Department of Information Engineering and Computer Science

Master's Degree in
Computer Science

Final Dissertation

Improving low-latency adaptive bitrate live video streaming over HTTP/3

Supervisor

Fabrizio Granelli

[illegible]

Student

Matteo Contrini

Academic year 2021/2022

Contents

Abstract

Video streaming over the Internet has become increasingly popular in recent years and now accounts for more than 50% of the total traffic. One of the main challenges video developers have to tackle is providing a user experience that is as smooth as possible. In live video streaming, low-latency in particular, the problem is especially challenging and requires finding a trade-off between reducing the probability of video freezes (rebuffering) and keeping a high video quality.

In this thesis, we first develop a testbed to evaluate the performance of adaptive bitrate video streaming technologies. We then run some emulations and experiment with real-world network datasets to analyze how existing solutions behave. We find that some configurations show suboptimal behavior and propose some improvements based on that. The main proposed solution, which takes advantage of HTTP/3 features and modern browser APIs such as WebCodecs and WebAssembly, shows that it is possible to greatly reduce playback stalls and to provide a smooth playback experience with stable live latency, as the experimental results show.

Chapter 1

Introduction

During the last decade, the usage of **online video streaming** has exploded. From sports to entertainment, the market has seen strong growth in the past few years and video streaming now accounts for more than half of Internet traffic.[13]

Live events streaming in particular is becoming increasingly relevant, with a 15-fold growth in just five years, according to statistics. This is in part due to the effect of the pandemic, through which many organizations started to live stream events and kept doing it, and in part due to a more general shift from traditional broadcasting means such as DVB-T (digital terrestrial), cable TV and satellite to Internet streaming. A special category of live streaming is **low-latency streaming**, as we will see, which is particularly relevant for use cases such as sporting events, gaming, and live news coverage.

In this chapter, we will briefly introduce the context of our work, namely, how modern video streaming works and what the main challenges are. We will then present how these challenges are usually tackled and why we believe that a different point of view could be needed to provide a satisfying user experience to viewers. Finally, we will introduce our proposal that we believe improves the live streaming quality of experience.

1.1 Live video streaming and latency

Video streaming is typically divided into two main categories: on-demand and live. With **video on demand** (VoD), we typically mean prerecorded content that can be watched by the user at any time, while **live** refers to events that are produced and transmitted as they happen.

When considering the live streaming scenario, **latency** is an important factor. For example, in sporting events viewers expect to see the action as soon as possible without delays, especially if they are able to compare the live stream with other broadcasting means such as digital terrestrial.

According to Wowza, a leading company in the streaming industry, a live stream can be considered **low latency** if it has a latency of less than 5 seconds.[1] However, latency can be defined in many ways. A typical definition is the glass-to-glass (or end-to-end) latency, computed as the time it takes for a video frame captured by a camera (the "glass") to be shown on the screen of the user (the other "glass"). Other times, the latency is measured from the moment the video frame is *ingested*, which in practice means the moment when it is fed as input to the video transcoder.

1.2 Network constraints

Video streaming over the public Internet faces the problem of **slow or congested networks**. Bandwidth (or throughput) could in fact be very low or oscillating, creating situations where there is not enough bandwidth to play the video stream smoothly. This is particularly challenging with live video streaming, where we cannot rely on the fact that the video is already available and therefore it cannot be buffered in advance by the player.

Although modern compression standards achieve significant compression ratios, leading to bitrates usually below 10 Mbps even for high-resolution content at good quality, not all networks are always able to provide such bitrates reliably.

[1] `https://www.wowza.com/low-latency`

In fact, while modern access networks such as fiber optics and 4G/5G can be quite reliable and provide a high throughput, Internet connectivity is still far from being perfect. As an example, very high capacity networks are still not dominant in most countries. Mobile networks coverage can be problematic too, and Wi-Fi connectivity can be unstable due to interferences and limitations of the wireless protocols.

An additional source of issues that is often overlooked is the Internet Service Provider network, whose backhauling infrastructure needs to be sized properly to avoid congestion. Interconnections with CDNs and other providers are also very important, as they can become congested during peak hours if capacity upgrades are not properly planned in advance.

These issues led to the development of a number of video streaming technologies, among which **Adaptive Bitrate Streaming** protocols based on HTTP are nowadays considered the de facto standard in the industry.[4]

1.3 Adaptive bitrate technologies

The core idea of **Adaptive Bitrate Streaming** (ABR) is to have video content encoded in multiple renditions, or variants, with different resolutions and bitrates. The set of video renditions and their parameters constitute a *bitrate ladder*. Video players use different techniques to understand which rendition is the best for the current device and network conditions, and thus *adapt* accordingly. The aim is to provide uninterrupted playback at the best possible quality.

The two most common protocols for adaptive streaming are **Apple HLS** (HTTP Live Streaming) and **MPEG-DASH** (Dynamic Adaptive Streaming over HTTP). As the names suggest, both work over HTTP and are conceptually similar.

They require video content to be encoded in short video segments (no more than a few seconds) so that players can switch rendition at segment boundaries. Bitrate adaptation algorithms should find a trade-off between reducing the probability of video freezes (rebuffering) and keeping a high video quality level, but neither HLS or DASH mandate how players should choose the bitrate for the next segment. Multiple approaches are possible, as we will see in more detail in the following chapters.

An important benefit of using HTTP for video streaming is that it makes it easy for streaming platforms to scale, thanks to the widespread availability of **Content Delivery Networks** (CDN) and their global distribution networks.

1.4 Challenges in live video streaming

Both HLS and DASH support **live streams**. They do so by making new video segments available as soon as they are encoded.

Delivering live streams introduces a new category of challenges for video players, which need to find a trade-off between stream reliability and lower latency. In fact, lowering latency usually means keeping a smaller buffer of media data on the client, with the disadvantage that rate adaptation algorithms might not be able to react timely to network fluctuations. If video segments cannot be loaded in time, this can cause rebuffering.

1.5 A different live user experience

When a live video streaming setup struggles to adapt to network conditions, the most noticeable effect is **rebuffering**. The playback gets stuck while the player tries to complete the download of the next video segment, and the **latency tends to grow**. As a consequence, playback will lag behind the live event and users will receive the action late, possibly by even tens of seconds.

This kind of experience is **different from other broadcasting mechanisms**. For example, digital terrestrial and satellite do not have the concept of buffering: if the video signal is not good enough, video and/or audio are temporarily unavailable while the signal recovers, but no latency is introduced, i.e. viewers always see the live action with a fixed delay independently from the signal quality.

Adaptive streaming research has been mostly focused on avoiding empty buffers as much as possible, although in low-latency scenarios completely avoiding rebuffering is a very hard task. Instead, there are methods that can be integrated in an adaptive streaming setup so that the linear TV experience that viewers are already used to can also be replicated in video streaming. Although subjective, it would likely result in a better Quality of Experience (QoE).

1.6 Contributions and results

In this work, we first develop a **testbed** for experimenting with HTTP-based live streaming protocols. We then use the testbed to evaluate the performance of existing live streaming protocols such as HLS and DASH with different configurations and HTTP versions. One of the things we observe is that streaming over **HTTP/3**, a recent version of HTTP based on the QUIC transport protocol, shows an unexpected and suboptimal behavior in terms of prioritization of requests. This is especially evident in comparison to HTTP/1.1 and HTTP/2.

Among other improvements, we therefore introduce a modified version of the `HLS.js` library that takes advantage of HTTP/3 features to **override the default prioritization strategy** of HTTP responses. By giving priority to audio segment requests, we get closer to a user experience where the lack of video buffer does not cause playback stalls.

The next step is to implement a strategy we call **segment dropping**: when a video segment is needed for playback but is still being loaded, we interrupt the request and thus reset the underlying QUIC stream, freeing bandwidth that would otherwise be used to download a video segment that is already outdated.

The dropped segment needs to be replaced with a placeholder that we call **filler segment**. This segment is dynamic and generated on the fly in the frontend, taking advantage of the WebCodecs API and WebAssembly.

The experimental results show that this solution almost eliminates video playback stalls, providing smooth playback of the audio track even when the video could not be loaded. In this way, even when the video rate adaptation algorithm is slow to react to new network conditions, a smooth user experience is preserved.

1.7 Thesis structure

This thesis is organized as follows. In **Chapter 2**, we introduce the main concepts and technologies surrounding video compression and streaming over HTTP. In **Chapter 3**, we introduce the testbed setup that we developed to evaluate the performance of streaming protocols. In **Chapter 4**, we show the results of the analysis of some experiment runs that show suboptimal behavior. In **Chapter 5**, we propose some improvements to the live streaming user experience and show the results of the implementation of a modified version of HLS. Finally, in **Chapter 6** we acknowledge some limitations of our work and mention a few ideas that are left for future work.

Chapter 2

Background

In this section, we will introduce the main concepts and technologies on which video streaming, and therefore our work, is based. We will start with video compression, going through important concepts such as *Group of Pictures* (GOP), also presenting some popular video codec formats. We will then delve into video container formats, introducing MPEG-2 Transport Stream, MP4 and Fragmented MP4. We will also present how adaptive bitrate streaming works and the main technologies, such as Apple HLS and MPEG-DASH, and go through the evolution of HTTP versions (HTTP/3 in particular). Finally, we will briefly introduce the main network emulation tools.

2.1 Video compression concepts

Video is known to be one of the heaviest types of content to be stored and transmitted. Since treating uncompressed video is unfeasible, video compression or video coding formats have been developed and standardized over the years, with multiple implementations (**codecs**) being released. The objective of video codecs is to limit the output data rate, measured in bits per second, while trying to maintain a perceptually good video quality.

The process of compressing a video is known as **encoding** and is carried out by an encoder. When a video file is already encoded in a video coding format and needs to be re-encoded in another one, we often use the term **transcoding**.

In this section, we will go through the main techniques that are used by most video coding standards.

2.1.1 RGB vs Y'CbCr

The most basic form of compression can be obtained by converting individual pictures representing the video to a **color space** that better exploits the characteristics of human vision, and then applying compression of some of the components in the new color space.

A very common color space family used in digital video and images is **Y'CbCr**, sometimes improperly called YUV (which refers to the analog domain). This color space takes advantage of the fact that the human vision system is much more sensitive to light variations (brightness) than to color. The Y'CbCr color space decomposes the color information of a pixel into three components:

- **Y'**: the luma[2], representing the brightness of the image;
- **Cb**: the blue chroma component, representing a projection of the blue color;
- **Cr**: the red chroma component, representing a projection of the red color.

This approach is different from the common RGB (Red Green Blue) color space, where the luma is not isolated from the color components, and allows to apply compression more effectively through the **chroma subsampling** technique. Chrome subsampling refers to the practice of reducing the resolution of the chroma components while keeping the luma at full resolution. Since our eyes are

[2]Sometimes called luminance, although luma is the correct term for digital video. Luma is the non-linear gamma-corrected representation of luminance.

less sensitive to color than to brightness, we can reduce color resolution by even 75% with almost no perceptual impact on quality.

The vast majority of digital video that can be found on the Internet or transmitted through digital television is compressed with the **Y'CbCr 4:2:0** color space, the most common format for non-professional content. In Y'CbCr 4:2:0, the luma component (Y') is encoded at full resolution, while the chroma components are stored at 1/4 of the resolution, that is, instead of storing 8 chroma samples for every 8 pixels, we only keep 2, as shown in Figure 2.1.

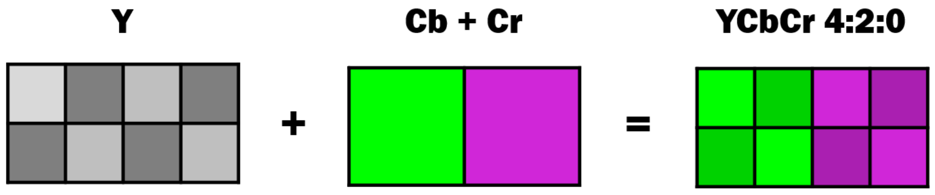


Figure 2.1: 4:2:0 chroma subsampling. The 4 refers the width of the grid, which has a (fixed) height of 2. The 2 in the notation specifies that the horizontal resolution is halved (resulting in two "horizontal" samples), while the 0 that there are no different samples between the first and the second row.

