_id`" column would have to work as primary keys.

segment	video_ID
50376	80166369
50376	70208599
50376	541018
50376	70281532
98589	80166369
98589	70108777
98589	80046311
98589	21878564
112632	80166369
112632	70108777
112632	70273614
112632	70098887

Fig. 5.18. Structure of the equivalent Segment sizes database (MySQL).

To test the performance between implementations (mongoDB vs MySQL), we introduced in both databases all the Segments of a reduced set of 150 videos. Fig. 5.19 represents the timespan of the insertion of such dataset in both databases, where mongoDB results clearly faster.

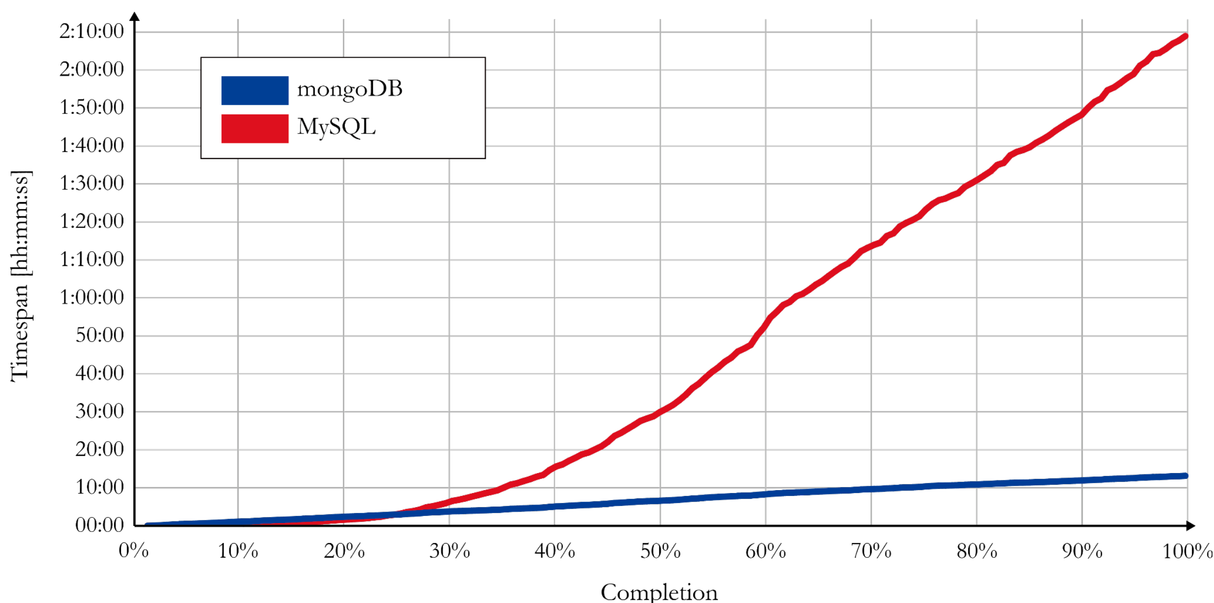


Fig. 5.19. Insertion times comparison between mongoDB and MySQL.

Furthermore, Fig. 5.20 represents a scatter plot which describes the search times comparison between mongoDB and MySQL for 1000 different queries. Each query consists of returning all IDs associated with a given Segment size. A point x = 1.5 ms and y = 1.25 s (notice that each axis is measured in different units) implies that a query which took 1.5 ms to be solved in mongoDB took 1.25 seconds in MySQL. Despite having most of the points laying over the X axis (representing queries which took fairly the same to be solved in mongoDB and MySQL), a considerable amount of queries took much longer to be solved with MySQL. As it will be explained in 5.3, the matching algorithms conduct several queries before concluding which video was being watched, so mongoDB is definitely preferable over MySQL when it comes to searches. For our work, MongoDB clearly outperforms MySQL both in insertions and searches.

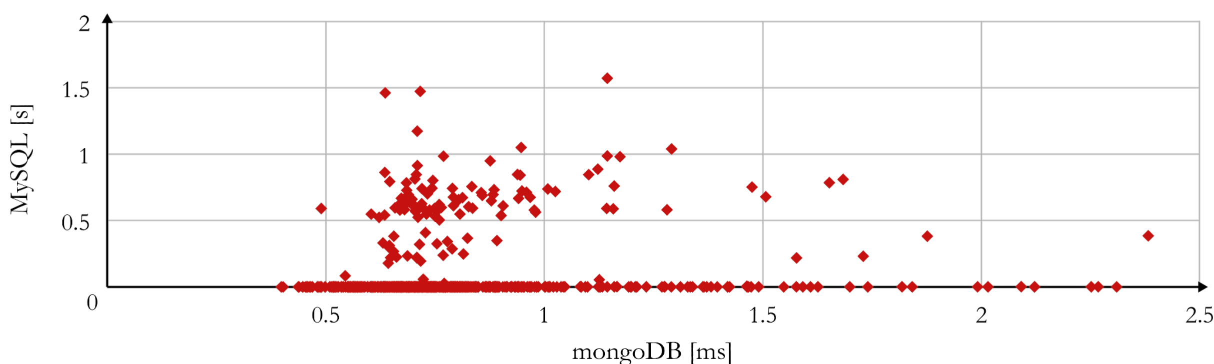


Fig. 5.20. Search times comparison between mongoDB and MySQL.

Data insertion into mongoDB's Segment database works as follows: if a Segment size is not registered yet – that is, if a file for a specific Segment size does not exist, a new document is created, with its `"_id"` parameter set to the value of the new Segment size and its `"list"` parameter set to an array containing the Netflix video ID associated to such Segment size. From that point onward, when a

Segment size is already registered, any new insertion related to that Segment size appends the video ID to the existing `“list”` array.

5.2.2. Indexing of video metadata

While the video ID is used internally to store the Segment ↔ video mappings for the entire Netflix catalog, it would be preferable to display to the user the metadata of the product visualized during capture (e.g.: movie or TV show title, season and episode number, episode name, etc.) rather than an unintuitive ID.

To gain access to all this metadata, we used the API that Netflix clients internally use to retrieve this information: `shakti`. This API was designed for private use, but the community has managed to discover at least some of the requests that it can handle, which have been documented in [14]. Among all requests feasible, the following two stand out:

- Request for all product IDs within a Netflix category (i.e.: Action & Adventure, Anime, Sci-fi, Comedy, etc.).
- Request for video metadata.

In order to request all video IDs within a Netflix category given the category ID (e.g.: category `6548` – comedies), the following array must be sent as payload to the API’s endpoint:

```
["genres", 6548, {"from": 0, "to": 3500}]
```

This request returns up to 3500 IDs associated with category `6548`. The 3500 threshold was set arbitrarily to guarantee that all products linked to a category are returned in each request – no category does really have as many products.

Within the returning list of IDs, TV shows (or in general any content split into episodes) are represented by an ID which refers to the entire TV show as a whole. For example, the TV show *Disenchantment* has `80095697` as its ID, which does not point to any video or episode but to the whole product. In fact, each episode has its own ID (first episode: `80117711`, second episode: `80145115`, third episode: `80145116`, etc.) but these episode IDs are not returned within this `shakti` request.

In order to request all the metadata for a movie, it is enough to know its ID. For example, to request metadata for `60023642` (*Spirited Away*), the following arrays must be sent as payload to the API’s endpoint:

```
["videos",60023642,["title","synopsis","maturity","summary","seasonCount"]],
["videos",60023642,"boxarts","_1280x720","webp"]
```

The metadata retrieved via `shakti` when sending the previous payload is represented in Fig. 5.21 in its JSON structure.

Spirited Away
60023642

```
{}
├--- title: "Spirited Away"
├--- id: 60023642
├--- synopsis: "Chihiro wanders into a magical world where a witch rules -- and those who dis
├--- maturity: "0+"
├--- type: "movie"
└--- boxart: "https://occ-0-784-778.1.nflxso.net/dnm/api/v6/X194eJsgWBDE2aQbaNdmCXGUP-Y/AAAAB
```

Fig. 5.21. Video metadata retrieved via `shakti` for `60023642`.

When it comes to TV shows, the metadata retrieved includes the information relative to its seasons and episodes. All the extra parameters that the API requires to access the metadata from seasons and episodes are detailed in [14]. A snippet of the information retrieved via `shakti` associated with the TV show `80095697` (*Disenchantment*) is represented in Fig. 5.22:

Disenchantment
80095697

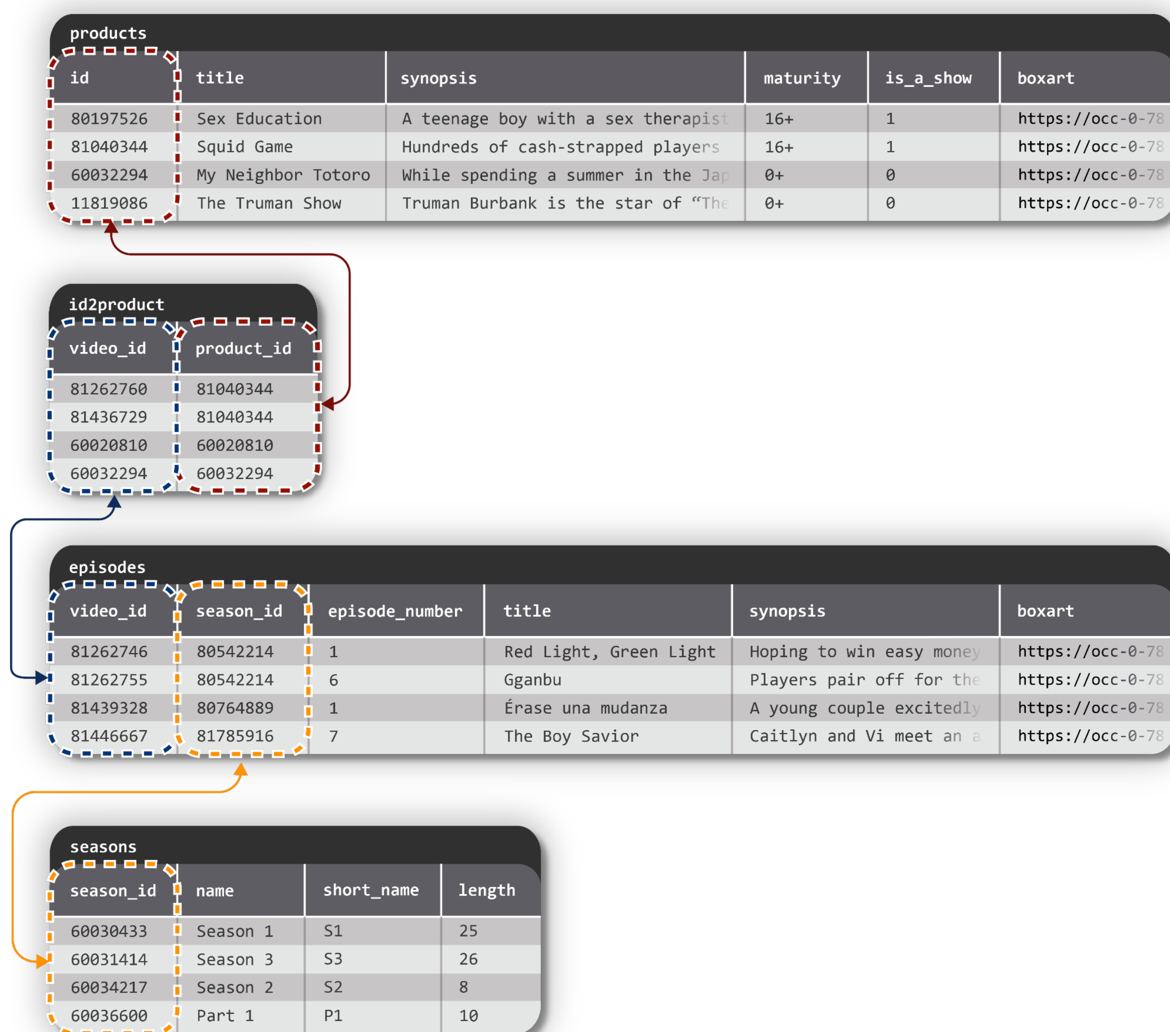
```
{}
├--- title: "Disenchantment"
├--- id: 80095697
├--- synopsis: "Princess duties call, but she'd rather be drinking. Free-spirited Bean exasp
├--- maturity: "12+"
├--- type: "show"
├--- boxart: "https://occ-0-784-778.1.nflxso.net/dnm/api/v6/X194eJsgWBDE2aQbaNdmCXGUP-Y/AAAAB
├--- seasonCount: 3
└--- seasons
        ├--- 80117549:  //First season
        |       ├--- season_info
        |       |       ├--- name: "Part 1"
        |       |       ├--- shortName: "P1"
        |       |       └--- length: 10
        |       └--- episodes
        |               |
        |               ├---o P1-E1  A Princess, an Elf, and a Demon Walk Into a Bar
        |               |     80117711
        |               |     {}
        |               |     ├--- title: "A Princess, an Elf, and a Demon Walk Into a Bar"
        |               |     ├--- episode_number: 1
        |               |     ├--- synopsis: "Princess Bean already has the wedding day blues
        |               |     └--- boxart: "https://occ-0-784-778.1.nflxso.net/dnm/api/v6/9p
        |               |
        |               ├--- 80117715   //Second episode of first season
        |               ├--- 80117716
        |               ├--- 80117717
        |               └--- ...
        |
        ├--- 80174140:  //Second season
        └--- ...
```

Fig. 5.22. Video metadata retrieved via `shakti` for TV show `80095697`.