Compared to a typical 8-bit RGB image (equivalent to Y'CbCr 4:4:4, i.e. no subsampling), requiring 24 bits per pixel, a Y'CbCr 4:2:0 image requires only 12 bits per pixel, resulting in a 50% saving with very similar perceived quality. When using tools like `ffmpeg`, Y'CbCr 4:2:0 is often called `yuv420p` or similar.

2.1.2 Inter-frame and intra-frame compression

All the main video coding standards released since the early 1990s are based on a **hybrid codec model** that exploits both temporal and spatial redundancy of videos to achieve high compression ratios.

Temporal compression, or **inter-frame compression**, relies on the fact that there is usually a high similarity between consecutive video frames. On the other hand, spatial compression, or **intra-frame compression**, takes advantage of the fact that pixels that are close to each other within a picture are usually highly correlated.

Video **codecs** (en**co**der/**dec**oder) are implementations of video coding standards that are able to convert the input video into a coded version in a way that is reversible, i.e. such that the decoder can reconstruct the original video with some approximation. Encoders should output a compressed representation that is as efficient as possible while trying to preserve the fidelity of the original video.

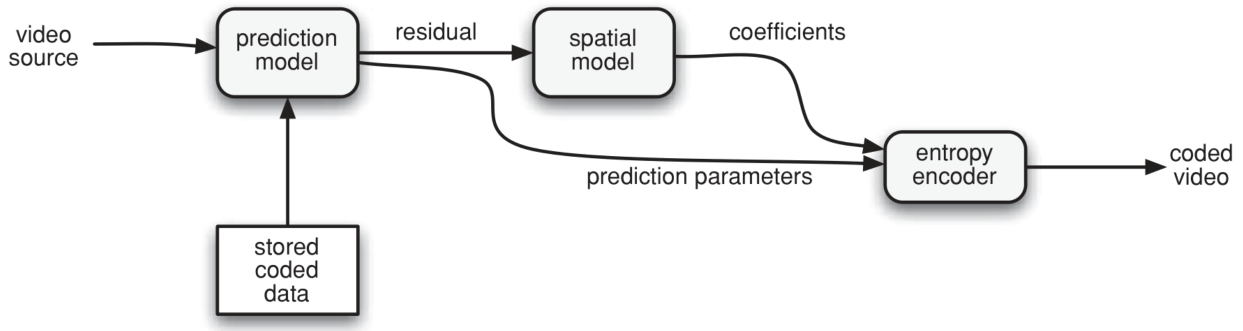


Figure 2.2: High-level hybrid codec architecture.[26]

In general, a hybrid video codec works in three phases:[26]

- **Prediction model**: exploits spatial or temporal redundancy by producing a prediction of the current frame (or block of a frame) being analyzed. This is done by looking at a previous reference frame or at other parts of the same frame. The outputs of this phase are a **residual frame**, which is the difference between the actual frame and the predicted frame, and the parameters that define how the prediction was obtained. By encoding only the residual and the parameters we can basically store only the "error" of the prediction and greatly reduce the amount of data that needs to be coded. The prediction can be obtained in two ways:

  - **Temporal prediction**: in its most basic form, the prediction is the difference between the current frame and a previous (or future) reference frame. In practice, we also need to take into account the motion that occurs between frames by running a **motion estimation** algorithm that for each block[3] in the current frame finds the best matching block in the reference frame. The **motion-compensated block** becomes the prediction used to calculate the residual, which becomes the output of the prediction phase together with the **motion vectors** that explain how the prediction block was obtained (i.e. how it "moved" with respect to the reference frame).
  - **Spatial prediction**: when encoding a block of the image, a prediction of the pixel values of the block is first calculated by looking at neighboring pixels. Statistically, pixels close to each other are expected to be similar due to spatial redundancy. Usually, this step consists of looking at the pixels on the left and/or top edges of the block. The predicted block is then subtracted from the current block to obtain a **residual block**, which is passed on to the next phase together with the information that tells how the prediction of the block was obtained.

- **Spatial model**: in most codecs, this phase consists of compressing the residual frame through a transform and quantization step, followed by the encoding of the obtained coefficients.
  - The **transform** step often consists in applying the **Discrete Cosine Transform** (DCT) to transform the blocks from the space to the frequency domain. The output of the DCT is a matrix of coefficients, which can be used to faithfully reconstruct the original block, although without achieving any compression.
  - In the **quantization** step, coefficients that have insignificant impact, such as values that are close to zero, are discarded, enabling to represent the original block with some approximation by storing a smaller number of DCT coefficients (the non-discarded ones). In the decoder, the subset of coefficients can then be fed into the **IDCT** (the inverse of the transform), obtaining a reconstruction of the original block. The fidelity of the reconstruction depends on how strong the quantization step was, that is, in practice, on the **quantization parameter** (QP).
  - After quantization, the remaining coefficients are reordered through a **zigzag scan** of the matrix, so that the most significant coefficients will be at the beginning of the sequence (a typical property of the DCT), and then encoded through **Run-Level Encoding** (RLE).

- **Entropy coding**: in this phase, all the information collected in previous phases, including quantized coefficients, quantization parameters and motion vectors, is encoded in a bit stream. To exploit the statistical redundancy of symbols in the data, techniques like **Variable-Length Coding** (VLC), including Huffman coding, or arithmetic coding and **Context-aware Arithmetic Encoding** (CAE) are used, achieving further compression.

The output of a hybrid video encoder is a *bit stream* (or *bitstream*), namely a compressed sequence of coded residual coefficients and other parameters. The decoder applies the same process inversely to reconstruct the original video frames, with some approximation.

2.1.3 Frame types and GOP

As we have seen, the prediction model can be based on temporal (inter-frame) prediction or spatial (intra-frame) prediction. The type of prediction and which reference frames are used for the prediction determine the type of frame. In particular:

[3]Blocks are small regions of pixels.

- **I-frames** (*Intra-frames*) are frames that do not require any other frames to be decoded and are thus compressed only by intra-frame prediction.
- **P-frames** (*Predicted frames*) are frames that can reference other previous frames when performing temporal prediction. Depending on the codec, a frame can reference one or more previous frames.
- **B-frames** (*Bi-directional predicted frames*) are frames that can reference both previous and future frames as reference frames in temporal prediction. They are more computationally expensive to encode, but are usually the most compressible ones.

Because of inter-frame dependencies, the display order of frames is very often different from the decoding order. For this reason, in codecs jargon there is the distinction between **Presentation Time Stamps** (PTS), and **Decode Time Stamps** (DTS).

Depending on the type of frames that are used to encode a video sequence, different **prediction structures** can be obtained. For example, low delay applications could use a structure like the one in Figure 2.4, where only I-frames and P-frames are used and P-frames always reference the previous frame. I-frames should be inserted into the stream periodically to allow more efficient random access (*seeking*). Another reason is that in some cases scene cuts might justify the use of an I-frame instead of a P-frame, depending on how drastic the scene change is.

Most of the time, the arrangement of the frames is much more complex and is usually defined by a **Group of Pictures (GOP)** structure, as seen in Figure 2.5. A GOP always starts with an I-frame and is most of the time *closed*, which means that it is independent from previous and future GOPs. As a consequence, in a closed-GOP scenario the decoder does not need access to previous of future GOPs to be able to decode frames in the current GOP.

The structure of a GOP can be summarized as a string sequence, like `IBBPBBPBBPBBI`, or through two numbers, `M` and `N`, that respectively determine the distance between P-frames, and the GOP size. For example, the structure in Figure 2.5 can be expressed as `M=3, N=12`.

2.1.4 Popular video coding formats

The dominant video coding standard is currently **H.264**, sometimes referred to as *Advanced Video Coding* (AVC) or *MPEG-4 Part 10*, developed by a joint team of the ITU (Video Coding Experts Group) and ISO (Moving Picture Experts Group, or **MPEG**). H.264 was first standardized in 2003; nevertheless, it is estimated that H.264 is still used today by more than 80% of the companies in the industry, thanks to its good performance (both in terms of compression and encoding/decoding efficiency) and the accessible royalties structure.[4]

It is worth noting that the H.264 standard only defines the format and syntax of the H.264 bitstream and how to decode it, but it does not specify how to actually encode a video. Therefore, the term

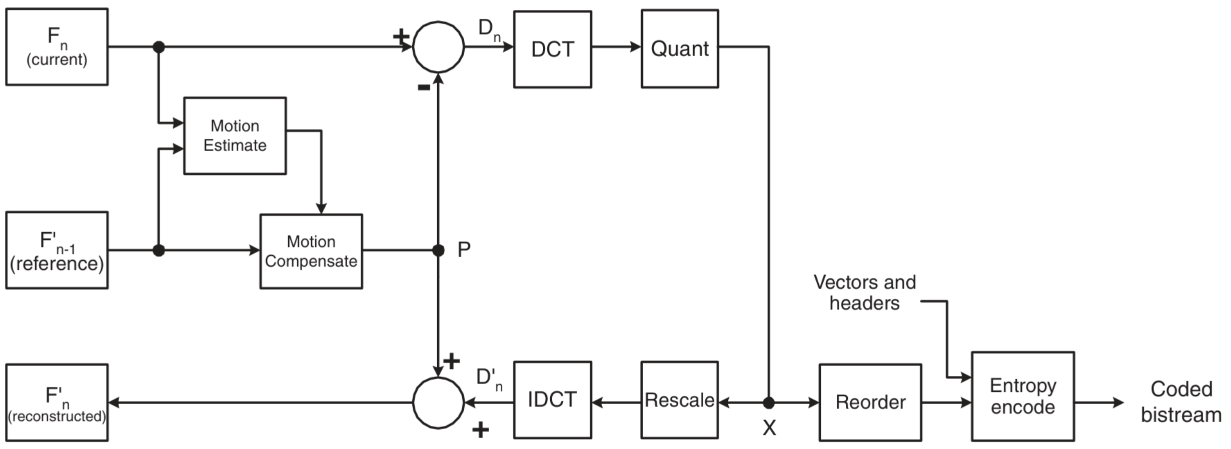


Figure 2.3: Detailed hybrid codec architecture.[26]

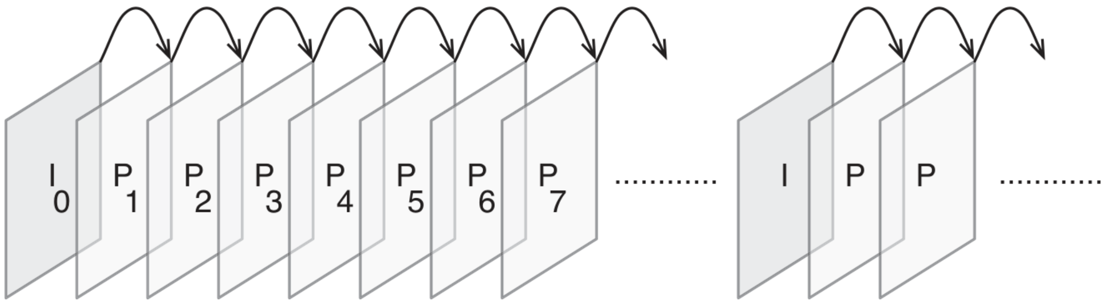


Figure 2.4: Simple GOP structure with no B-frames.[26]

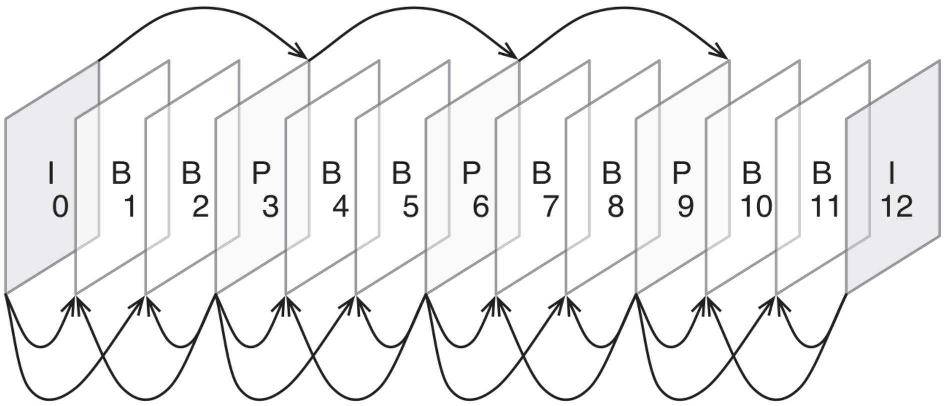


Figure 2.5: Typical GOP structure, `M=3, N=12`.[26]

codec should only be used to refer to the actual implementations of H.264, which typically include both an encoder and a decoder.

The successor of H.264, first published in 2013, is **H.265**, also known as *High Efficiency Video Coding* (HEVC) or *MPEG-H Part 2*, which provides 25% to 50% better compression at the same bitrate in comparison to H.264. It is especially suitable for high-resolution content such as 4K UHD, but it struggled to reach wide adoption due to the complex and expensive royalties structure that slowed down hardware support.[22]

In competition with the H.26x family of formats, Google released royalty-free **VP8** in 2008, followed by **VP9** in 2012. Thanks to Google controlling a large fraction of the browser market share and the YouTube streaming platform, VP9 became a popular alternative to H.264, often delivering better video compression ratios with comparable quality.[4]

The successor to VP9 was incorporated into **AV1** (first released in 2018), the royalty-free video format developed by the *Alliance for Open Media* (AOMedia), an initiative backed by large companies such as Google, Apple, Meta, Microsoft, Amazon, Netflix, Cisco, NVIDIA and Intel, among others.[4] AV1 is much more complex than H.264 and achieves up to 50% better compression when compared to H.265, making it especially suitable for high-resolution content such as 4K UHD or 8K UHD. The downside of AV1 is that since it is quite complex, codec implementations are often prohibitively expensive to run in software, thus requiring hardware support. However, having widespread hardware support is an effort that can take quite a few years.[18]

H.264

As we have seen, H.264 is still the dominant video format in the industry. Since the standard covers only the decoding part of the codec, as shown in Figure 2.6, many codec implementations have been released, achieving different compression results. One of the most popular H.264 encoders is `x264`,

[4]`https://aomedia.org/membership/members/`

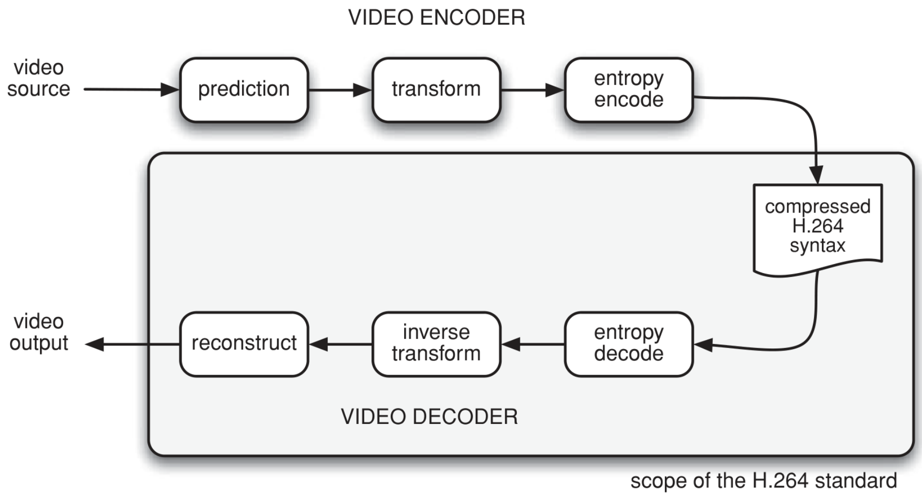


Figure 2.6: The scope of the H.264 standard.[26]

which is open source and software-based, typically scoring as the best speed/quality trade-off in H.264 codec comparisons.[6] `x264` is also the default codec in `ffmpeg`, a very popular tool for video and audio manipulation and compression.

H.264 is based on the hybrid codec model we have introduced earlier. In H.264, video frames are divided into 16x16 pixels **macroblocks** (MB), on which the prediction model is applied. Macroblocks can be split into partitions that can be as small as 4x4 pixels (not necessarily square), so that the same macroblock can reference different macroblocks, possibly in different reference frames.

In H.264, the transform is an approximation of the DCT and can be applied on 4x4 or 8x8 blocks. Quantization can be controlled by the QP parameter, or step size, which ranges from 0 to 51 and is usually adjusted automatically by the encoder depending on the input configuration. Finally, for the entropy coding step, H.264 supports both the variable-length coding (VLC) and arithmetic coding (CABAC) techniques.

As not every device is capable of supporting all the features of the standard, H.264 includes the concept of profiles and levels. A **profile** defines the H.264 features the decoder must support to be able to decode the compressed video. Common profiles are: the baseline profile, which does not include support for B-slices[5] or CABAC; the main profile, which supports both B-slices and CABAC; the high profile, which includes additional optimizations such as adaptive selection of the block size for the transform step. The **level** instead specifies an upper limit on frame size, decoding rate, and memory required to decode the video.

Finally, an important part of H.264 is the syntax of the coded bitstream. The H.264 data is organized into a sequence of packets known as **Network Abstraction Layer Units** (NAL units or **NALUs**). Since units can be of varying length, there should be a way to distinguish when a unit ends and the next one begins. There are mainly two approaches to solve this problem:

- Transporting NAL units by wrapping them in **packets**, which could be network packets or a structure defined by the **container format**, as we will see in the following sections.

- Treating the coded bitstream as a **byte stream**. In this case, a *start code*, a 3-byte sequence that acts as a synchronization marker, is inserted before each NAL unit so that the decoder can identify the boundaries of the units. This byte stream format is defined by the *Annex B* of the standard, which is the reason why this format is very often referred to as `annexb`.

[5]In H.264 the concept of I-frames, P-frames and B-frames is replaced by the slices equivalent. A slice is a subset of macroblocks contained in a video frame.

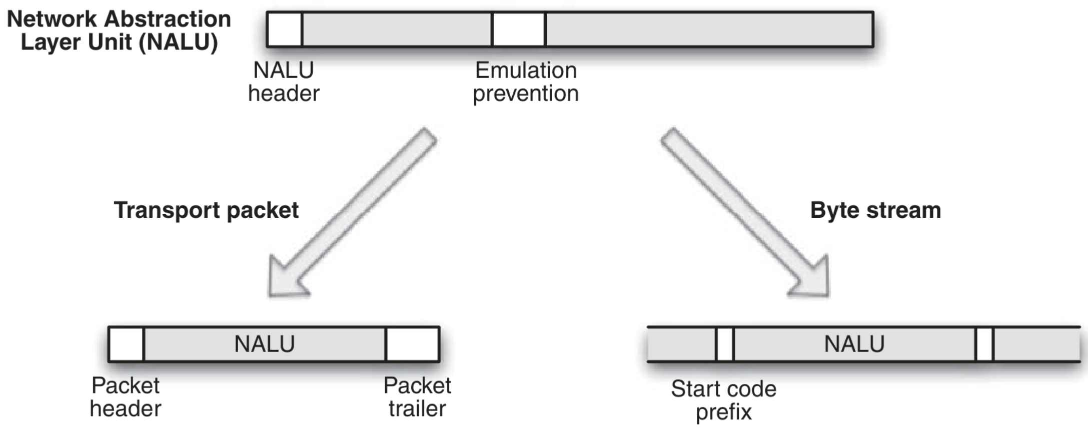


Figure 2.7: H.264 bitstream structure and encapsulation alternatives.[26]

2.2 Digital audio and compression

Digital audio is generally represented through the **Pulse Code Modulation** (PCM) method, which is characterized by two main parameters: **sample rate**, which defines how frequently the sound level was measured when captured from the analog domain, and **bit depth**, which refers to the number of bits used to store a sample.

Although uncompressed PCM audio is more tractable than video, since it is much lighter (a stereo audio track with 44.1 kHz sample rate and 16-bit samples requires “only” 1.4 Mbps), audio is typically highly compressible since human sound perception is limited. Audio coding formats such as **MP3** and **AAC** are based on perceptual coding, as they tend to discard sound information that would otherwise be inaudible and can achieve excellent quality with savings of up to 90%.[12]

AAC is nowadays the most widely used audio format in streaming scenarios and is supported by virtually every device.[4] AAC comes in multiple variants, with **Low Complexity** (AAC-LC) being the most common, as it is widely supported and provides good compression ratios and quality. Other common versions are those of the **High Efficiency** (HE) family, which are optimized for low-bitrate applications.

In the AAC bitstream, audio samples are organized into packets that contain a fixed number of samples, typically 1024. To reduce compression artifacts, AAC uses a modified version of the DCT transform that works with overlapping samples. This means that, in practice, for each AAC packet encoded or decoded another packet with the same number of samples is required. For this reason, AAC encoders add at least 1024 silence samples before the first actual audio sample, a technique called **priming**. Since this delay could introduce synchronization problems when audio is combined with video, decoders need to detect priming and correctly take into account the encoder delay.[1]

2.3 Container formats

Bitstreams produced by video and audio encoders often do not contain enough information to allow video players to actually play the video or audio file. For example, in H.264 the timing information is optional and without it the decoder does not know how to assign timestamps to individual video frames, or even determine the duration of the video.[2] In AAC, the raw frames do not contain information such as the sample rate or the variant of AAC used to encode the samples.[12]

Container formats solve this problem by wrapping the bitstreams in a structure that is common between different coding formats. Container formats also allow the grouping of multiple video, audio and subtitle tracks, an operation known as **muxing** (from *multiplexing*) and performed by **muxers**. The inverse process of extracting tracks from a container format is instead known as **demuxing**.

Two popular container formats used in the context of video streaming are **MPEG-2 Transport Stream** and **MP4/Fragmented MP4**.

2.3.1 MPEG-2 Transport Stream

MPEG-2 Transport Stream (or **MPEG-2 TS**), defined by the MPEG-2 Part 1 standard, is a container format originally designed for digital television broadcasting systems. For this reason, it includes features such as error correction and synchronization, providing protection against degraded transmission channels.

In MPEG-2 TS, a transport stream is composed of multiple programs or services, which could correspond, for example, to different TV channels. Each program can consist of multiple video and audio tracks, known as **elementary streams**, multiplexed to compose the transport stream. In each elementary stream, the raw coded data is packetized into packets known as PES packets (Packetized Elementary Stream), each having a fixed length of 188 bytes. The length of the packet includes headers, which transport data such as timing information. Hence, Transport Stream often has a non-negligible overhead.[19]

An H.264 bitstream muxed in an MPEG-2 stream must be in the Annex B format (i.e. with start codes before NAL units).

2.3.2 MP4 and fragmented MP4

MP4 is a container file format based on the **ISO Base Media File Format** (ISOBMFF). It was extended from Apple QuickTime File Format (`.mov`) and first published in 2001 as MPEG-4 Part 14.[6][5]

In the **ISOBMFF** format the video and audio bitstreams, such as an H.264-compressed video, are stored as tracks and organized in a **box structure**. The MP4 format is capable of storing metadata such as video duration and per frame display and decoding timestamps, dealing with issues such as synchronization between tracks and random access (seeking) of the media file.

Some ISOBMFF box (or atom) types that are particularly relevant are:

- `ftyp` (*file type*), which includes some general information about the file type, e.g. the version of the MP4/QuickTime specification the file is compliant with.
- `moov` (*movie header*), containing metadata about the tracks, such as the creation time, the duration, the resolution, the bitrate, the tables that allow players to quickly find the part of the file where a specific frame or timestamp can be found, etc.
- `mdat` (*movie data*), containing the actual audio and video samples.

An H.264 bitstream muxed in an MP4 container does not need start codes and therefore must not be in the Annex B format. H.264 bistreams in MP4 files are stored in a format known as `avcC`, where the H.264 NAL units are preceded by their length.

A typical structure of an MP4 file consists of a `ftyp` box followed by a `moov` box (containing the tracks metadata), and finally the actual data stored in a `mdat` box. The `moov` box is sometimes placed at the end of the file, since the information required to create the box is typically available only at the end of the encoding process.