All the retrieved metadata relative to movies and TV shows (together with its seasons and its episodes) was stored in a MySQL database. A relational database was chosen as it is the most straight-forward and intuitive method to operate with this sort of datasets. The implementation involves 4 tables:

1. **`products`**. This table contains the ID, title, synopsis, boxart and maturity rating for all Netflix products. A flag is also stored which determines whether a given product is a movie or a TV show.
2. **`id2product`**. This table relates the ID of each video to the ID of the product to which it belongs. For example, `video_id` `80117711` (*Disenchantment* S1-E1 ID) is linked to `product_id` `80095697` (*Disenchantment* TV show ID). For movies, `video_id` = `product_id`.
3. **`episodes`**. This table is populated with all the episodes that make up all Netflix's TV shows. Each episode is defined by its video ID, the ID of the season to which it belongs, the number and title of the episode, a brief synopsis and its boxart.
4. **`seasons`**. This table is populated with all the seasons that make up all Netflix's TV shows. Each season is defined by its season ID, its full name, its short name and its length in episodes.

Fig. 5.23 offers a depiction of all 4 tables and the relationships established among them.



products

id	title	synopsis	maturity	is_a_show	boxart
80197526	Sex Education	A teenage boy with a sex therapist	16+	1	https://occ-0-78
81040344	Squid Game	Hundreds of cash-strapped players	16+	1	https://occ-0-78
60032294	My Neighbor Totoro	While spending a summer in the Jap	0+	0	https://occ-0-78
11819086	The Truman Show	Truman Burbank is the star of "The	0+	0	https://occ-0-78

id2product

video_id	product_id
81262760	81040344
81436729	81040344
60020810	60020810
60032294	60032294

episodes

video_id	season_id	episode_number	title	synopsis	boxart
81262746	80542214	1	Red Light, Green Light	Hoping to win easy money	https://occ-0-78
81262755	80542214	6	Gganbu	Players pair off for the	https://occ-0-78
81439328	80764889	1	Érase una mudanza	A young couple excitedly	https://occ-0-78
81446667	81785916	7	The Boy Savior	Caitlyn and Vi meet an a	https://occ-0-78

seasons

season_id	name	short_name	length
60030433	Season 1	S1	25
60031414	Season 3	S3	26
60034217	Season 2	S2	8
60036600	Part 1	P1	10

Fig. 5.23. MySQL tables for Netflix metadata storage and their interrelationship.

5.2.3. Crawler

The crawler is the program which iterates through Netflix catalog and extracts their Segments (as described in 5.2.1.) and their metadata (as explained in 5.2.2.) to introduce them in their corresponding database. Its workflow is represented in Fig. 5.24 and further explained in the subsequent paragraphs.

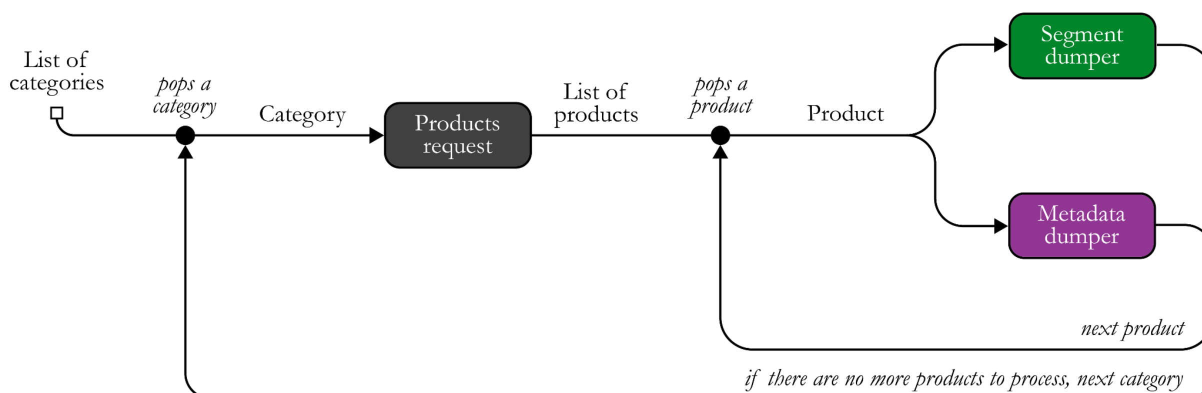


Fig. 5.24. Netflix catalog crawler workflow.

The list of categories which feed the crawler is documented in [15], which describes absolutely every category used internally by Netflix. For each category, all its products are requested; and for each product:

1. all its Segments are dumped (for the movie itself or for all its underlying episodes if the product is a TV show; in both cases for different profile families as represented in Fig. 5.15).
2. all its metadata is dumped (for the product itself and for all its underlying episodes if the product is a TV show).

Once the crawler runs out of categories to process, it sleeps for 12 hours and it then starts another crawling loop, cycling through the entire Netflix catalog again in search for new products, new episodes of already existing TV shows and new Representations.

The crawler keeps track of all the Representations whose Segments have successfully been dumped on its own independent MySQL table called `profiles`. A snippet of this table is included in Fig. 5.25.

profiles

video_id	profile	width	height	bitrate	fps
81085383	playready-h264hpl31-dash	1280	720	665	23.976
81085383	playready-h264hpl40-dash	1920	1080	1204	23.976
81085383	vp9-profile0-L31-dash-cenc	1280	720	567	23.976
81085383	vp9-profile0-L40-dash-cenc	1920	1080	874	23.976

Fig. 5.25. `profiles` table, storing all the Representations whose Segments are successfully introduced in mongoDB.

Keeping track of all the Representations enables the crawler to detect if a never-seen Representation has arised for a given ID (based on its resolution, framerate or bitrate), prompting its Segment dump.

As of January 28th 2022, the databases are populated with Segments and metadata of 6069 products, of which 1804 correspond to TV shows (or in general, content split into episodes) and 4265 to movies. Overall, 45012 videos (among movies and episodes of TV shows) populate the Netflix catalog.

5.3. Matching

Matching is the stage which correlates the Segments downloaded in a capture with the Segment sizes and metadata indexed in the databases, resulting in an accurate identification of the captured Netflix content.

Two different algorithms have been developed to perform the matching operation: **ballot** and **deterministic**. Each approach correlates the Segments in the capture with the sizes in mongoDB differently, resulting in different success ratios and computational costs.

5.3.1. Ballot algorithm

"Ballot" is a matching algorithm which correlates the Segments by performing a votation process in which the most upvoted ID will be considered to belong to the Netflix video captured. An example of this algorithm in action is depicted in Fig. 5.26 and further detailed in the paragraphs below.

The list at the top of Fig. 5.26 contains the sizes of all the Segments downloaded within a capture of Netflix traffic. Such list outputs after analyzing the corresponding capture file with the method explained in 5.1.

The algorithm retrieves from mongoDB the list of video IDs associated with each Segment size and keeps track of how many times has each ID appeared among iterations. The information that mongoDB returns includes the parameter `list`, which is an array listing all video IDs which have the Segment size in `_id`. For example, in the first iteration, the array mapped to Segment size `252711` is `[81030842, 81264749, 70244169, 80197874, 81043228, ...]`.

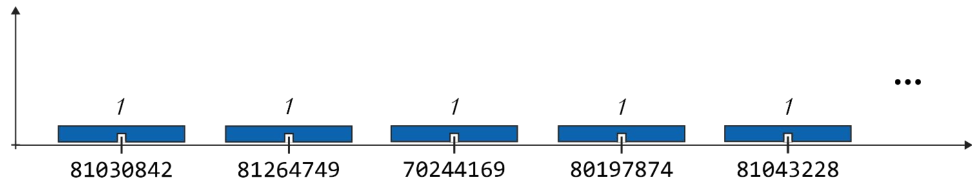
As of the 1st iteration, all the IDs included in the array have appeared once overall. However, as of the 2nd iteration, `81264749` has appeared twice; and as of the sixth, 6 times. In this particular iteration, the second most voted ID is `81030842`, which has appeared 3 times overall.

The ballot algorithm is set to hit a match when the difference between the most popular ID and the second most popular ID is 4 votes. For a justification for this threshold and a general performance comparison when setting the threshold to different values, refer to 6.2. For this example, the match hits after 7 iterations.

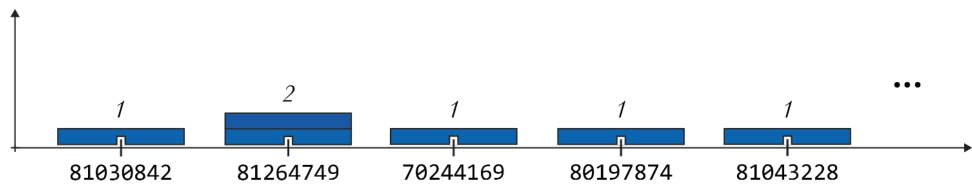
```
//List of captured Segments

[252711, 922245, 10644, 969232, 629934, 113085, 169177, 282495, 727589, 922245,
313072, 1478, 541701, 90091, 128567, 970490, 446675, 1062458, 369444, 1275858, 291205,
144433, 996218, 615268, 727589, 220781, 308792, 1661478, 1031142, 914533, 969232,
500858, 650343, 5214545, 544880, 2283410, 541701, 3436822, 832161, 2666657]
```

1 252711 ↦ {_id:252711, list: [81030842, **81264749**, 70244169, 80197874, 81043228,…]}

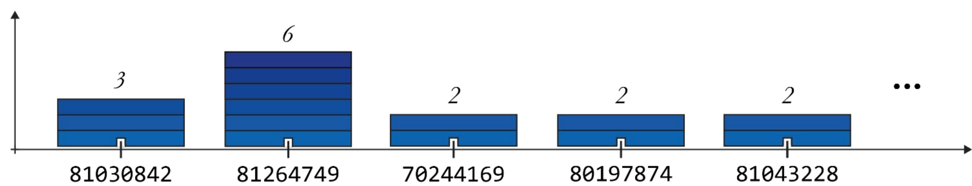


2 922245 ↦ {_id:922245, list: [81439371, 80220477, 81024085, **81264749**, 81411818,…]}



⋮

6 113085 ↦ {_id:113085, list: [**81264749**, 70244169, 80133036, 70177139, 80150490,…]}



7 169177 ↦ {_id:169177, list: [81362866, 81232380, **81264749**, 80098810, 81110498,…]}

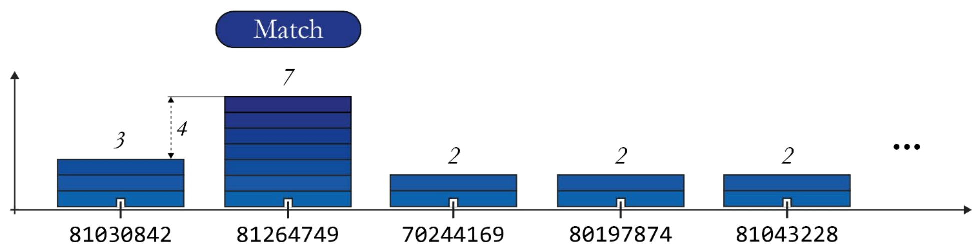


Fig. 5.26. Ballot matching for the identification of 81264749.

5.3.2. Deterministic algorithm

"Deterministic" is a matching algorithm that specifies which video was being watched on capture by intersecting the lists that mongoDB returns until a single ID remains. It is based on the theoretical fact that the ID of the captured content must be present in all the ID lists retrieved from mongoDB in each iteration. An example of this algorithm in action is depicted in Fig. 5.27 and further detailed in the paragraphs below.

```
//List of captured Segments

[252711, 922245, 10644, 969232, 629934, 113085, 169177, 282495, 727589, 922245, 313072,
1478, 541701, 90091, 128567, 970490, 446675, 1062458, 369444, 1275858, 291205, 144433,
996218, 615268, 727589, 220781, 308792, 1661478, 1031142, 914533, 969232, 500858,
650343, 5214545, 544880, 2283410, 541701, 3436822, 832161, 2666657]
```

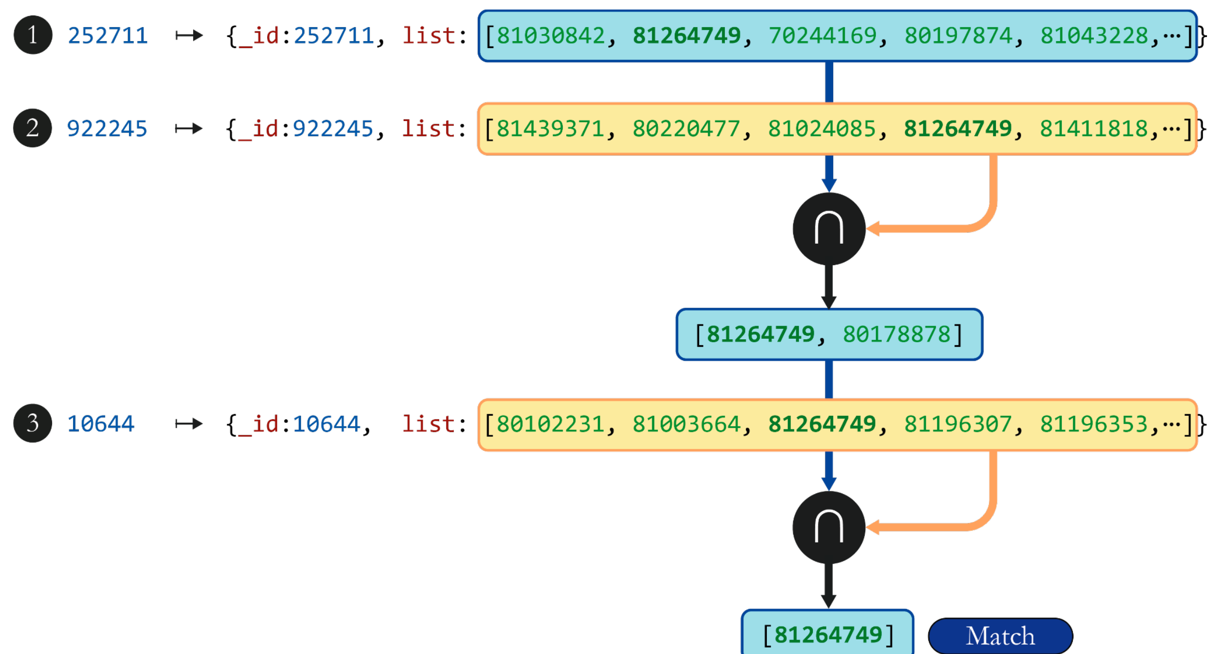
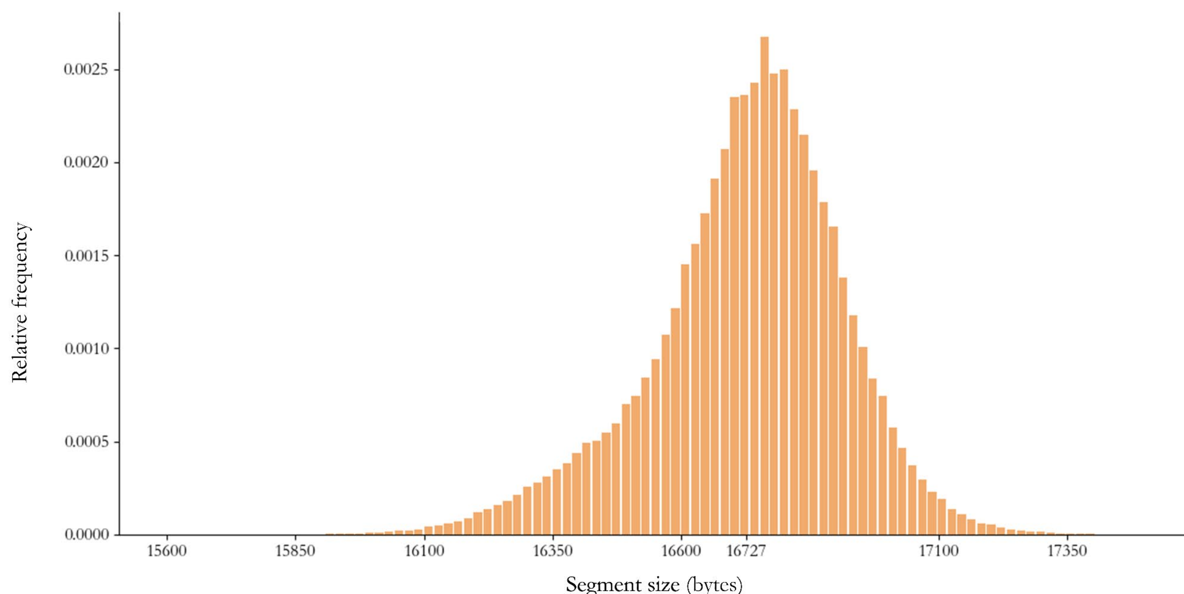


Fig. 5.27. Deterministic matching for the identification of `70244169`.

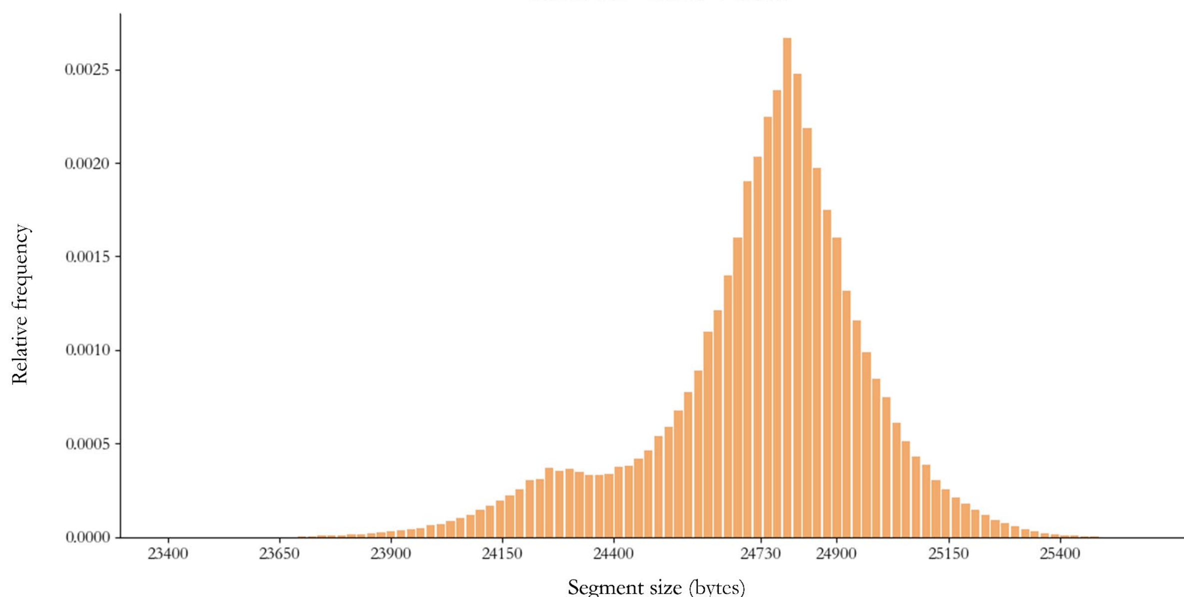
The capture used for this example is the same used for the demonstration of the ballot algorithm in Fig. 5.26 – that is why the list of captured Segments is identical. For each iteration, it is shown the JSON that mongoDB returns when querying for a Segment size. Such JSON includes the parameter `list`, which is an array listing all video IDs which have the Segment size in `_id`. For example, in the first iteration, the array to Segment size `252711` is [81030842, 81264749, 70244169, 80197874, 81043228, …]. In the second iteration, an intersection is calculated between the array that yields from the 1st iteration and the array that results after querying to mongoDB the list mapped to `922245`. This intersection only contains 2 IDs, which are the only IDs common to the first and second Segment sizes. This intersection is used by the next iteration to calculate further intersections. The process finishes when the intersection has only one ID, which corresponds to the only Netflix video having all the Segments analyzed throughout iterations. In this example, the match hits after 3 iterations.

There is one caveat worth studying for the Deterministic algorithm – together with video Segments, the client also downloads audio Segments. The PCAP analyzer cannot tell the difference between video and audio Segment requests, so it proceeds to calculate its size as expected. The distributions for audio

Segment sizes for the `heaac-2-dash` codec are plotted in Fig. 5.28 for both bitrates available: 64 kbps and 96 kbps. This is valid for all videos across Netflix; thus, the audio Segment sizes are not identificative of a Netflix video and must be discarded from the list of Segments to prevent poisoning the intersections and misleading the video identification.



(a)



(b)

Fig. 5.28. Audio Segment sizes distribution for both bitrates available for `heaac-2-dash` codec. a) 64 Kbps b) 96 Kbps

Audio Segments are always requested in batch, in groups of 2 or in groups of 16 (i.e.: for audio download, the byte range requested within HTTP requests can cover up to 16 audio Segments). Considering the audio Segment sizes distributions as Gaussian, the confidence intervals (with 99.9999% confidence level) covering the sizes of the requests for audio Segments are represented, by bitrate and by batching type, in Table 5.4:

	64 kbps	96 kbps
2 by 2 requests	31612 - 35297	47088 - 51833
16 by 16 requests	252903 - 282374	376704 - 414661

Table 5.4. Sizes for requests of batched audio Segments.

Any Segment whose size falls within any of the ranges specified in Table 5.4 will be considered an audio Segment and discarded. Of course, it can occur that a retrieved video Segment happens to have a size within those ranges, being discarded unnecessarily. This, however, is a minority case. Let Fig. 5.29 be an example of this – the distribution for the sizes of video Segments for 720p videos (H.264 codec) is represented, and the discarded ranges are represented in red. It is shown that only a minor fraction of Segments is discarded to prevent misleading calculations, which leaves a wide range of Segment sizes available in order to guarantee the operability of the Deterministic algorithm.