For some use cases, such as adaptive video streaming, a more suitable structure is the one provided by the **fragmented MP4** format, or `fMP4`. In this format, the sequence is divided into fragments that can be transmitted and played independently. The structure of an `fMP4` file still has `ftyp` and `moov` boxes at the beginning, shared among all fragments, while the rest of the file consists of a sequence of `moof` and `mdat` boxes, where the `moof` box contains metadata about the fragment, such as the fragment index and the timestamp, and the `mdat` contains the video or audio samples for a single fragment (typically a few seconds of content).

2.4 Adaptive bitrate technologies

In a typical live streaming architecture, we can generally distinguish three main parts: **ingestion**, **transcoding and packaging**, and **distribution**.

[6]The actual box structure is defined in MPEG-4 Part 12 (ISOBMFF).

```
[ftyp] size=8+16
  major_brand = isom
  minor_version = 1
  compatible_brand = isom
  compatible_brand = avc1
[moov] size=8+424781
  ...
  [trak] size=8+412772
  ...
  [trak] size=8+5742
  ...
  [trak] size=8+5718
  ...
[mdat] size=8+355431683
...
```

(a) MP4 file structure.

```
[ftyp] size=8+20
  major_brand = isom
  minor_version = 1
  compatible_brand = isom
  compatible_brand = avc1
  compatible_brand = iso5
[moov] size=8+1542
  ...
[moof] size=8+2088
  ...
[mdat] size=8+198008
  ...
[moof] size=8+2088
  ...
[mdat] size=8+198008
  ...
```

(b) Fragmented MP4 (`fMP4`) file structure.

Figure 2.8: Comparison between MP4 and `fMP4` file structures. Extracted with the `mp4dump` tool (Bento4).

Ingestion refers to the process of making the input feed available to the transcoder server for processing. Often, this means having a client push the stream through an ingestion protocol such as **Real Time Messaging Protocol** (RTMP) or **Secure Reliable Transport** (SRT). In some cases, a more suitable approach is a pull-based one, where it is the backend system's responsibility to "pull" the video and audio streams from another system and pass them onto the pipeline.

Usually, the input feed is a high-bitrate/high-quality stream that is not suitable for streaming. The de facto standard for video streaming in the industry is **Adaptive Bitrate Streaming** (ABR) or **HTTP Adaptive Streaming** (HAS), which is based on the idea of producing multiple video renditions in advance on the server side, each of them having different encoding parameters, such as video resolution, framerate and most importantly average/peak bitrate. The aim is to provide the client with the best video variant according to the estimated network throughput. The two main ABR protocols in the industry are **MPEG-DASH** and **Apple HLS**, as we will see in detail.

In order to allow clients to switch to a different video rendition when network conditions change, video files are split into small segments with a fixed duration that is usually between 2 and 10 seconds. At each segment boundary, the client can choose to switch to a different quality level by simply starting to download video segments of another rendition. The strategy that determines which video quality level should be requested next is usually implemented on the client side by **rate adaptation algorithms**.

Video segment files can be generated statically and published on a web server (**static packaging**) or dynamically on the fly (**dynamic packaging** or Just-In-Time packaging). The packaging step is performed by a **packager**, or segmenter, which also produces the metadata that allows the client to obtain the list of video renditions (identified by their bitrate and other encoding parameters) and the list of segment files for each rendition. In MPEG-DASH, there is a single-file manifest containing this information, while in HLS there are multiple playlist files.

It should be noted that the video and audio files produced by this step need to be in a coding and container format compatible with the chosen adaptive bitrate technology. For example, MPEG-DASH (see Section 2.4.2) requires streams to be muxed in a fragmented MP4 container. Also, for best performance, the segments should start with an I-frame, so that the player can switch between renditions at segment boundaries. For this reason, the GOP size of the renditions should match the length of the segment. Another important thing to take into account is the bitrate, which should not oscillate too much so that the bitrate adaptation algorithms can use the average bitrate as a reference value to make decisions.[17]

Although ABR streaming is a generic term for adaptive bitrate technologies, most of the time segmented video and audio tracks and manifest files are delivered to the client through the **Hypertext Transfer Protocol** (HTTP). Using HTTP for **distribution** makes it easy for streaming platforms to scale, thanks to the widespread availability of **Content Delivery Networks** (CDN) and their global distribution networks. One of the important advantages of relying on CDNs is the possibility to cache video segments at the edge of the networks, as close as possible to the final users, therefore reducing latency and the probability of congestion.

Also, transporting video traffic over HTTP instead of, for example, UDP avoids issues with firewalls and is in general highly reliable, since video becomes normal HTTP traffic. Apart from scalability, mentioned above, this approach makes it possible to take advantage of important HTTP features such as HTTPS for secure communication.

2.4.1 Apple HTTP Live Streaming (HLS)

HTTP Live Streaming (HLS) is an adaptive streaming protocol based on HTTP designed by Apple. First introduced in 2009 and later published as RFC 8216, it is estimated to be the most popular streaming protocol, with more than 73% video developers using it.[23][4] The reason for this is not only the fact that HLS is the only HAS protocol that can be used on Apple mobile devices, but also that it is a simple protocol to understand and implement.

HLS uses the **extended M3U format** (`.m3u8` extension) to define a set of playlists that contain information such as the list of streams or the list of segments for a particular stream. In particular, a **master playlist** provides a set of variant streams[7], each of which is a different version of the same content, usually differing by bitrate and other encoding parameters. Players download the master playlist at the beginning of the playback session and use the information contained in it to adapt to network conditions by switching between variants. An example of a master playlist is shown in Figure 2.9.

```
#EXTM3U
#EXT-X-STREAM-INF:BANDWIDTH=1280000,AVERAGE-BANDWIDTH=1000000
low.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=2560000,AVERAGE-BANDWIDTH=2000000
mid.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=7680000,AVERAGE-BANDWIDTH=6000000
high.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=65000,CODECS="mp4a.40.5"
audio-only.m3u8
```

Figure 2.9: Example of a HLS master playlist with variants. The first three variants contain both video and audio, while the fourth variant is audio-only. The first variant, corresponding to the `low.m3u8` media playlist, is defined with an average bitrate of 1000 kbps and a peak bitrate of 1280 kbps.

When players want to play a specific variant, they load the **media playlist** corresponding to the variant. A media playlist must specify the maximum duration of the segments in seconds, followed by a list of **media segments**, characterized by their duration and URI (relative or absolute). In the case of on-demand content, the media playlist must contain the full list of segments that make up the video or audio file, while in the case of live streams, the playlist may include only a sliding window of segments. An example of a media playlist is shown in Figure 2.10.

The HLS media segments can be muxed into the MPEG-2 TS format (`.ts` files) or in the fragmented MP4 format (`.mp4` or `.m4s` files). The supported video codecs are H.264 and H.265. When using `fMP4` as the container format, media playlists must specify an **initialization segment** containing the `ftyp` and `moov` boxes with the `#EXT-X-MAP` tag.

[7]In the HLS RFC, the terms *variant* and *rendition* have different meanings. We will use the term *variant* in this section to adhere to the RFC, while using the two terms interchangeably in the rest of the thesis.

```
#EXTM3U
#EXT-X-TARGETDURATION:6
#EXT-X-VERSION:3
#EXTINF:6,
segment-1.ts
#EXTINF:6,
segment-2.ts
#EXTINF:3,
segment-3.ts
#EXT-X-ENDLIST
```

Figure 2.10: Example of a simple HLS VoD media playlist with three segments and a segment target duration of 6 seconds.

In addition to the RFC, Apple maintains an *Authoring Specification for Apple Devices* defining additional rules for HLS streams, which are often taken as “safe” practices in the industry when producing content for adaptive bitrate streaming.[3]

The authoring specification provides a set of recommendations to ensure compatibility with HLS implementation that can be found on Apple devices, including the following:

- A table with a possible set of variants with their bitrate and frame rate (a *bitrate ladder*), for both H.264 and H.265, plus a recommendation to adjust them according to the specific use case.
- An indication that the peak bitrate should be no more than 200% of the average bitrate, although it is often recommended to make the bitrate as constant as possible.[17]
- A recommendation for a segment duration of 6 seconds, although 4 seconds or lower are common values in practice.[17]

HLS also includes support for encryption, ads, trick play (scrubbing and fast forward), subtitles and multiple audio tracks.

There is a variant of HLS, known as **Low-Latency HLS** (LL-HLS), which is designed for low-latency live streaming. It is based on a technique called *Blocking Preload Hints*, which relies on the fact that segments are made available in smaller parts called partial segments, which are published as soon as possible and can thus be downloaded before the full segment is made available. The HLS manifest now contains a “hint” about which partial segment should be requested next, so that the client can send the HTTP request before the partial segment is actually available. The server will “block” and respond with the part as soon as it is available. LL-HLS has a mandatory requirement for HTTP/2, therefore we did not take it into consideration for the work in this thesis, which focuses on HTTP/3.

2.4.2 MPEG Dynamic Adaptive Streaming over HTTP (DASH)

MPEG Dynamic Adaptive Streaming over HTTP (DASH) is an alternative to HLS for adaptive streaming over HTTP. It was developed by the MPEG group and first published in 2012. Since it is the officially supported ABR protocol on HbbTV, the entertainment system for smart TVs, it gained popularity and is often mandatory for streaming platforms that want to target TVs. It is not supported on iOS and Apple TV, so HLS is often still an easier choice to ensure broad compatibility.

DASH is a very complex standard, so we will just give a general overview of how it works.[7][27] There are several differences between HLS and DASH:

- Unlike HLS, DASH is codec-agnostic, therefore it can be used with any codec without waiting for the standard to be updated.
- It uses the `fMP4` container format by default, thus reducing the overhead introduced by MPEG-2 TS.

- DASH does not use M3U playlists: there is a single manifest file (`.mpd` extension) in XML format.

The manifest file has a nested structure that combines the equivalent of the master and media playlists of HLS. The main elements in a DASH manifest file are:[8]

- **Adaptation sets**: they define a group of representations (renditions) on which adaptation algorithms are executed. A common approach is to have one video adaptation set and multiple audio adaptation sets depending on the number of supported audio languages.
- **Representations**: they represent the individual renditions in an adaptation set. Each representation usually has different encoding parameters, such as the bitrate, resolution and framerate, but could also have different codecs.
- **Media segments**: within a representation, there are multiple ways to specify how the player should obtain the media segments. For example, with `SegmentList` we can insert the full list of segments in the manifest file. A smarter approach is to use the `SegmentTemplate` element, which allows the packager to define a pattern for predictable sequences of segments.
- **Index segments**: they are the equivalent of the initialization segment, introduced with HLS.

A minimal example of a DASH manifest file is shown in Figure 2.11. In this case, there are two adaptation sets, one for video and one for audio. The video adaptation set contains two representations, corresponding to 720p and 540p. The file names of the segments are determined on the fly by the player using the `SegmentTemplate` definition. A similar approach is used for the audio adaptation set.

In the example of Figure 2.11, the `type` field is set to `static`, which means that this manifest is for on-demand content. In the case of a live stream, the `type` has value `dynamic` and some additional fields are required. For example, the `availabilityStartTime` field specifies the timestamp corresponding to the start of the live stream, and is used by the player to calculate which segment should be requested next. Another important field is the `suggestedPresentationDelay`, which determines the live latency target (which can be overridden by the player).

The MPEG-DASH standard also has an extension for low-latency streaming, known as **Low-Latency DASH** (LL-DASH). The approach of LL-DASH is conceptually similar to that of LL-HLS, but is technically implemented in a different way. Specifically, it takes advantage of the *Chunked Transfer Encoding* (CTE) feature of HTTP/1.1. Similarly to LL-HLS, LL-DASH makes use of chunks that can be downloaded as soon as they are published. The client sends a single HTTP request per segment and waits for chunks to be delivered through HTTP CTE, decoding and playing them as soon as possible. Compared to LL-HLS, LL-DASH has a smaller overhead since it sends a much lower number of HTTP requests.[15]

2.4.3 Common Media Application Format (CMAF)

Before Apple introduced support for `fMP4` segments in 2016, a common issue for video platforms was that supporting both HLS and DASH required two copies of the same content, one in MPEG-2 TS format and the other in `fMP4`. This was needed because not all platforms support HLS and vice versa.