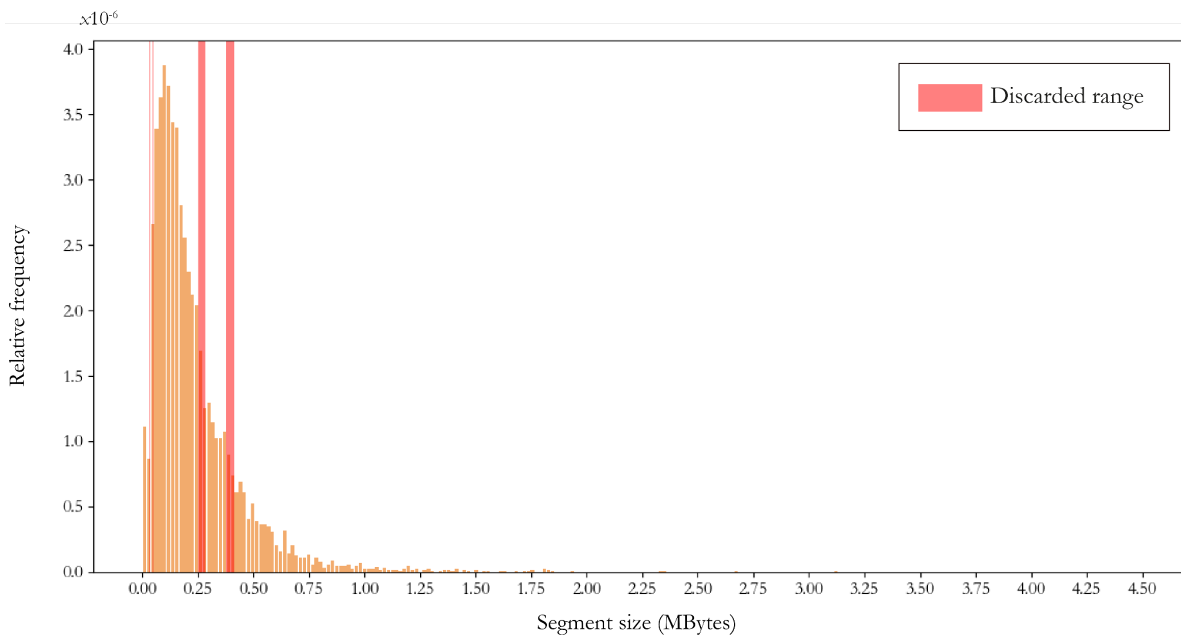


Fig. 5.29. Discarded Segment size range along the Segment size distribution for 720p videos in H.264 codec.

6. Results

6.1. Generation of the set of captures for testing

For the purpose of testing, debugging and measuring the hit rates for the developed algorithms, an extensive set of Netflix captures was required, which was generated with an automated system that we developed. This capture generation system is possible thanks to Selenium [17]: a tool which enables the automation of web browsers. Our system uses Selenium to take control of a Google Chrome instance, programmed to visualize back to back all the Netflix videos listed in a randomly generated set of IDs. All the random IDs in the set are retrieved from our databases. Programming a web browser instance with Selenium to automate Netflix playback is as easy as telling the browser to visit the URL `https://www.netflix.com/watch/[ID]`, replacing `[ID]` iteratively with each ID of the list. As the Google Chrome bot plays a video, the traffic is captured with `tcpdump` and stored on disk.

Actually, to make the capture set truly representative of the variety of Netflix catalog, the playback for each video is triggered at 4 different timestamps. The initial instant of playback can be forced within web browser clients with the `GET` parameter `'t'`, setting its value to the exact second in which we want the playback to begin. For example, visiting `https://www.netflix.com/watch/70244169?t=600` triggers the playback of `70244169` (*Code Lyoko* S1-E1) starting at the 10:00 timemark. Implementing this into our Selenium bot enables us to capture the download of 4 different Segment sequences for the same video, which will allow us to test our algorithms in a great variety of different scenarios.

With this tool, we created 2 different capture sets:

- **Capture set A**. The purpose of this set is to conduct a representative performance test for both algorithms. In this set, the captures are triggered when the Netflix client starts to fill the buffer, which is the moment of the entire playback with the highest density of Segment downloads. This allows for short captures with a high number of Segments, ideal for an intensive test of our algorithms. This set consists on 5000 captures (1250 videos captured at 4 different timestamps) which last for 15 seconds each.
- **Capture set B**. The purpose of this set is to capture the traffic generated once the initial buffer filling finishes, which is when the Segment requests slow down as they now occur only when a Segment has been popped from buffer. With this set, we will test whether the algorithms are able to identify a Netflix video based on traffic not recently started. As the Segment requests are now scattered in time, captures have to be longer to obtain a representative set of Segments. Captures start 15 seconds into playback (one the buffer is filled) and last for one minute. This set consists on 300 captures (75 videos captured at 4 different timestamps).

Throughout the explanation of the performance results for both algorithms, we will be focusing primarily on the Capture Set A, as it is the one specifically designed to measure the hit rate for both approaches. It is important to remark that the algorithms were run at a moment in development which had the Segments for 23328 videos introduced in mongoDB. The total amount of videos in Netflix catalog approximates 45000. Having indexed the Segments of more than half the catalog is enough to determine if the algorithms work with big Segments datasets.

6.2. Ballot algorithm

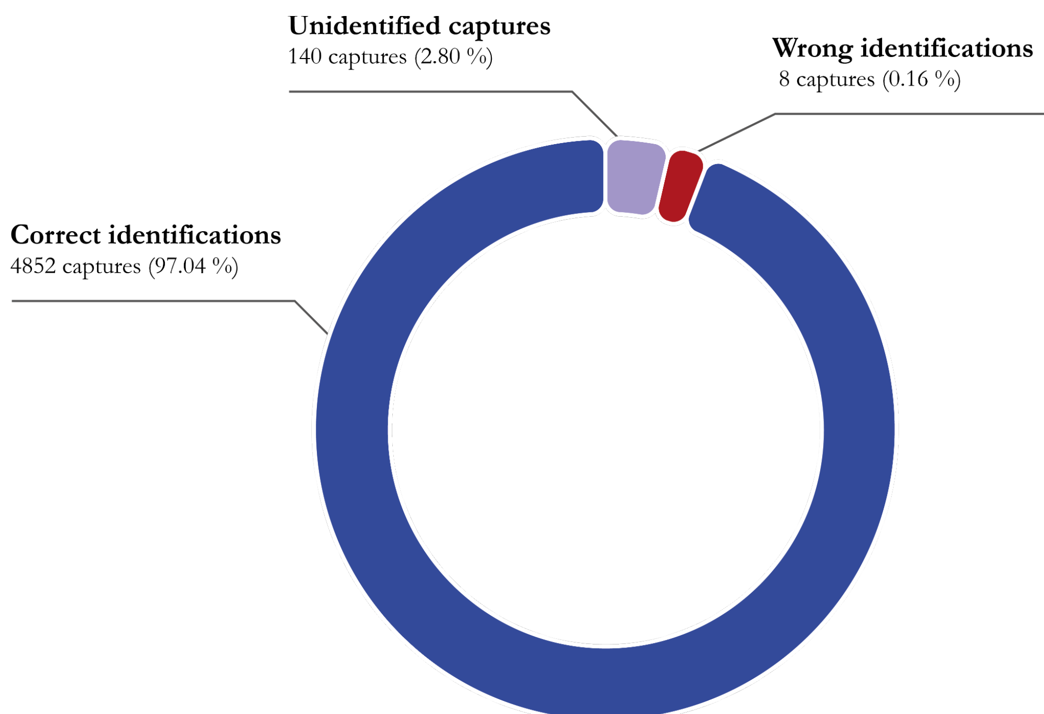


Fig. 6.1. Hit rate of the Ballot algorithm.

When testing the Ballot algorithm with the capture set A, out of 5000 captures:

- 4852 captures have been correctly identified.
- 8 captures have been wrongly identified (e.g.: the algorithm identified the traffic to belong to video `81085381` when it truly belonged to `81295950`).
- 140 captures have not been identified, which means that the algorithm has not been able to associate the traffic to any Netflix video.

The reasons as to why a capture can be identified incorrectly will be exposed later. Besides that, unidentified captures are always caused by the algorithm running out of Segments to keep the votation going before arriving to the match condition. We found 3 possible scenarios that lead to unidentified captures. Out of the 140 unidentified captures:

- 9 captures fell into what we called an **ID tie**.
- 7 captures correspond to Netflix videos which happen to be **mapped to multiple IDs**, confounding the algorithm. All these captures are gathered in Table 6.2.
- 124 captures incurred in what we called the **"Undetectable ID"** scenario.

We will now proceed to explain the previous three scenarios independently.

An **ID tie** occurs when the votation is on the right track to identify the correct video, but as iterations go by the matching criteria is never met due to the small difference in votes between the real ID and an interfering one. Let us take the capture 1 in Table 6.1 as an example of an ID tie. After 7 iterations, `81313928` has been voted 6 times while `81313924` has been voted 5 – the difference is not big enough to end up identifying the correct video (`81313928`) and the algorithm run out of Segments to keep the voting going. Notice that both IDs correspond to chapters of the same TV show, which share the same intro, causing them to share a considerable number of Segment sizes. If the capture had lasted for longer

and more Segments had been captured, the algorithm would have presumably ended up identifying the video. All the captures that incur in an ID tie are gathered in Table 6.1.

Capture num.	Real captured video	Iterations	Most voted	Second most voted
1	81313928	7	81313928 *Kongsuni and Friends (S2-E9)* *6 votes*	81313924 *Kongsuni and Friends (S2-E5)* *5 votes*
2	80174284	61	70258366 *Oh My God* *7 votes*	80174284 *Boddy Keneddy for President (S1-E1)* *7 votes*
3	80174284	67	80174284 *Boddy Keneddy for President (S1-E1)* *9 votes*	80209629 *Busted! (S1-E5)* *6 votes*
4	81000864	53	81000864 *Abducted in Plain Sight* *9 votes*	70014611 *Mobile Suit Gundam I* *5 votes*
5	81000864	51	81000864 *Abducted in Plain Sight* *7 votes*	81189912 *Drive* *6 votes*
6	81208560	8	81208532 *Kid-E-Cats (S1-E3)* *8 votes*	81208560 *Kid-E-Cats (S1-E31)* *8 votes*
7	81208539	8	81208539 *Kid-E-Cats (S1-E10)* *8 votes*	81244369 *Brothers* *5 votes*
8	81208572	8	81208572 *Kid-E-Cats (S1-E43)* *8 votes*	81208530 *Kid-E-Cats (S1-E1)* *7 votes*
9	81200812	43	81200808 *Meet the Adebanjos (S2-E16)* *5 votes*	81200812 *Meet the Adebanjos (S2-E20)* *5 votes*

Table 6.1. Captures that incurred in an ID tie.

Among all captures that fall into an ID tie, the ones with the most unexpected behavior are 2 – 5 and 9. In all of these, the number of iterations is significantly bigger than the number of votes of the most popular ID. This phenomenon is pending investigation, to determine why there are so many Segments which do not contribute to the identification of the actual video.

Having a video **mapped to multiple IDs** also unables the ballot algorithm to identify a video. For some reason, Netflix has the movie *"Wild Dog"* mapped to 4 different IDs at least: 81444683, 81444684, 81444685 and 81427455. Therefore, it is not surprising that the captures 10 – 13 fall into an ID tie, as the real ID (81444684) interferes with an equivalent ID (81444685). Furthermore, the indian movie *"Kurup"* is mapped to 2 different IDs (81555298 and 81555313), each featuring a different playback language (for this film, the multi-language feature is not handled as the rest of Netflix catalog does). Thus, it should come as no surprise that captures 14 – 16 incur in an ID tie. A solution to this scenario has not been implemented yet, but it would involve the identification of all multi-mapped videos in our databases. All the captures in Set A corresponding to videos mapped to several IDs are gathered in Table 6.2.

Capture num.	Real captured video	Iterations	Most voted	Second most voted
1	81444684	54	81444684 *Wild Dog* *54 votes*	81444685 *Wild Dog* *51 votes*
2	81444684	84	81444685 *Wild Dog* *84 votes*	81444684 *Wild Dog* *84 votes*
3	81444684	117	81444685 *Wild Dog* *117 votes*	81444684 *Wild Dog* *117 votes*
4	81444684	85	81444685 *Wild Dog* *85 votes*	81444684 *Wild Dog* *85 votes*
5	81555313	68	81555298 *Kurup (Hindi)* *68 votes*	81555313 *Kurup (Tamil)* *85 votes*
6	81555313	32	81555313 *Kurup (Tamil)* *32 votes*	81555298 *Kurup (Hindi)* *29 votes*
7	81555313	57	81555298 *Kurup (Hindi)* *57 votes*	81555313 *Kurup (Tamil)* *57 votes*

Table 6.2. Captures of videos mapped to multiple IDs.