This problem led to the proposal and standardization of **Common Media Application Format** (CMAF), a common specification that defines how media should be packaged and segmented for delivery. Fragmented MP4 is a CMAF-compatible format and, since it is supported by both HLS and DASH, it can lead to significant savings. Video platforms can, in fact, produce and store a single set of video segment files and deliver them to all devices.[30]

2.4.4 Media Source Extensions (MSE)

One of the main medium through which users watch video streams is web browsers. Apple includes native support for its HLS protocol in its own browser, Safari, which is pre-installed on all Apple devices (and is also the only browser engine that can be used on iOS). Other browsers typically do

[8]`https://www.brendanlong.com/the-structure-of-an-mpeg-dash-mpd.html`

```xml
<?xml version="1.0" encoding="utf-8" ?>
<MPD xmlns="urn:mpeg:dash:schema:mpd:2011" profiles="urn:mpeg:dash:profile:full:2011"
     type="static" mediaPresentationDuration="PT60S">
    <Period>
        <!-- Video -->
        <AdaptationSet id="0" mimeType="video/mp4" segmentAlignment="true"
                       startWithSAP="1" maxWidth="1280" maxHeight="720">
            <SegmentTemplate timescale="1000" duration="2000" startNumber="1"
                             initialization="$RepresentationID$/init.mp4"
                             media="$RepresentationID$/seg-$Number$.m4s" />
            <Representation id="720p" codecs="avc1.64001F" width="1280" height="720"
                            scanType="progressive" frameRate="25" bandwidth="3301246" />
            <Representation id="540p" codecs="avc1.64001F" width="960" height="540"
                            scanType="progressive" frameRate="25" bandwidth="2409696" />
        </AdaptationSet>
        <!-- Audio -->
        <AdaptationSet mimeType="audio/mp4" startWithSAP="1" segmentAlignment="true">
            <SegmentTemplate timescale="1000" duration="2000" startNumber="1"
                             initialization="$RepresentationID$/init.mp4"
                             media="$RepresentationID$/seg-$Number$.m4s" />
            <Representation id="audio" codecs="mp4a.40.2" bandwidth="131632"
                            audioSamplingRate="48000">
                <AudioChannelConfiguration
                    schemeIdUri="urn:mpeg:mpegB:cicp:ChannelConfiguration"
                    value="2" />
        </Representation>
        </AdaptationSet>
    </Period>
</MPD>
```

Figure 2.11: A minimal example of an MPEG-DASH manifest files, showing two adaptation sets (video and audio). The video set contains two representations, 720p and 540p.

not include support for HLS and DASH, but instead support the **Media Source Extensions** (MSE) API, through which JavaScript libraries can implement HLS and DASH.[8]

MSE is a set of APIs that abstract the implementation of media playback in the browser. MSE exposes types and methods that allow the client-side code to push video and audio data for playback.

More in detail, MSE defines the `MediaSource` type as a source of media data for the `<video>` or `<audio>` HTML5 media elements, to which a `MediaSource` is attached. The client code interacts with the `MediaSource` to push new data, while the media elements fetch the data from the `MediaSource`.

`MediaSource` instances expose methods for attaching a set of `SourceBuffer` objects, which represent individual media tracks such as video or audio streams. When new media data is available, such as a new video segment, the client calls the `appendBuffer()` method on the corresponding `SourceBuffer`, providing the new data for the track to the browser. The bytes of data must of course conform to the format set when creating the `SourceBuffer`, which must be supported by the browser. Most of the time this means the ISOBMFF (`fMP4`) or WebM container formats. Codec support instead often depends on the browser build/configuration and platform.

The libraries that implement HLS or DASH must take care of parsing the manifest files, downloading the segments, implementing rate adaptation algorithms, possibly transmuxing from MPEG-2 TS to fragmented MP4, appending the media data to the source buffers while leaving the MP4 demuxing and stream decoding tasks to the browser.

In practice, this means that there can be many implementations of HLS and DASH, all relying on the Media Source Extensions internally. For HLS, a popular implementation is `HLS.js`[9], while for

[9]`https://github.com/video-dev/hls.js/`

DASH the official reference implementation is `dash.js`[10]. Several other JavaScript players include support for HLS and DASH, such as `Video.js`, `Shaka Player`, `THEOPlayer`, and `JW Player`, some of which are open source while others are commercial.

2.5 Hypertext Transfer Protocol (HTTP) evolution

When browsing the Web, we use an application layer protocol known as **Hypertext Transfer Protocol** or **HTTP**. In addition to being used for transferring HTML web pages and related files, HTTP can also be used by adaptive streaming technologies to deliver video and audio segments to the client, as mentioned previously.

In this section, we will briefly go through three important versions of HTTP, with their limitations and peculiarities.

2.5.1 HTTP/1.1

For a long time, the dominant HTTP version has been **HTTP/1.1**, first published in 1997.[16] It is based on the Transmission Control Protocol (TCP), the reliable connection-oriented transport layer protocol.

To send an HTTP request, a client should open a TCP connection with the server. It will then transmit the request line, followed by a list of headers, an empty line, and optionally a payload or message body. The following is an example of an HTTP/1.1 `GET` request to the URL `http://example.com/index.html`.

```
GET /index.html HTTP/1.1
Host: example.com
User-Agent: curl
Accept: */*
```

Once the server has received the request, it will do its processing and then start sending the response on the same TCP connection. The response consists of a status line, a list of headers, and optionally a message body, as in the following example.

```
HTTP/1.1 200 OK
Date: Thu, 22 Dec 2022 08:00:00 GMT
Content-Type: text/html; charset=UTF-8

<html>
<head>
...omitted...
```

For encrypted communication, HTTPS uses **Transport Layer Security** (TLS) to create a secure communication channel over TCP. The HTTP/1.1 request is then transmitted over the encrypted TLS session.

When keep-alive (persistent connections) is enabled, the same connection can be reused to send multiple HTTP requests, avoiding the overhead of establishing a new TCP connection and TLS session. HTTP/1.1 also supports request pipelining, meaning that multiple requests can be sent on the same TCP connection before receiving the responses, although it is rarely used because not all server implementations support it properly.

The main limitation of HTTP/1.1 is, however, that it does not support multiplexing of responses, meaning that responses must be transmitted sequentially by the server. This causes a problem known as **HTTP head-of-line blocking**, where the current request blocks the following requests. In an attempt to work around this limitation, browsers typically open multiple TCP connections to the same web server when loading web pages, up to a (configurable) limit.[11]

[10]`https://github.com/Dash-Industry-Forum/dash.js/`

[11]For example, Firefox uses 6 as a default connection limit, but the value can be changed through the `network.http.max-persistent-connections-per-server` preference.

2.5.2 HTTP/2

HTTP/2, published as a Proposed Standard in 2015 as RFC 7540, was an attempt to solve the main issues of HTTP/1.1, most notably the head-of-line blocking problem. At the same time, the aim of HTTP/2 was to use less network resources while reducing the perception of latency.[10]

In HTTP/2, clients establish a **single TCP connection** with the server. To allow requests and responses **multiplexing**, the protocol introduces a binary framing format in which requests and responses, which maintain the same HTTP semantics as in HTTP/1.1, are split into smaller units, known as frames. Frames can be interleaved, allowing multiple requests and responses to be transmitted in parallel. In order to identify to which request a frame belongs to, the concept of **stream** is introduced. Streams can be seen as virtual connections that are very light to create and generally correspond to single HTTP requests. Streams can also be reset, with the effect of immediately stopping the transmission of the associated data frames.

An important feature of HTTP/2 is **stream prioritization**. Since multiple streams can co-exist on the same TCP connection, there is the need for a mechanism to make better decisions on which stream goes first or is given priority. HTTP/2 allows clients to assign a **weight** between 1 and 256 to each stream, where a higher number means higher priority. In addition, clients can specify that a stream is dependent on another stream through its ID, effectively creating a **dependency tree**. The structure of the tree, and therefore the order in which requests are responded, is defined by the dependency relationships, while the division of resources between a set of "children" streams is determined by their weights.

The dependency tree approach of HTTP/2 is very flexible and allows clients to implement different **multiplexing** or **prioritization strategies**, with widely different performance results. In [31], Wijnants et al. examined the behavior of the main browsers to determine which prioritization scheme they implemented. It turns out that only Mozilla Firefox chose to adopt a complex tree-based approach. Google Chrome implements a much simpler sequential FCFS approach based on dynamic weights (giving more weight to requests deemed more important according to heuristics), Microsoft Edge (legacy) uses a naive round-robin strategy with no weights at all, while Safari follows a weighted round-robin scheme. Surprisingly, the experimental results showed superior performance when using the relatively simple strategy implemented by Chrome.

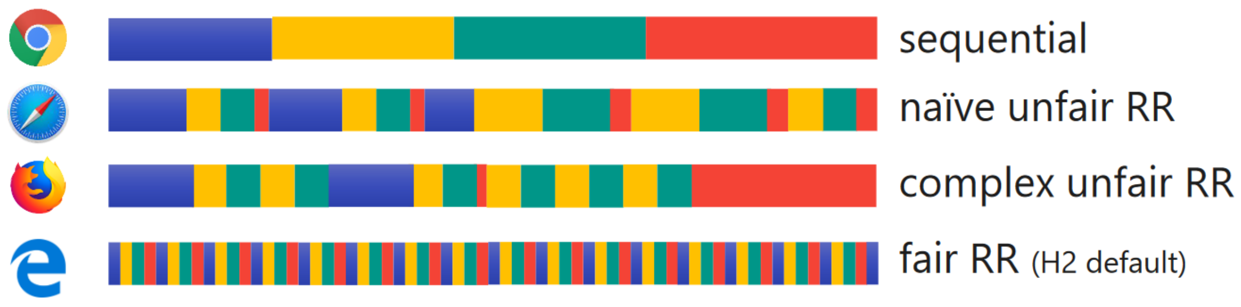


Figure 2.12: Different HTTP/2 multiplexing strategies implemented by browsers. Courtesy of Robin Marx.[12]

In HTTP/2, the use of HTTPS is mandatory, making the protocol more secure by design. HTTP/2 also achieves better performance by compressing headers, thanks to the `HPACK` compression algorithm. Finally, a feature known as **HTTP/2 Server Push** allows the server to push additional unsolicited content in response to HTTP requests, theoretically reducing the need for additional round-trip times to request resources.

While all these features make sense from a theoretical point of view, in practice HTTP/2 did not manage to provide the expected performance improvements, for multiple reasons:

- The **Server Push** feature never really took off, with virtually no CDN having support for Server Push. Additionally, in 2022 Google removed Server Push support from Google Chrome,

[12]`https://quic.edm.uhasselt.be/files/fosdem.pdf`

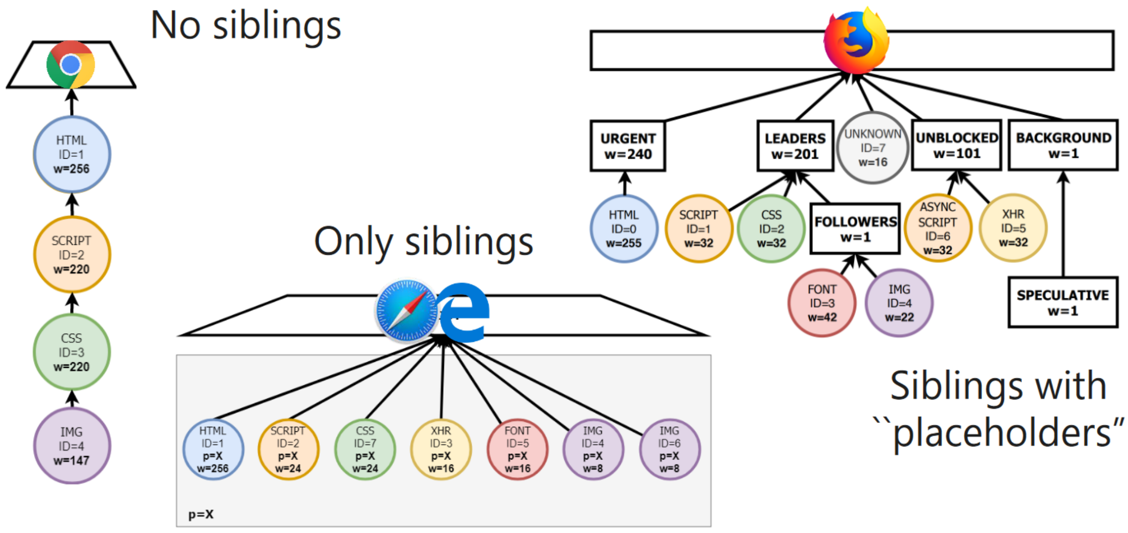


Figure 2.13: Graphical representation of the HTTP/2 dependency trees in the main web browsers. Courtesy of Robin Marx.

citing extremely low usage, almost no gain in performance, and complexities in the browser implementation. Instead, it suggested developers to use the **preload hints** or **103 Early Hints** features, which are easier to use and can provide significant performance gains.[13]

- **Stream prioritization** was found to be inconsistently implemented by clients and servers, with a significant number of large CDNs apparently ignoring prioritization.[14] This led to the prioritization scheme of HTTP/2 being deprecated and replaced by a new mechanism at the same time HTTP/3 came out.

- When the **sequential prioritization strategy** is used, we get the same HTTP head-of-line blocking issue seen in HTTP/1.1, worsened by the fact that only a single TCP connection is used.

- The use of a single TCP connection leads to another problem known as **TCP head-of-line blocking**, where the bottleneck is not the HTTP design but TCP itself. In fact, while HTTP/2 multiplexes requests in multiple streams, at the transport layer TCP only sees a single opaque byte stream. As a consequence, TCP retransmissions delay the transmission of all streams and not just the one affected by packet loss. Additionally, TCP congestion control may reduce the throughput of the connection in case of congestion, affecting all the requests, while in HTTP/1.1 it is less likely that all the connections are affected at the same time.

2.5.3 HTTP/3

To overcome the limitations of TCP-based approaches, the newest version of HTTP, **HTTP/3**, is based on a new transport protocol called QUIC, which is built on top of the lightweight UDP. While HTTP/3 by itself does not introduce new features, the fact that it is based on a new protocol like QUIC allows to rethink how a transport layer protocol should work from the ground up. QUIC was published as a Proposed Standard in May 2021 with RFC 9000, with HTTP/3 following in June 2022 as RFC 9114, after a few years of work.[11]

More in depth, the following are some main conceptual differences between HTTP/2 and HTTP/3:

- Since QUIC is based on UDP, TCP features such as connection establishment, flow control, congestion control and retransmissions are now implemented by QUIC.

[13]`https://developer.chrome.com/blog/removing-push/`
[14]`https://github.com/andydavies/http2-prioritization-issues`

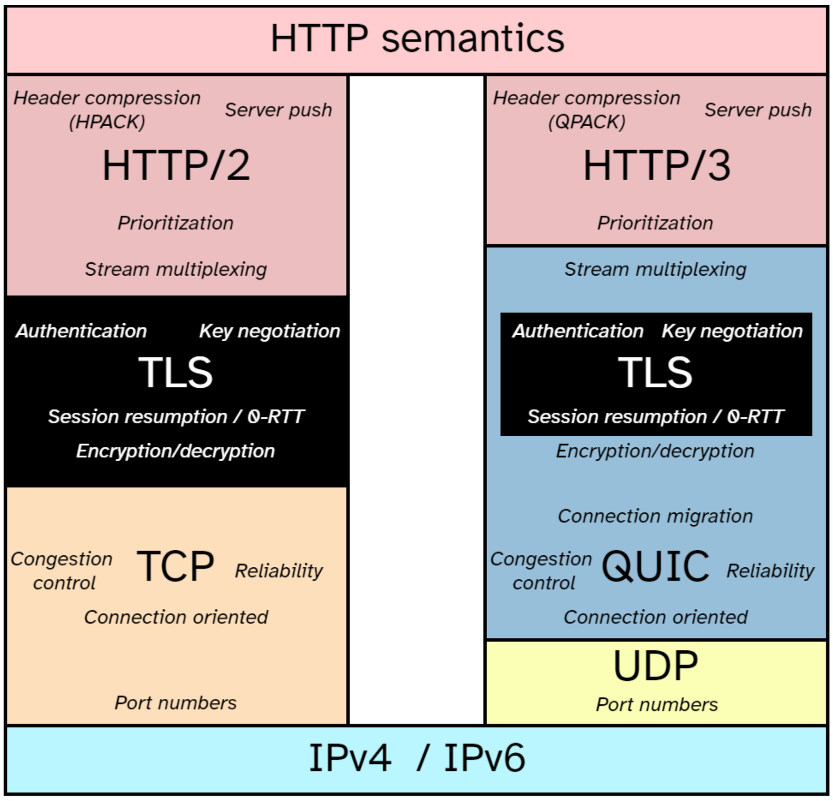


Figure 2.14: Comparison of HTTP/2 and HTTP/3 protocols stack. Courtesy of Robin Marx.[15]

- **TLS v1.3** is tightly integrated with QUIC, making security features such as encrypted communication mandatory. At the same time, this approach allows innovative features such as 1-RTT handshakes or 0-RTT session resumption to be seamlessly implemented, providing further performance improvements.
- Unlike TCP, QUIC provides a seamless **connection migration** feature that allows the reuse of connections when there are changes in network connectivity.
- **Stream multiplexing** is moved "down" from the application layer to QUIC. HTTP/3 takes advantage of QUIC streams to provide request multiplexing. HTTP/3 clients or servers can cancel HTTP requests by **resetting** the underlying QUIC stream.
- HTTP/2 **stream prioritization** is replaced by a simpler priority mechanism, implemented at the HTTP/3 level. QUIC does not support stream prioritization.

The **head-of-line blocking** issue caused by TCP in HTTP/2 is not present in HTTP/3, since streams are now a concept of the transport layer. Unlike TCP, in fact, QUIC does not see a single byte stream but multiple streams that can be treated independently (with respect to retransmissions, for example). However, if the scheduling strategy implemented by the server is sequential, there could still be head-of-line blocking because the slow transmission of a stream would block the other ones. The multiplexing scheduler depends on the server implementation and is often round-robin, according to the experiments run by Marx et al.[20]

The prioritization scheme used by HTTP/3 was introduced in RFC 9218 (*Extensible Prioritization Scheme for HTTP*), published together with HTTP/3. As the title suggests, the new mechanism is actually not HTTP/3-specific and is defined for both HTTP/2 and HTTP/3. Clients that implement RFC 9218 can communicate their preferences on how the server should prioritize responses in two ways:[21]

- Through an HTTP header called `Priority`, in which the priority for the request is specified.

[15]https://github.com/rmarx/h3-protocol-stack

- Through a `PRIORITY_UPDATE` frame, so that requests can be reprioritized after they have been sent. With HTTP/3, priority update frames are transmitted over a unidirectional QUIC stream dedicated to control commands.

The HTTP requests priorities defined by RFC 9218 are characterized by two parameters:

- The **urgency** parameter (`u`), an integer value ranging from 0 to 7 in descending order of priority, with a default value of 3.
- The **incremental** parameter (`i`), indicating whether an HTTP response can be processed incrementally by the client, and therefore if it can benefit from being received concurrently. For example, the download of a standard (non-progressive) JPEG image file would not benefit from being transmitted incrementally since the JPEG decoder needs the whole image to start decoding the picture. On the other hand, a progressive JPEG image can be rendered at increasing quality levels progressively while it is being downloaded, and could therefore benefit from being delivered incrementally. The default value of the incremental parameter is 0 (`false`).

When using the `Priority` header, the urgency and incremental parameters can be specified as a string in the following format: `u=3, i=0`.

Since HTTP/3 uses the same transport layer port as previous HTTP versions (port 443) but with a different transport layer protocol (TCP vs UDP/QUIC), a **protocol negotiation** mechanism is needed. This is currently done through the HTTP `Alt-Svc` response header, through which servers can specify if and on which port alternative services such as HTTP/3 are available.

Browsers usually send the first connection through TCP-based HTTP. If the response contains an `Alt-Svc` header indicating that HTTP/3 is available, the TCP connection is closed and the browser switches to QUIC on subsequent requests. An example of a header advertising HTTP/3 on port 443 is `Alt-Svc: h3=":443"; ma=2592000`, where `ma` (max-age) corresponds to the number of seconds for which the alternative service configuration should be considered valid (defaults to 24 hours). There is also a proposal to negotiate the HTTP version through DNS records, but it is still a draft.[16]

Server implementations

Even before the QUIC specification was published, several open source libraries implemented QUIC. Some of the most notable ones are Cloudflare's `quiche`, Fastly's `quicly`, Google's `quiche`, Meta's `mvfst`, and LiteSpeed's `lsquic`.[20] Some of these libraries also provide support for HTTP/3, although they are not HTTP web servers that can be used out of the box.

Popular web servers such as **Apache** and **nginx** currently do not support HTTP/3. For example, nginx support for HTTP/3 was delayed because OpenSSL, the TLS implementation used by nginx, has yet to implement the interfaces needed by QUIC. nginx is therefore planning to switch to another TLS library and expects HTTP/3 support to be finalized by the end of 2022.[17]

Some lesser known web servers that include HTTP/3 support are Fastly's h2o (based on `quicly`), LiteSpeed Web Server (based on `lsquic`), and Caddy (based on `quic-go`).

Browser support

It is estimated that around 75% of the market share of browsers is capable of browsing with HTTP/3. Google Chrome added support for HTTP/3 in 2020, Mozilla Firefox in 2021, while Apple Safari support for HTTP/3 is still experimental at the time of writing.[18] According to Cloudflare Radar, HTTP/3 traffic accounts for around 28% of global web traffic, with HTTP/2 at 65% and HTTP/1.x at 7%.[19]

[16]`https://datatracker.ietf.org/doc/draft-ietf-dnsop-svcb-https/`
[17]`https://www.nginx.com/blog/our-roadmap-quic-http-3-support-nginx/`
[18]`https://caniuse.com/http3`
[19]`https://radar.cloudflare.com/`

Debugging tools

When conducting research or developing applications based on HTTP/3, it is often useful to dig deeper into the raw HTTP/3 or QUIC frames. There are several tools that make this possible:

- **Wireshark**, the popular network protocol analyzer, includes support for QUIC and HTTP/3, although still very basic and missing most features.[20]
- `qlog`, a project of the IETF QUIC Working Group with the objective of introducing a common logging format for QUIC. Both client and server implementations can support `qlog`, providing a way to capture and export network traces with QUIC-related events in a common format. `qlog` captures can then be parsed by `qlog` analyzers.[21]
- Chrome's **NetLog dump**, which is a detailed capture of network activity from the perspective of the browser. It can be generated by browsing to `chrome://net-export` and following the instructions. The format of the output dump file is proprietary, but there are visualization tools that support the NetLog dump format.[22]
- `qvis`, developed by the authors of [20], is a suite of tools to analyze `qlog`, NetLog dumps or Wireshark captures and extract advanced insights from QUIC traces. The suite includes many visualization tools such as sequence diagrams, congestion control plots, multiplexing and packetization visualizations, and other detailed statistics.[23]

2.6 Network emulation

When developing network-based applications or running benchmarks against the network, it is often useful to be able to set up a testbed that emulates an actual network. There are several software tools that do this, among which there are **Mininet** and **ComNetsEmu**, which we will briefly explain in the following sections.

2.6.1 Mininet

Mininet is an emulation tool that allows to create realistic virtual networks on a single machine. In Mininet, you can create hosts and switches, connect them through links and make them interact. Mininet also supports **Software-Defined Networking**, which makes it possible to have switches controlled by an SDN controller using the OpenFlow protocol.

With Mininet, you can create arbitrarily complex custom topologies through the CLI or the Python API. Everything can be done on a single machine thanks to the light impact of the tool. In fact, unlike traditional virtualization-based approaches, Mininet implements lightweight OS-level network virtualization by using features of the Linux kernel, such as network namespaces. The result is that Mininet boots much faster, usually in seconds, can easily scale with modest system requirements and performs well with high-bandwidth applications.[24]

2.6.2 ComNetsEmu

ComNetsEmu is an extension to Mininet that aims at complete host isolation thanks to **Docker containers**. Since ComNetsEmu enforces stricter node isolation, it is heavier than Mininet.