Last, a capture incurs in a **"Undetectable ID"** scenario when, after a huge amount of iterations, the expected ID is nowhere to be seen. Table 6.3 represents one of these captures, where after 111 iterations, the most voted and second most voted IDs have obtained 16 and 15 votes respectively and neither of them corresponds to the expected ID – thus, being undetectable. This phenomenon is also pending investigation, but two plausible hypothesis could explain this behavior:

- The Segment sizes for video 80025221 may have not been introduced correctly in mongoDB.
- The sizes of the Segments within the capture were incorrectly calculated by the PCAP Analyzer.

Real captured video	Iterations	Most voted	Second most voted
80025221 *El Internado (S7-E2)*	111	70229498 *Players* *16 votes*	80236314 *Mission: Impossible – Fallout* *15 votes*

Table 6.3. Example of a capture that incurred in an ID tie.

Leaving the unidentified captures aside, the Table 6.4 collects all the captures which were incorrectly identified.

Capture num.	Real captured video	Algorithm result	Iterations	Most voted	Second most voted
1	81468690 *Mr. Robot (S2-E11)*	70225597 *Vettai*	37	70225597 *20 votes*	80104257 *15 votes*
2	81468695 *Mr. Robot (S3-E4)*	80998890 *Upstarts*	37	80998890 *19 votes*	70014611 *14 votes*
3		81016196 *Mersal*	74	81016196 *27 votes*	80188823 *23 votes*
4	81323630 *F is for Family (Season 4 Recap)*	81001887 *6 Underground*	115	81001887 *27 votes*	80166185 *27 votes*
5		81001887 *6 Underground*	76	81001887 *22 votes*	80166185 *17 votes*
6		81144150 *Sooryavanshi*	74	81144150 *20 votes*	70229046 *15 votes*
7		81001887 *6 Underground*	101	81001887 *25 votes*	70229046 *20 votes*
8	80025136 *El Internado (S6-E13)*	80017528 *Happy New Year*	126	80017528 *22 votes*	80188823 *17 votes*

Table 6.4. Incorrectly identified captures.

What all 8 incorrectly identified captures have in common is that the number of total iterations is significantly bigger than the number of votes of the most popular ID. Furthermore, neither of the most voted IDs is the expected one. These characteristics fit the description of the “Undetectable ID” scenario. In fact, all these 8 captures can be conceived as “Undetectable ID” captures which, by chance, happen to hit the matching criteria before running out of Segments.

All the results explained up to now represent the behavior of the Ballot algorithm when the difference threshold between the most voted and second most voted ID is set to 4 votes. Table 6.5 represents the hit rates when setting the threshold to other values.

Threshold	Correct identifications	Wrong identifications	Unidentified captures
2	97.06%	0.76%	2.18%
3	97.08%	0.26%	2.66%
4	97.04%	0.16%	2.80%
5	97.02%	0.14%	2.84%
6	96.94%	0.06%	3.00%

Table 6.5. Ballot algorithm hit rates for different threshold values.

As the Ballot algorithm gets more selective increasing its threshold, the wrong identifications decrease, but so tend to do the correct identifications. The ratio of unidentified captures increases with the threshold. The threshold was set to 4 during the testing of the algorithm, offering a reasonable trade-off between all three parameters.

6.3. Deterministic algorithm

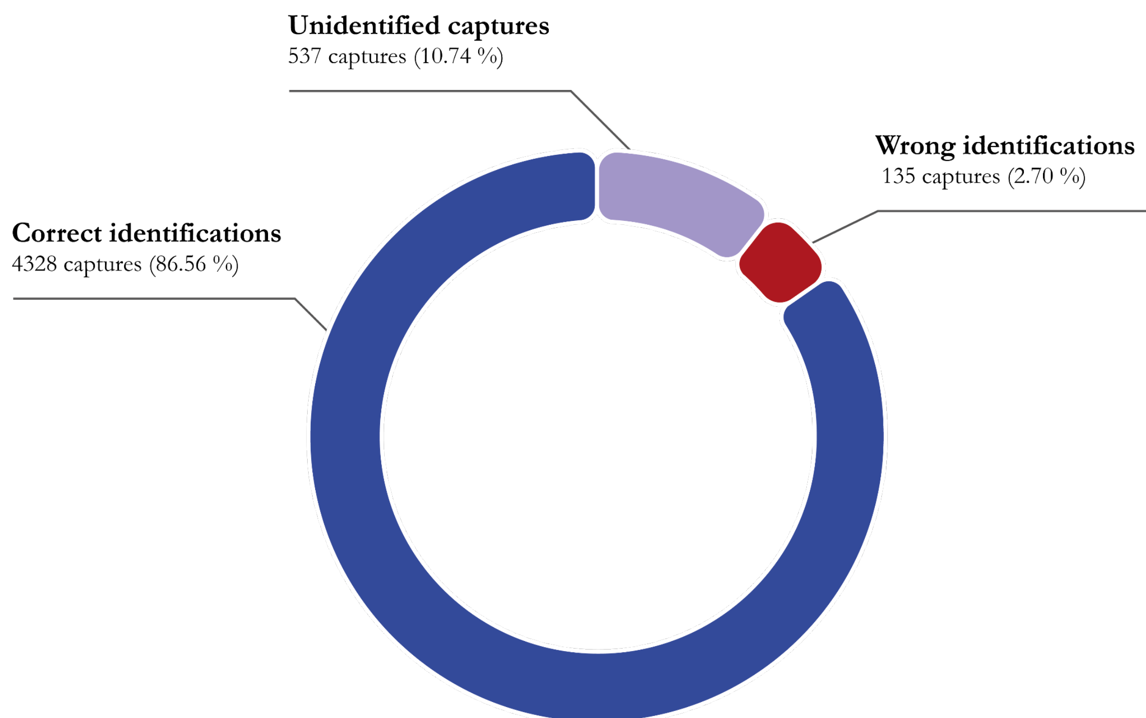


Fig. 6.2. Hit rate of the Deterministic algorithm.

When testing the Deterministic algorithm with the capture set A, out of 5000 captures:

- 4328 captures have been correctly identified.
- 135 captures have been wrongly identified.
- 537 captures have not been identified.

It is important to remember that this algorithm calculates the intersection between successive ID lists until only one remains. Such ID, by definition, is the only one common to all Segment sizes analyzed among all iterations. The hit rate that this algorithm should theoretically achieve is 100%. Having 135 wrongly identified captures and 537 unidentified captures when using a deterministic algorithm can only mean that:

- either the PCAP analyzer calculated wrongly the size of at least one Segment in the capture,
- or the PCAP analyzer started a Segment size calculation over a HTTP request which in reality did not correspond to a Segment request.

Both scenarios involve misleading Segment sizes which poison the intersections calculated by this algorithm, corrupting the identifications as a result. The source of this problem lies on the PCAP analyzer – further investigation and debugging is required to improve the deterministic algorithm to obtain the 100% hit rate that it should achieve on paper.

6.4. Comparison of algorithms

The Table 6.6 compares several statistical metrics between the ballot and deterministic algorithms:

Ballot algorithm	Metric	Deterministic algorithm
97.04%	**Hit rate** Ratio of correctly identified captures relative to the total.	86.56%
≤ 100%	**Maximum theoretical hit rate** Maximum hit rate that the algorithm should achieve theoretically once all the PCAP analyzer bugs are fixed.	100%
90.33%	**Exclusive hits** Ratio of correct identifications achieved exclusively by each algorithm, relative to the total amount of unidentified or misidentified captures from the other algorithm.	72.86%

Table 6.6. Comparison of hit rate statistics between both algorithms.

While on paper the deterministic algorithm should obtain a 100% hit rate, the tests performed demonstrate that this is not the case because, as of right now, the intersections can be poisoned by misleading Segment sizes. The ballot algorithm is more resistant to those misleading Segments and, in fact, it obtains a hit rate very close to 100%.

Furthermore, with the current state of development of both algorithms, the ballot approach was able to correctly identify 607 captures out of the 672 captures (90.33%) that the deterministic approach was not able to tackle (those being 537 unidentified + 135 misidentified captures). On the other hand, the deterministic algorithm was able to identify 102 captures out of 148 (72.86%) that the ballot algorithm could not (those being 140 unidentified + 8 misidentified captures). Each algorithm has a great potential to tackle the captures that result problematic to the other algorithm. Therefore, a third algorithm could be developed as the merge of the ballot and the deterministic approach, to solve each capture by the most effective method. However, still the only way to obtain a 100% hit rate would be to address the issues related to misleading Segments.

6.5. Behavior towards *in medias res* captures

We call "*In medias res* captures" to all the captures in the Set B, which were triggered once the buffer was filled, from when the Segments start to be requested only once another Segment is popped from buffer. These captures are less convenient to test the algorithms quickly, but get closer to a real capture environment, in which a capture can be triggered with the Netflix traffic already started. The hit rates for both algorithms analyzing Capture Set B are represented in Fig. 6.3.

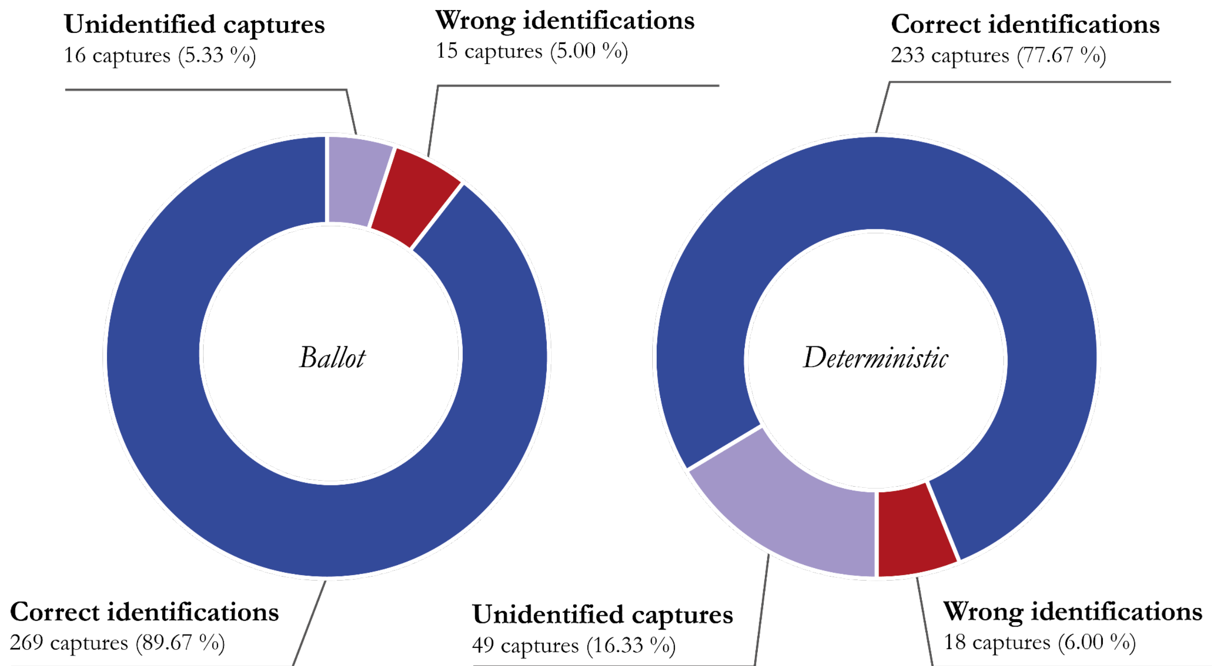


Fig. 6.3. Hit rates for both algorithms for the *"in medias res"* capture set.

The hit rates demonstrate that video identification is still possible for both algorithms despite capturing already started traffic. The pie charts depict that the ratios of correct identifications have worsened for both algorithms when comparing them with the results obtained with Capture Set A. This, however, is not a problem – the statistics demonstrate that video identification is still possible for *in medias res* captures. These ratios would tend to the ratios obtained with Set A if the *in medias res* captures were held for longer than a minute, but the important milestone is that the general operation of the algorithms is still possible when capturing already started traffic.

7. Conclusions

We have demonstrated that the identification of Netflix videos via the characterization of its encrypted traffic is still feasible as of 2022. In fact, we have developed a solution for video identification which improves the previous existing work in several aspects.

The fundamental concept on which our algorithms are based is the accurate calculation of the sizes of downloaded Segments, instead of estimating them with sliding windows. Despite introducing a minor percentage of miscalculations, this deterministic approach for the obtention of Segment sizes made possible the identification of videos with few Segments, which speeds up the identification process to a few seconds.

Our work also improves the state of the art by implementing a much faster method to extract an up-to-date snapshot of Netflix catalog. This is done by using Netflix's internal API (`shakti`) to obtain the products deterministically, instead of having to scrap Netflix front-end. Furthermore, we managed to use another Netflix API (via `pymsl` library) to gain access to sizes of Segments much faster than having to download them by visualizing each video for a few seconds. Our method also leads to a higher level of customization, as we can specify which profiles we want to download the Segments from.

All these led to the development of Netflix identification algorithms capable of successfully identifying a video based on its encrypted traffic. With a set of 5000 captures and 23328 videos registered in the databases, we obtained a practical 97.04% success rate for our algorithms – although a theoretical 100% success rate can be achieved with further development.

8. Future work

We have implemented an efficient algorithm for the identification of Netflix encrypted content, but there are some extra features pending development which would enhance the operability of our programs. The most remarkable pending features are:

- Improving the Deterministic algorithm until 100% hit rate. This task involves:
    - Investigation as to why "Undetectable ID" captures (6.2.) exist.
    - Identification of Netflix videos associated with multiple IDs (6.2.).
    - Registering in our databases the Segments for all profiles available across all Netflix clients (5.2.1.1.).
- Improving the Segment size calculator (5.1.4.) to prevent calculations coming to a halt when a TCP packet with TLS header is missing.
- Enabling our algorithms to work in live capture environments with multiple Netflix videos being played simultaneously.

References

[1] A. Reed, B. Klimkowski. "Leaky Streams: Identifying Variable Bitrate DASH Videos Streamed over Encrypted 802.11n Connections". 13th IEEE Annual Consumer Communications & Networking Conference (CCNC), 2016.

[2] A. Reed, M. Kranch. Identifying HTTPS-Protected Netflix Videos in Real-Time. Dept. of Electrical Engineering and Computer Science. United States Military Academy at West Point, 2017.

[3] MPEG. Information technology — Dynamic adaptive streaming over HTTP (DASH) — Part 1: Media presentation description and segment formats (ISO/IEC 23009-1:2019), 2019.

[4] Traci Ruether. Streaming Protocols: Everything You Need to Know. Wowza Media Systems. 2021. https://www.wowza.com/blog/streaming-protocols, visited January 2022.

[5] Ron Garrison. Structure of a MPEG-DASH MPD. OTTVerse. 2021. https://ottverse.com/structure-of-an-mpeg-dash-mpd/, visited January 2022.

[6] C. Barnatt [ExplainingComputers]. *Explaining Digital Video: Formats, Codecs & Containers* [Video]. YouTube. https://youtu.be/-4NXxY4maYc, visited January 2022.

[7] MPEG. Information technology — Coding of audio-visual objects — Part 12: ISO base media file format (ISO/IEC 14496-12:2020), 2020.

[8] Netflix. Open Connect. https://openconnect.netflix.com/, visited January 2022.

[9] Netflix. MSL. https://github.com/Netflix/msl, visited January 2022.

[10] Ilya Grigorik. Transport Layer Security (TLS). High Performance Browser Networking. 2013. https://hpbn.co/transport-layer-security-tls/, visited January 2022.

[11] Michael Driscoll. The New Illustrated TLS Connection. 2019. https://tls13.ulfheim.net/, visited January 2022.

[12] truedread. pymsl. https://github.com/truedread/pymsl, visited January 2022.

[13] MongoDB. https://docs.mongodb.com/, visited January 2022.

[14] oldgalileo. shakti. https://github.com/oldgalileo/shakti, visited January 2022.

[15] List of Netflix Categories. What's on Netflix. https://www.whats-on-netflix.com/library/categories/, visited January 2022.

[16] Microsoft. Silverlight for Media. https://www.microsoft.com/silverlight/media/Default, visited January 2022.

[17] The Selenium Browser Automation Project. https://www.selenium.dev/documentation/, visited February 2022.

[18] Dean Takahashi. Comcast: Pandemic drove peak internet traffic up 32% in 2020. VentureBeat. https://venturebeat.com/2021/03/02/comcast-peak-internet-traffic-rose-32-in-pandemic-in-2020/, visited February 2022.

[19] Netflix. Protecting Netflix Viewing Privacy at Scale. The Netflix Tech Blog. https://netflixtechblog.com/protecting-netflix-viewing-privacy-at-scale-39c675d88f45, visited February 2022.

Appendix

A. Intuitive introduction to adaptive pseudo-streaming.

Multiple paradigms exist when it comes to content delivery. Let us take an example of a video that wants to be consumed by an end user. Let the size of that video be 3 GB. Let us also suppose that the server delivering such video sends that particular content to the client at 250 Mbps. The client will receive its desired content at, approximately, 250 Mbps.

The first paradigm treats the video being delivered as a single binary file. As such, the video cannot be decoded and played on destination until the whole binary stream is fully downloaded on the client. A 3 GB video at a 250 Mbps delivery rate would take around 100 seconds to be fully delivered to the end host. Thus, the content playback cannot start until around two minutes have passed at least.

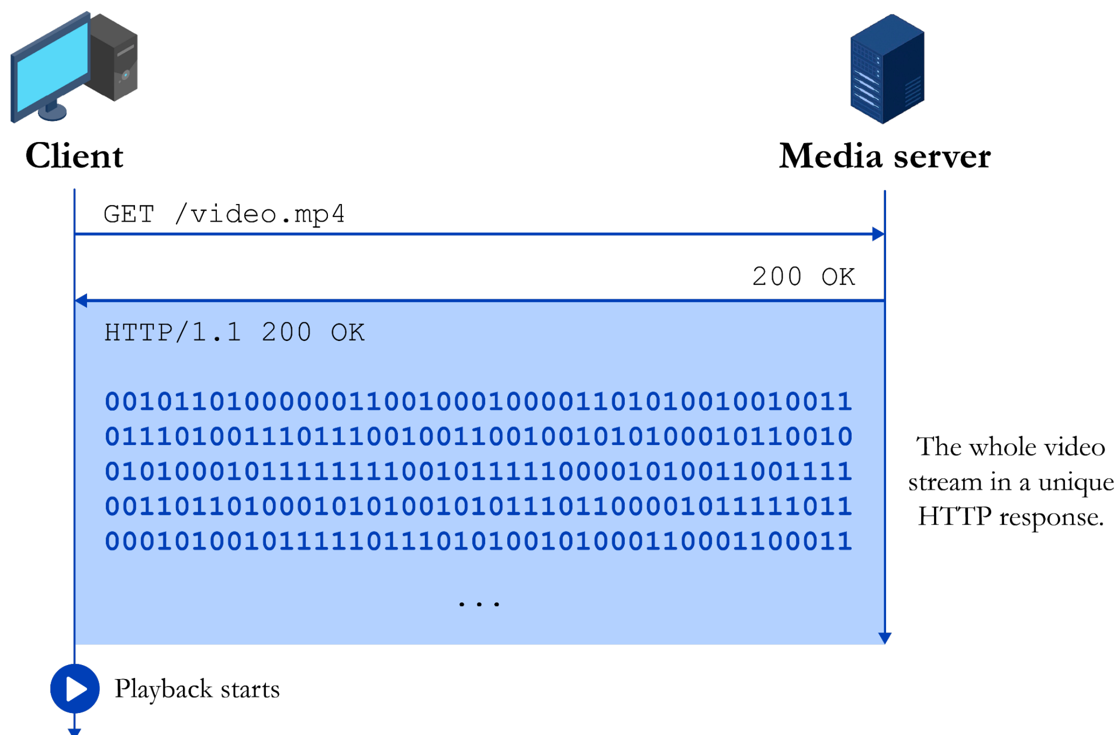


Fig. A.1. Content delivery based on direct download.

This approach has several technical limitations, but the biggest one is that playback cannot start until the content is completely downloaded.

Desirably, users would like to start the playback of content as fast as possible. As direct download cannot solve this issue, pseudo-streaming arises as the natural evolution to full-download in content distribution.

Pseudo-streaming is based on dividing the content that wants to be delivered into small, accessible chunks that can be requested individually as the content playback takes place.

Let us now suppose that the 3 GB video is split into 100 MB chunks, resulting in 30 divisions. As each video chunk is now its own unique file and can be requested individually, content playback can start as

soon as the first chunk is fully downloaded. An uninterrupted and smooth playback experience will take place if the client downloads the subsequent chunks as it plays the already downloaded ones.

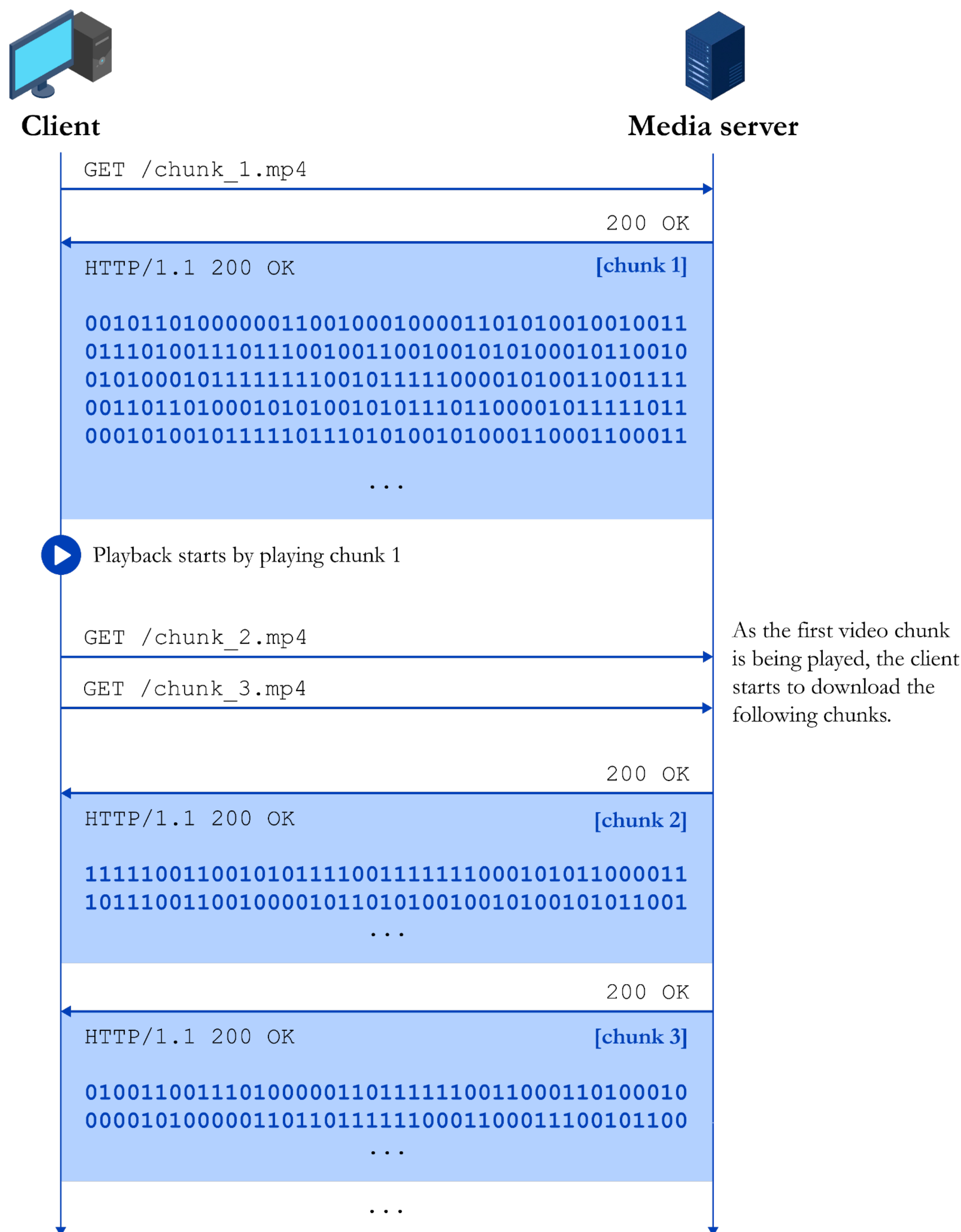


Fig. A.2. Content delivery via pseudo-streaming.