Similarly to Mininet, ComNetsEmu can be programmed through Python. In ComNetsEmu, both hosts and applications deployed on hosts are represented by Docker containers. The application containers are deployed as "siblings" containers to emulate a "Docker-in-Docker" architecture.

The recommended way to run ComNetsEmu is to use a Vagrant virtual machine, which takes care of setting up the software and all the required dependencies.[32]

[20] `https://gitlab.com/wireshark/wireshark/-/issues/16761`
[21] `https://github.com/quicwg/qlog`
[22] `https://www.chromium.org/for-testers/providing-network-details/`
[23] `https://qvis.quictools.info/`
[24] `http://mininet.org/overview/`

```
$ sudo mn
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:

mininet> nodes
available nodes are:
c0 h1 h2 s1

mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s1-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h2-eth0
c0

mininet> h1 ping -c 1 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=2.51 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 2.507/2.507/2.507/0.000 ms

mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)

mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h2
.*** Results: ['21.0 Gbits/sec', '21.0 Gbits/sec']
```

Figure 2.15: Example usage of the Mininet CLI with the default `minimal` topology. Mininet allows to run Linux network tools such as `ping` and `iperf` directly on the virtual hosts.

Chapter 3

Testbed setup

In this section, we will describe how we set up a **testbed** with ComNetsEmu to evaluate the performance of **low-latency adaptive bitrate live streaming** technologies. The testbed allows us to run experiments in different scenarios and configurations, extracting some important metrics and events.

We will then use this testbed to evaluate the performance of some existing streaming technologies (Chapter 4) and to propose some improvements (Chapter 5).

3.1 General architecture

The testbed for evaluating the quality of experience (QoE) of live streaming solutions was built with ComNetsEmu. The project is published as an open source project at `https://github.com/matteocontrini/live-streaming-testbed`.

The general architecture of the testbed is shown in Figure 3.1. The topology of the emulated network consists of three hosts (H1, H2, H3) and two switches (S1, S2). The link between the two switches represents an unreliable network such as an Internet access network. When the emulation is run, a series of experiments with different configurations are executed and the link parameters are continuously updated. For example, the bandwidth is varied to emulate a real network with oscillating throughput.

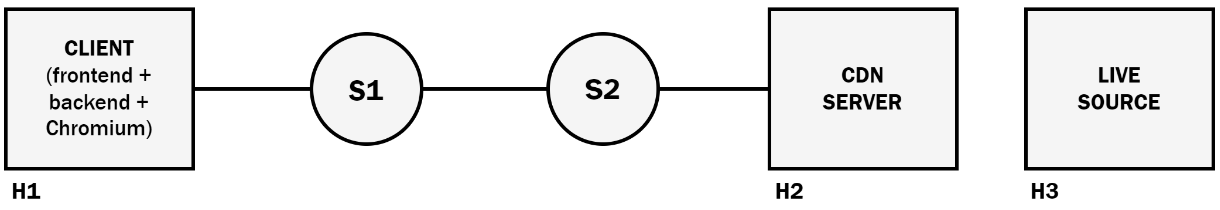


Figure 3.1: Diagram of the testbed architecture, based on ComNetsEmu.

On the three hosts, several applications are deployed as Docker containers.

- `H3` hosts a script that performs the **packaging of a live stream** in both HLS and DASH formats. It runs an `ffmpeg` command that simulates a live stream, taking an MP4 file as input. The output of this application, which we call *live source*, are the manifest and playlist files for both HLS and DASH and the corresponding video and audio segments in `fMP4` format.

- `H2` acts as an **HTTP CDN server** for the packaged stream. In practice, it is an `h2o` web server that serves the files produced by the live source, over HTTP/1.1, HTTP/2 and HTTP/3. It also takes care of provisioning the public key certificates that are required for HTTPS.

- `H1`, which is on the other side of the main network link, is the client. It consists of a frontend application containing a **video player** that plays the live video stream generated by the server.

The frontend application also takes care of collecting several metrics from the player and transmitting them to a Node.js backend application. The frontend application runs in a **Chromium headless** instance, which is started by the backend application through the **Puppeteer** library.

To start the emulation through ComNetsEmu, one first needs to clone the source code of the testbed. The ABR video renditions should then be encoded and written to an MP4 file, which the live source will then use to simulate the live stream (we will see this in detail in Section 3.2). To start the Vagrant virtual machine and start a session, the following commands can be used.

```
vagrant up
vagrant ssh
```

Finally, after building the Docker images, the emulation can be started, as shown in the following script. The testbed source code actually contains scripts that can be used to execute these commands more easily. They also handle error handling and cleanup.

```
cd live-source && docker build -t live-source .
cd ../cdn && docker build -t cdn .
cd ../client && docker build -t client .
cd .. && sudo python3 topology.py
```

Before going into the details of how the specific components were developed, it should be noted that the default Vagrant configuration for ComNetsEmu (contained in a file known as `Vagrantfile`) does not work when the machine architecture is ARM. This is especially problematic on newer laptops that use Apple Silicon hardware, such as the M1 and M2 chips.

To solve this problem, two changes must be applied to the ComNetsEmu `Vagrantfile`. First, the Vagrant box (the operating system image) must be changed to an image that is built for the ARM architecture. Then, since x86 virtualization software such as VirtualBox cannot be used on ARM systems, a new virtual machine provider must be added. In practice, for Mac systems, this means enabling Parallels Desktop as a virtual machine provider. Figure 3.2 shows how the `Vagrantfile` of ComNetsEmu was modified for Apple Silicon support.

3.2 Live stream generation and packaging

To avoid having an actual live source such as a camera, the live video stream was simulated. There are multiple ways to simulate a live video stream. For example, there exist several streaming servers that handle video ingestion, transcoding and packaging. For our testbed, we chose to implement a simpler solution based on `ffmpeg` and a static input video file.

In fact, since we do not need the live stream to be dynamic, we decided to encode the video renditions beforehand. The renditions are then packaged as live streams each time the emulation is run, as if they were ingested and transcoded live. This solution makes the test setup much lighter in terms of computational resources, without affecting the final result.

When encoding the video renditions, we chose the following bitrate ladder. All video renditions are encoded with **H.264** at **25 fps** with a keyframe/I-frame interval of **2 seconds** and a constant bitrate.

- `1280x720` at 3.5 Mbps;
- `960x540` at 2.5 Mbps;
- `640x360` at 1.5 Mbps;
- `480x720` at 0.8 Mbps.

```
diff --git a/Vagrantfile b/Vagrantfile
index 0f07076..a21824f 100644
--- a/Vagrantfile
+++ b/Vagrantfile
@@ -20,6 +20,8 @@ VM_NAME = "ubuntu-20.04-comnetsemu"
 # When using libvirt as the provider, use this box, bento boxes do not support...
 BOX_LIBVIRT = "generic/ubuntu2004"

+BOX_PARALLELS = "jeffnoxon/ubuntu-20.04-arm64"
+
 ######################
 #  Provision Script  #
 ######################
@@ -105,6 +107,14 @@ Vagrant.configure("2") do |config|
     # Sync ./ to home directory of vagrant to simplify the install script
     comnetsemu.vm.synced_folder ".", "/vagrant", disabled: true
     comnetsemu.vm.synced_folder ".", "/home/vagrant/comnetsemu"
+
+    # Parallels Desktop
+    config.vm.provider "parallels" do |prl, override|
+      override.vm.box = BOX_PARALLELS
+      prl.name = VM_NAME
+      prl.cpus = CPUS
+      prl.memory = RAM
+    end

     # For Virtualbox provider
     comnetsemu.vm.provider "virtualbox" do |vb|
```

Figure 3.2: Diff showing how the ComNetsEmu `Vagrantfile` was modified to add support for Apple Silicon machines.

The audio is instead encoded with `AAC-LC` at 128 kbps. Of course, there are many other possible configurations, but the above is a common configuration that gives good quality results.[17]

The `ffmpeg` command that was used to produce the renditions is similar to the following:

```
ffmpeg -i $INPUT \
  -c:v libx264 -pix_fmt yuv420p -preset veryfast -r 25 \
  -g 50 -keyint_min 50 -sc_threshold 0 \
  -force_key_frames 'expr:gte(t,n_forced*2)' -refs 1 \
  -c:a libfdk_aac -ac 2 -b:a 128k \
  -map 0:a:0 -map 0:v:0 -map 0:v:0 -map 0:v:0 -map 0:v:0 \
  -s:v:0 1280x720 -b:v:0 3500k -bufsize:v:0 3500k -minrate:v:0 3500k -maxrate:v:0 3500k \
  -s:v:1 960x540 -b:v:1 2500k -bufsize:v:1 2500k -minrate:v:1 2500k -maxrate:v:1 2500k \
  -s:v:2 640x360 -b:v:2 1500k -bufsize:v:2 1500k -minrate:v:2 1500k -maxrate:v:2 1500k \
  -s:v:3 480x270 -b:v:3 800k -bufsize:v:3 800k -minrate:v:3 800k -maxrate:v:3 800k \
  abr.mp4
```

In detail, the meaning of the options is as follows.

- `-i` specifies the input media file, which can be in any format supported by `ffmpeg`.
- `-c:v` specifies the video codec, in this case `libx264`, a library that implements x264 (a popular open source software-based H.264 encoder).
- `-pix_fmt` defines the pixel format. A value of `yuv420p` means that the pixels are encoded with Y'CbCr 4:2:0 chroma subsampling.

- `-preset` specifies a set of configuration options that influence which H.264 features are used and thus the encoding speed. Research has shown that `veryfast` is a good trade-off between speed and quality.[17]
- `-r` sets the frame-per-second (fps) value. In this case, 25 fps is used.
- `-g` and `-keyint_min` tell the encoder the maximum and minimum GOP size in frames, that is, the minimum and maximum interval between key frames (I-frames).
- `-sc_threshold` adjusts a parameter related to scene cut detection, which is an algorithm that analyzes the difference between frames to determine whether a new I-frame should be inserted due to a scene change. Setting the parameter to 0 means that scene detection is disabled. This is a common choice when encoding for streaming, since I-frames are very expensive compared to other frame types.[17]
- `-force_key_frames` forces a key frame when the given expression evaluates to `true` at the current frame. This option is required because `x264` does not allow setting `-keyint_min` to a value greater than `keyint/2+1`, and clips the value if it is larger.[25] The expression `gte(t,n_forced*2)` means that a keyframe should be forced every 2 seconds (in the expression, `t` is the time of the current frame and `n_forced` is the number of frames forced so far).
- `-refs` specifies the number of reference frames that each inter-predicted frame can use. Research has shown that limiting the number of frames to 1 has a negligible impact on quality while reducing the encoding time.[17]
- `-c:a` specifies the audio codec, in this case `libfdk_aac`, a high-quality AAC implementation by the Fraunhofer research institute.
- `-ac` specifies the number of audio channels, 2 in this case (stereo).
- `-b:a` specifies the audio bitrate, which is constant (CBR) by default.
- `-map` is used to select which streams of the input file should be sent to the output. For example, `-map 0:a:0` takes the first audio track from the first input file and includes it in the output file. Multiple `-map` options for the same input stream have the effect of duplicating the stream. In this case, we use `-map` to create multiple video streams (one per rendition), within a single command.
- `-s:v` is used to scale the video to the specified resolution. In our command, this option is used once per rendition to change the resolution for reach of the (duplicated) video streams.
- `-b:v`, `-minrate:v`, `-maxrate:v`, `-bufsize:v` set the average target bitrate, the minimum bitrate, the maximum bitrate, and the buffer size of the encoder. By setting all the options to the same value the result is a constant bitrate (CBR) video stream, which is often recommended for adaptive bitrate streaming and helps complying with Apple HLS Authoring Specification.[17][3]

`ffmpeg` will choose the appropriate **H.264 profile and level** based on the selected preset and the other encoding parameters. In our case, the profile will be set to `High` for all the renditions, while the level varies between 2.1 and 3.1 depending on the resolution and bitrate.

The output generated by the command is a single MP4 file that contains multiple tracks, one per each video rendition and one for the audio, as shown in Figure 3.3.