As the first chunk's size is 100 MB, and the server delivers it at a 250 Mbps rate, the client will take around 3.2 seconds to download it and to start playback, instead of two minutes. In order to guarantee a smooth experience, the second chunk must already be downloaded when the first chunk's playback finishes. Otherwise, a playback interruption will happen, as depicted in Fig. A.3.

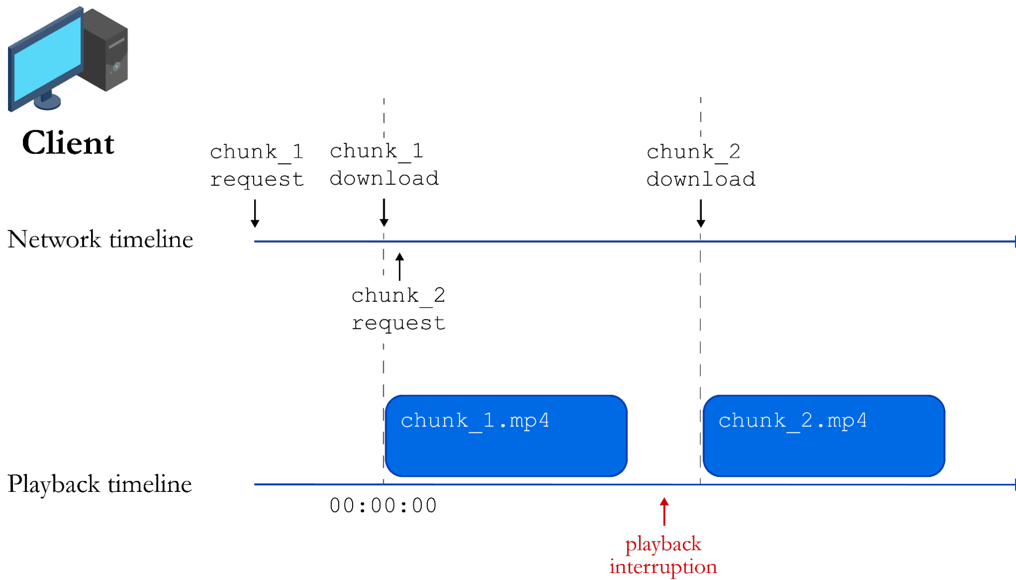


Fig. A.3. Playback interruption due to the lack of buffered chunks.

Clients implement a buffer which stores several adjacent chunks before starting playback. Playback starts when the buffer is sufficiently filled. As buffered chunks are consumed, subsequent chunks are requested to keep filling the buffer. Thus, interruptions never happen as long as the buffer keeps being fed. This behavior is represented in Fig. A.4.

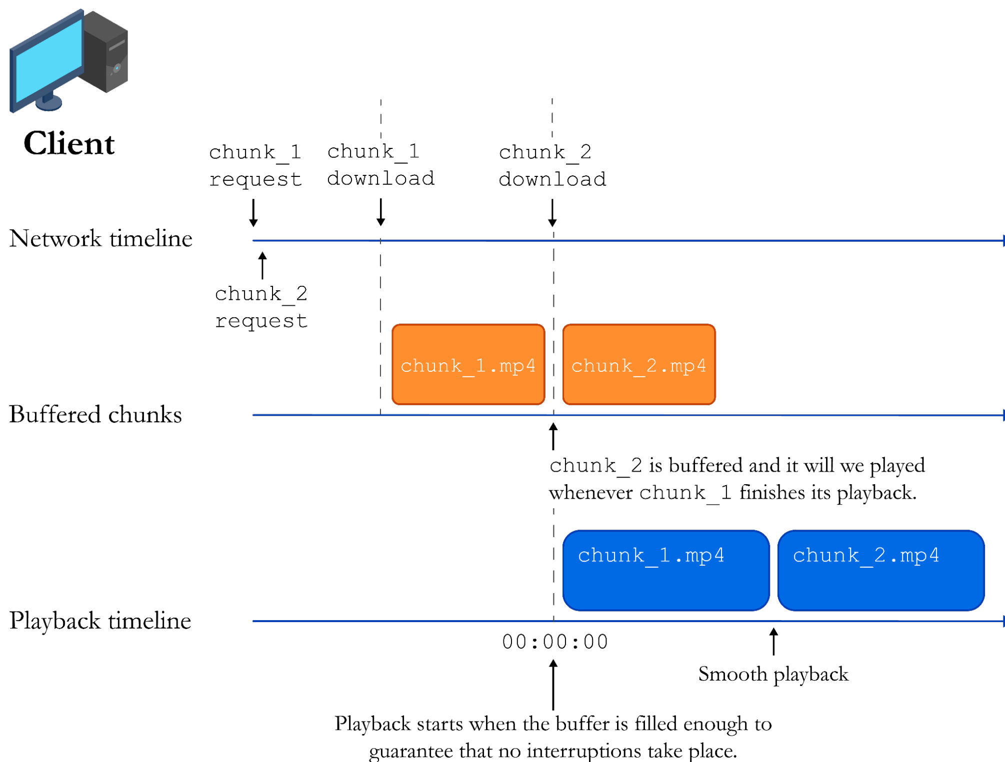


Fig. A.4. Smooth playback thanks to buffered chunks.

Pseudo-streaming overcomes the technical limitations that direct download has. To get started, playback can start without having the whole content downloaded, which allows for a much more immediate playback experience. Also, pseudo-streaming leads to a more efficient memory usage.

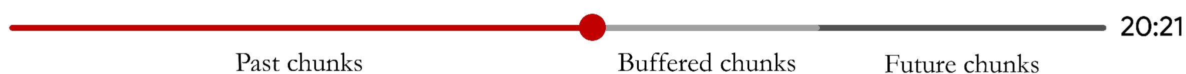


Fig. A.5. Playback bar for pseudo-streamed content.

While direct download would require to store the entirety of the content on a hard drive, pseudo-streamed content only requires to keep on RAM the buffered chunks (Fig. A.5). Previous chunks are popped from RAM whenever they are consumed (thus, they free up resources) and future chunks will be requested whenever they are needed (they do not use resources if they are not going to be consumed in the short term). Direct download content makes use of hard drive resources, even when the content is not being played.

Furthermore, letting the client decide when to request chunks allows it to react to changing network conditions. To illustrate this point, let us consider a video whose chunk composition is depicted in Fig. A.6.

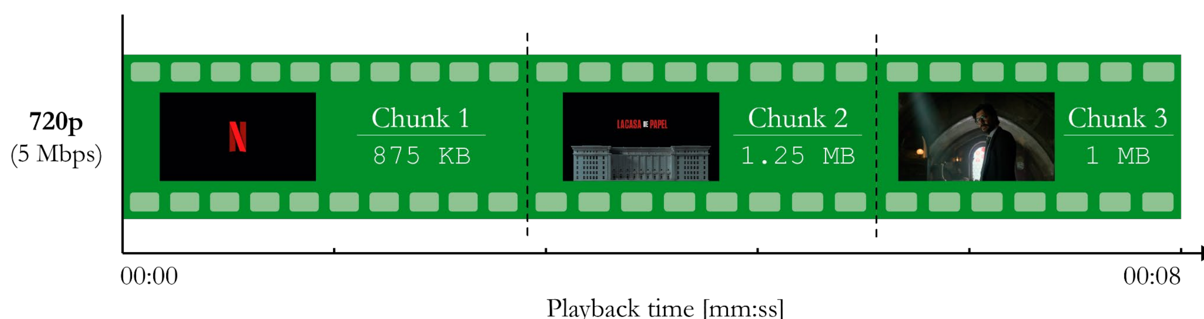


Fig. A.6. Chunk composition for a *Money Heist* episode displayed in 720p.

The figure represents the first 8 seconds of an episode of *Money Heist* with a 720p resolution. Those first seconds are made up of 3 chunks. The size of each chunk is represented within each subdivision (i.e.: the size of the first chunk is 875 KB). The target minimum bitrate for optimal playback is 5 Mbps. If the user receives the chunks at a bitrate lower than 5 Mbps, interruptions will occur.

This interruption behavior is outlined in Fig. A.7. The first two chunks are downloaded when the client's available bandwidth is 5 Mbps. Playback starts when 2 chunks are available in buffer. When the second chunk download finishes, the client's available bandwidth drops from 5 Mbps to 1 Mbps, which causes the third chunk download to last for almost 6 seconds. This download takes so long that the third chunk is not available when the second chunk has finished its playback, which leads to an interruption.

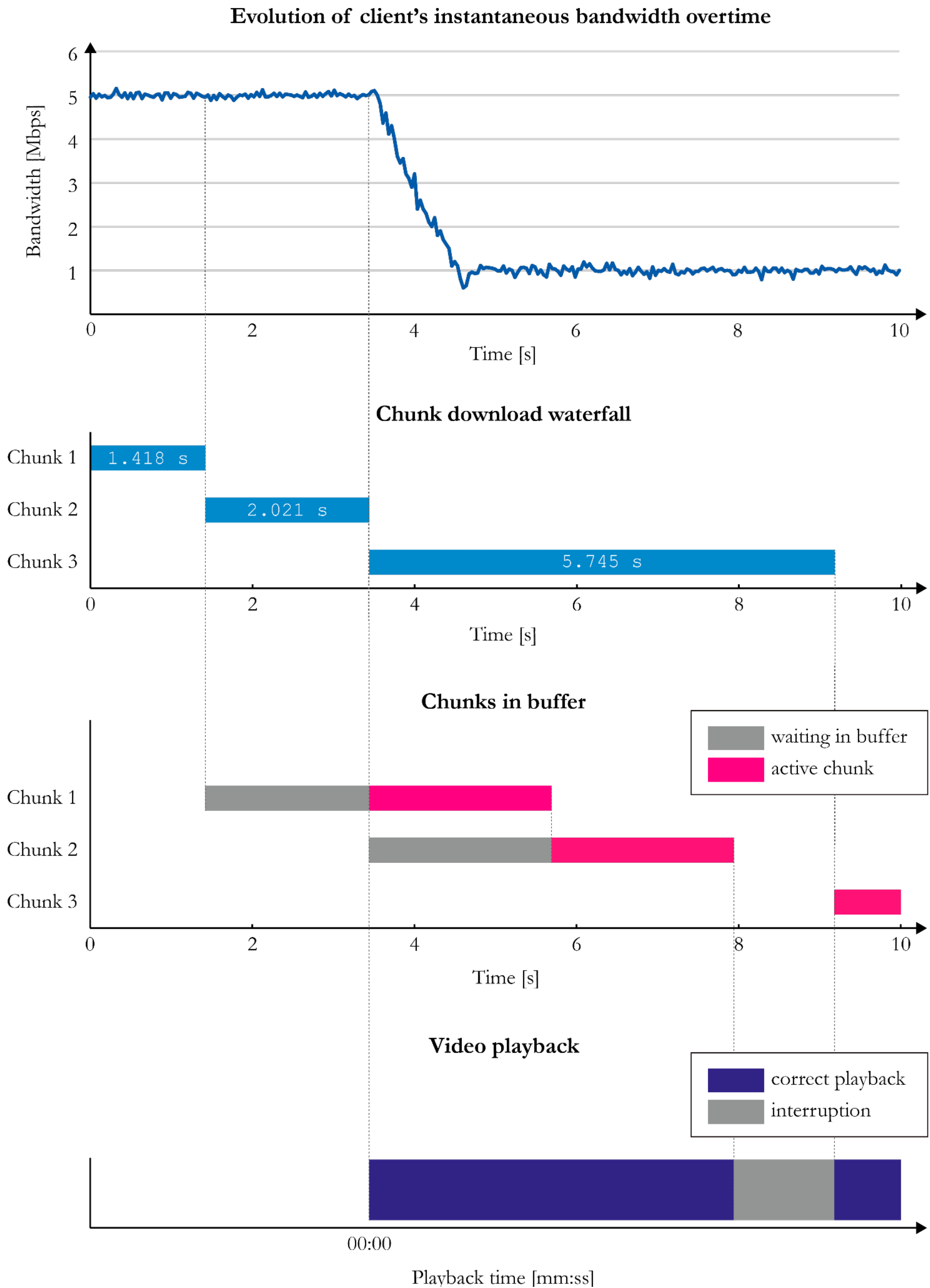


Fig. A.7. Playback interruption due to client's bandwidth decrease.

This interruption could have been avoided if the download of the last chunk lasted shorter in order to be introduced in buffer on time. To make such download faster, the only solution would have been to

download a chunk smaller in size. Let us now consider the same content with the same chunk composition but with 3 alternative resolutions (Fig. A.8).

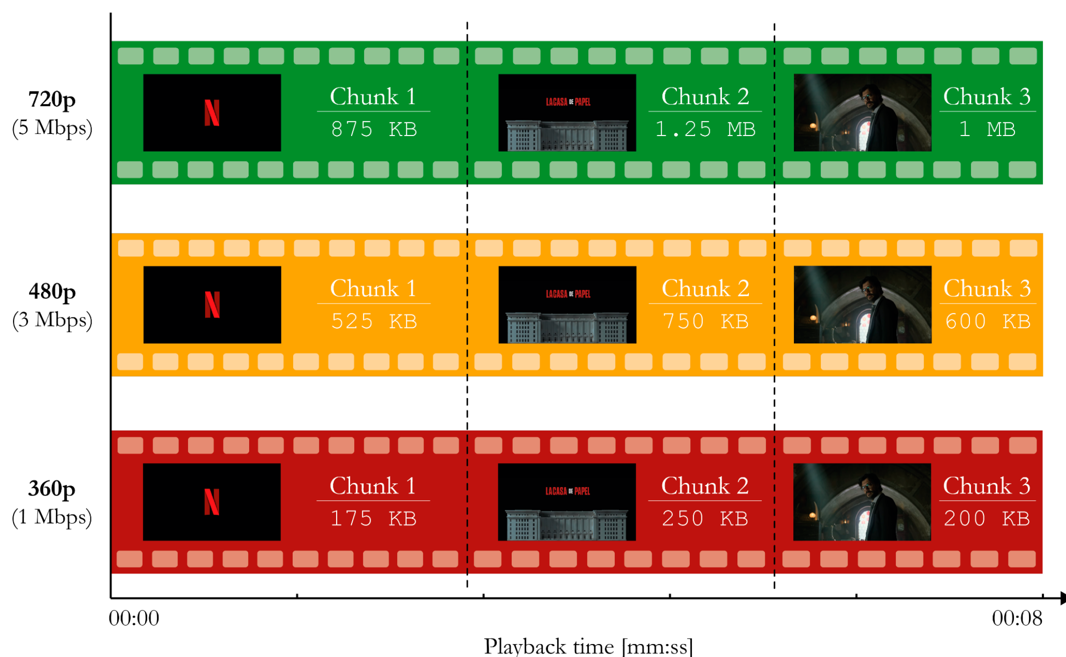


Fig. A.8. Chunk composition for a *Money Heist* episode for multiple resolutions.

In the same way that the client has total control over when to request new chunks, it does also have control over which kind of chunks to request. If the client detects the bandwidth drop, it can request a lower resolution chunk to accomodate to the worsened network conditions.

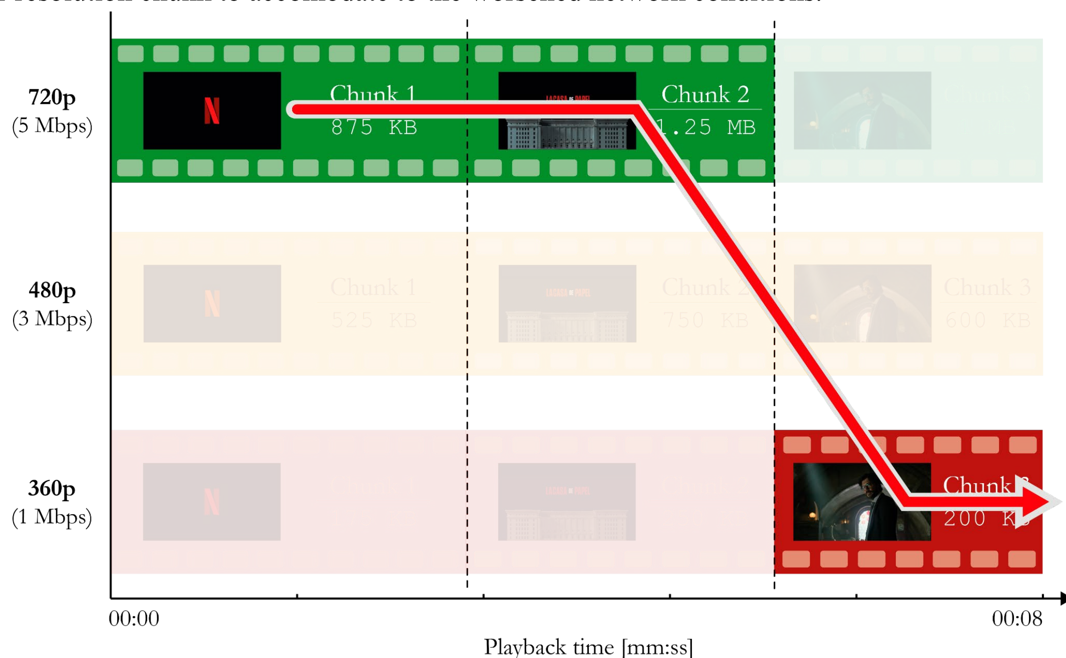


Fig. A.9. Chunk request sequence to accomodate to worsened bandwidth.

If the client requested the chunks as portrayed in Fig A.9, no interruption would occur (Fig A.10).

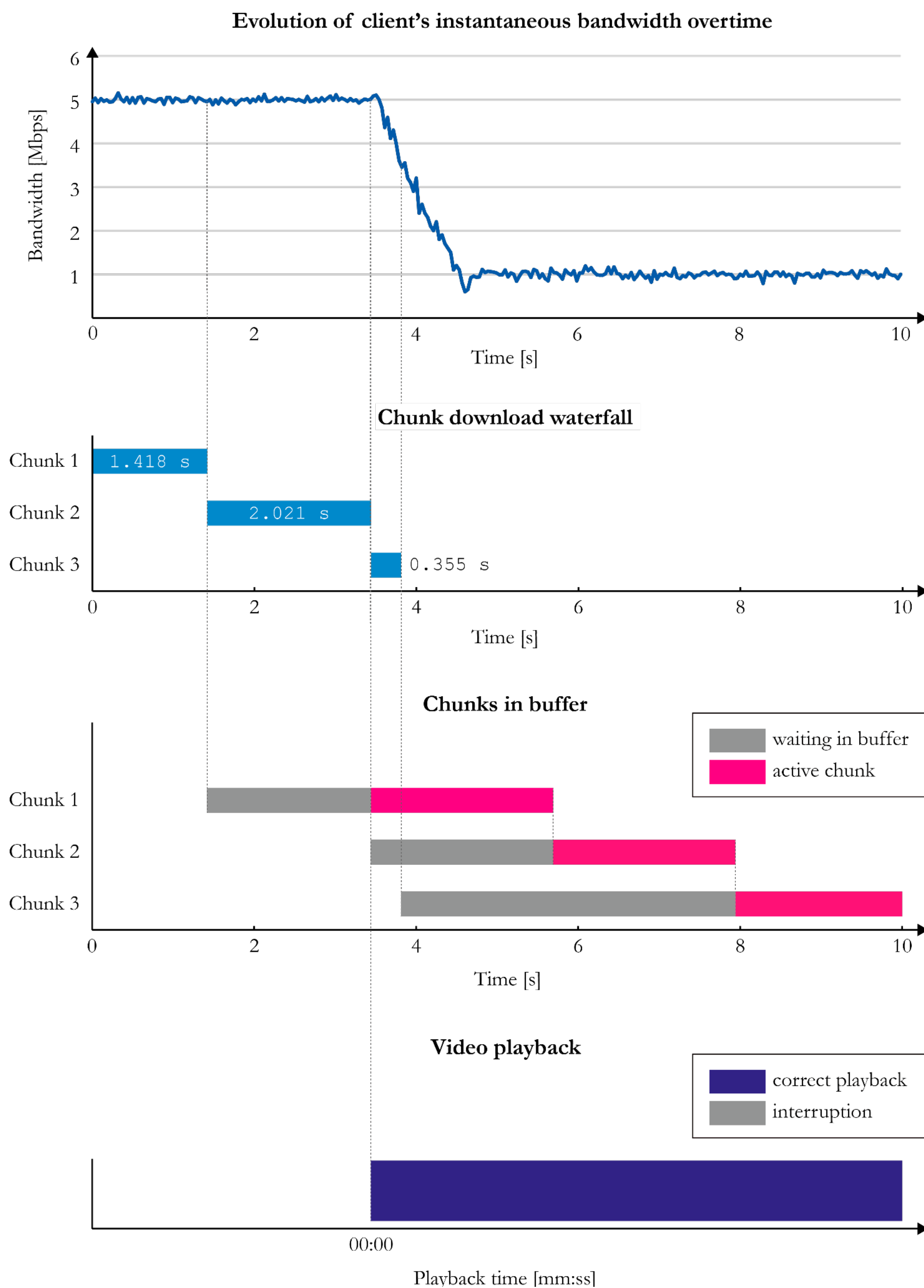


Fig. A.10. Uninterrupted playback using multiple video resolutions.

Overall, pseudo-streaming offers great advantages compared to direct download:

- Immediate playback without having the whole content downloaded.
- Content does not need to be stored on hard drive.
- Adaptability to changing network conditions.

The client has total control over chunk requests by deciding their quality and when to download them.

B. Digital media storage. Containers.

The Segments that any pseudo-streaming protocol delivers are, ultimately, video, audio or subtitle chunks. All this media is stored in servers. When it comes to digital media storage [6], three concepts must be clear: codecs, containers and formats.

A *codec* is a program or hardware able to both encode and decode a digital stream. Video and audio encoding is generally performed for the sake of compression. The difference from codec to codec lies in the algorithm used to compress the raw media signal and in the syntax used to store the information. Different algorithms lead to varying compression rates. As technology has evolved, codecs have become more efficient, achieving greater compression rates without significant drops in perceived quality. For example, some of the most popular video codecs are H.264 (a.k.a. AVC), H.265 (a.k.a. HEVC) and H.262 (a.k.a. MPEG-2); and indisputably, the most well-known image codecs to the general public are JPEG, PNG and GIF.

A *container* allows multiple data streams to be introduced into a single file. Containers define how all individual streams are multiplexed. The difference between containers lies in the multiplexing technique and in the metadata introduced for further detailing those multiplexed streams.

The use of containers is very common in digital video storage. For example, if `video.mp4` is a video that a user has stored on a hard drive, the user expects both video and audio to play (and sometimes even subtitles) when opening such file. Clearly, `video.mp4` is a container which wraps different media streams into a single file. Some of the most remarkable video containers to the general public are AVI, MP4 or QuickTime File Format.

In another vein, the use of containers is not applicable to images, as there are no supplementary streams required for the delivery and rendering of pictures.

The *format* represents the binary syntax used to store the information. The format used by a particular file is generally denoted by its extension. For example, the document `chapter_1.pdf` has `.pdf` as its extension, and this lets a computer know that such file must be processed by a PDF viewer. This program understands the binary format used by the PDF file to store the information and, therefore, such information can be retrieved by the program to be displayed to the user.

In the field of images, the format is directly related to the codec used. For instance, the file `photo.jpeg` is the binary stream that outputs when encoding a raw image using the JPEG codec (ignoring metadata). The extension `.jpeg` tells the computer that such file must be processed by a program that implements the JPEG codec, in order to decode the information and render the image on screen.

However, in the scope of videos, the extension is not related to the codecs used. For example, the file `video.mp4` is the binary stream that outputs when multiplexing several encoded media streams in accordance to the syntax defined by the MP4 container. The extension `.mp4` tells the computer that such file uses a MP4 container to combine multiple media streams, but it tells nothing about the codecs used. Any program able to interpret a MP4 container will see that `video.mp4` wraps several encoded streams. The container itself includes the metadata that specifies the codec required to decode each stream for playback. This makes it clear that codecs and containers work independently: a container can wrap multiple streams encoded with a wide range of codecs.