When the emulation starts, the *live source* application container is started. This application acts as the HLS and DASH packager and is again a `ffmpeg` script. The script takes the previously generated MP4 file as input and generates the `fMP4` chunks (for each rendition), plus the HLS playlists and the DASH manifest.

The `ffmpeg` command used for packaging is similar to the following:

[25]`https://github.com/mirror/x264/blob/b093bbe7d9bc642c8f24067cbdcc73bb43562eab/encoder/encoder.c#L1111`

```
Input #0, mov,mp4,m4a,3gp,3g2,mj2, from 'abr.mp4':
  Metadata:
    major_brand     : isom
    minor_version   : 512
    compatible_brands: isomiso2avc1mp41
    title           : Big Buck Bunny, Sunflower version
    artist          : Blender Foundation 2008, Janus Bager Kristensen 2013
    composer        : Sacha Goedegebure
    encoder         : Lavf59.31.100
    comment         : Creative Commons Attribution 3.0 - http://bbb3d.renderfarming.net
    genre           : Animation
  Duration: 00:10:34.64, start: 0.000000, bitrate: 7884 kb/s

  Stream #0:0[0x1](und): Audio: aac (LC) (mp4a / 0x6134706D), 48000 Hz, stereo, fltp, 128 kb/s
  ↪  (default)

  Stream #0:1[0x2](und): Video: h264 (High) (avc1 / 0x31637661), yuv420p(progressive), 1280x720 [SAR
  ↪  1:1 DAR 16:9], 3238 kb/s, 25 fps, 25 tbr, 12800 tbn (default)

  Stream #0:2[0x3](und): Video: h264 (High) (avc1 / 0x31637661), yuv420p(progressive), 960x540 [SAR
  ↪  1:1 DAR 16:9], 2341 kb/s, 25 fps, 25 tbr, 12800 tbn (default)

  Stream #0:3[0x4](und): Video: h264 (High) (avc1 / 0x31637661), yuv420p(progressive), 640x360 [SAR
  ↪  1:1 DAR 16:9], 1409 kb/s, 25 fps, 25 tbr, 12800 tbn (default)

  Stream #0:4[0x5](und): Video: h264 (High) (avc1 / 0x31637661), yuv420p(progressive), 480x270 [SAR
  ↪  1:1 DAR 16:9], 751 kb/s, 25 fps, 25 tbr, 12800 tbn (default)
```

Figure 3.3: `ffprobe`'s output showing the 5 video tracks of the ABR MP4 file. Some metadata was omitted for brevity.

```
ffmpeg -re -i $SOURCE \
  -map 0 -c copy \
  -utc_timing_url 'https://time.akamai.com/?iso' \
  -seg_duration 2 \
  -dash_segment_type mp4 \
  -use_template 1 -use_timeline 0 \
  -init_seg_name 'init-stream-$RepresentationID$.m4s' \
  -media_seg_name 'chunk-stream-$RepresentationID$-$Number%05d$.m4s' \
  -adaptation_sets 'id=0,streams=v id=1,streams=a' \
  -hls_playlist 1 \
  -f dash \
  $OUT_DIR/manifest.mpd
```

Specifically, the meaning of the arguments and options is as follows.

- `-re` limits the read rate so that it aligns with the frame rate. Without this option, `ffmpeg` would read the whole video without respecting the playback rate.
- `-c` with a value of `copy` tells `ffmpeg` to copy the input streams without re-encoding, since we have already encoded them previously.
- `-utc_timing_url` defines the URL of a web page that returns the current UTC timestamp in ISO format. This is required by DASH for time synchronization in the browser.
- `-seg_duration` specifies the duration of each segment/fragment that is generated. For better

performance, segments should start with a key frame, which is the reason why we set the value at 2 seconds, corresponding to the GOP length of the input file.

- `-dash_segment_size` with a value of `mp4` specifies that the packer should output CMAF-compliant `fMP4` segments.
- `-use_template` and `-use_timeline` enable the use of DASH `SegmentTemplate` to avoid listing all the video segments in the manifest. This is possible because the segments have a fixed duration.
- `-init_seg_name` and `-media_seg_name` define the file name format for the initialization segment and the individual media segments.
- `-adaptation_sets` specifies which adaptation sets should be added to the DASH manifest. The expression `id=0,streams=v id=1,streams=a` defines two adaptation sets, one with all the video streams (ID 0) and the other one with the audio stream (ID 1).
- `-hls_playlist 1` specifies that the DASH muxer should also generate HLS playlists, using the same `fMP4` segments. The packager will create a `master.m3u8` file for the master playlist and a set of media playlists with file names `media_0.m3u8`, `media_1.m3u8`, etc.
- `-f dash` sets the `ffmpeg` output muxer to DASH.
- Finally, the path of the DASH manifest file (`.mpd`) is specified.

The output of the packager are the `fMP4` files, the DASH manifest and the HLS playlists. These are placed in a Docker-mounted directory, which is shared with the CDN container.

3.3 CDN server

The "CDN" part of the setup consists of the `h2o` HTTP server, an open source software maintained by Fastly developers. `h2o` is built from source during the initial creation of the Docker images, since we are using a fork of `h2o` with some modifications.

The web server is configured to listen on three separate ports for HTTP/1.1, HTTP/2 and HTTP/3, therefore making it possible to test the three protocols independently. As we shall see, instead of relying on the `Alt-Svc` HTTP header to inform the browser about the availability of HTTP/3 on a specific port, we will configure the browser to force HTTP/3 on the corresponding port from the beginning, avoiding upgrades from other protocols.

With respect to HTTP/2, a problem we encountered was the inability to tell `h2o` to enable/disable HTTP/2 only on a specific port. However, since HTTP/2 only works over TLS, the workaround was to use port 80 for HTTP/1.1 and port 443 for HTTP/2. This approach works because this configuration only allows HTTP/1.1 to be used on port 80, while on port 443 the connection is started with TLS and HTTP/2 is used. The browser knows that it should use HTTP/2 and not HTTP/1.1 over port 443 thanks to a TLS feature, called **Application-Layer Protocol Negotiation** (ALPN), which allows negotiation of the HTTP version during the TLS handshake.

In `h2o`, HTTP/3 must instead be explicitly enabled on a specific port. In our case, we chose port 444. Figure 3.4 shows a configuration file for `h2o` that implements what we have just said.

In order for HTTP/2 and HTTP/3 to work, a public key certificate must be generated and configured on the server side so that a TLS connection can be established. The certificates were generated with the `mkcert` tool that creates a Certificate Authority (CA) and also a domain certificate signed by the root CA. This is done before starting the HTTP server. The tool makes the operation as simple as this:

```
# Generate and install the root CA certificate
mkcert -install
# Generate a certificate for the domain "cdn.local"
mkcert cdn.local
```

```
# HTTP/1.1
listen: 80

# HTTP/2
listen:
  port: 443
  ssl: &ssl
    certificate-file: certs/cdn.local.pem
    key-file: certs/cdn.local-key.pem
    minimum-version: TLSv1.2
    cipher-preference: server
    cipher-suite: "ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA...

# HTTP/3
listen:
  port: 444
  type: quic
  ssl:
    <<: *ssl

hosts:
  "cdn.local":
    paths:
      /:
        file.dir: www
      /certs:
        file.dir: certs

access-log: /dev/stdout
```

Figure 3.4: Example configuration of the `h2o` HTTP server showing HTTP/1.1 on port 80, HTTP/2 on port 443, and HTTP/3 on port 444. The directory named `www` is exposed at the domain root, while the certificates are exposed at `/certs`.

Since the certificates are regenerated at runtime every time the container is run, the client container needs to access the root CA certificate to install/trust it on the local system (otherwise HTTP/2 and HTTP/3 would not work). In practice, this is done by exposing the certificates directory via HTTP on a known path, on the `h2o` server.

3.3.1 `h2o` patches for better priorities support

Although `h2o` supports HTTP/3, during the development of the testbed we discovered that the implementation did not include support for priorities as per RFC 9218 (see Section 2.5.3). Specifically, `PRIORITY_UPDATE` frames, used to re-prioritize HTTP/3 requests, were ignored by the server.

After debugging, we found that `h2o` still implemented the first draft of the extensible priorities specification, published in 2020. Between drafts, the type and structure of the `PRIORITY_UPDATE` frame changed but `h2o` still expected the old non-RFC-compliant frame format.[26]

We therefore forked `h2o` and patched it to properly support RFC 9218. We then submitted the patch as a pull request to the upstream GitHub repository. The pull request was later merged into the `master` branch of `h2o` by the repository maintainer.[27]

Another modification that we have made to `h2o` consists in adding a way to specify the priority of the request through a query string parameter. For example: `GET /segment.mp4?priority=2`. Although this is not strictly needed, it helps in a couple of cases when HTTP requests are sent through JavaScript in the browser. We will explain the rationale in more detail in Section 4.1.

[26]`https://www.ietf.org/rfcdiff?url1=draft-ietf-httpbis-priority-01&url2=draft-ietf-httpbis-priority-02&difftype=--html`

[27]`https://github.com/h2o/h2o/pull/3096`

3.4 Emulating the network link

Between the client and the server, there is the network link that acts as the Internet access link. This link is created through the Python script that starts the emulation, found in the `topology.py` file in the testbed repository. The link connects the two switches (S1 and S2), to which the client and server hosts are connected. In ComNetsEmu, links can be created as follows:

```
net.addLink(h1, s1, bw=100, delay='0ms')
net.addLink(s1, s2, bw=100, delay='10ms')
net.addLink(s2, h2, bw=100, delay='0ms')
```

In this case, we are creating a link between H1 and S1, between S1 and S2, and between S2 and H2. We also specify the network characteristics of the links, for instance, the bandwidth (Mbps), the delay or latency (milliseconds), and the packet loss (percentage).

Note that these configuration parameters are applied to both sides of the links. So, for example, it is not possible to have asymmetric links with different bandwidth values for upload and download. Moreover, it must be considered that specifying a delay of 10 ms on a link means that the actual latency introduced by the link is 20 ms per direction, corresponding to a **Round-Trip Time** (RTT) of 40 milliseconds. However, it is possible to choose different parameters for the two interfaces of a link.

In both Mininet and ComNetsEmu, the network configuration of a link can be changed after the links are created. Specifically, the `addLink` method returns an instance of Mininet's `Link` class (usually a subclass), which exposes methods to modify the link, as shown in the following code snippet.

```
link.intf1.config(bw=bw, delay=delay, loss=loss)
link.intf2.config(bw=bw, delay=delay, loss=loss)
```

To emulate a realistic network with oscillating bandwidth and jittery latency, we needed a way to apply a **custom network pattern**. This is not something that Mininet or ComNetsEmu provide out of the box, so we implemented a strategy that consists in varying the network configuration of the link interfaces every second of the emulation, depending on a network pattern contained in a CSV dataset.

This periodic update is executed by the Python script; however, the experiments and the related network patterns are managed by the Node.js backend application hosted on H1. For this reason, we implemented an **HTTP API server** in the ComNetsEmu script so that the containers can call the API to interact with the emulation "brain". In practice, there are two endpoints that are exposed on HTTP port 8080 by the Python script:

- `POST /update`, used to update the parameters of the link between switches S1 and S2. It expects a JSON payload containing the new parameters, in particular: `bw` (in Mbps), `rtt` (in milliseconds) and `loss` (in percentage).
- `POST /stop`, which stops the emulation and all the containers. This endpoint is needed because the ComNetsEmu script does not know when the experiments are completed, so the experiment runner must communicate this through the endpoint.

3.4.1 Network patterns

To run the emulations, two main network datasets were used. In particular:

- A 4G LTE dataset captured in real-life scenarios by the *Mobile and Internet Systems Laboratory* of the University College Cork.[25];
- A synthetic set of network patterns used in the context of the *2020 Grand Challenge on Adaptation Algorithms for Near-Second Latency* organized by Twitch for the *ACM Multimedia Systems Conference*.[28]

[28]`https://2020.acmmmsys.org/lll_challenge.php`

The **"4G" dataset** contains traces collected between 2017 and 2018 from two major Irish mobile operators with different mobility patterns (static, pedestrian, car, tram and train). The dataset contains 135 traces with an average duration of 15 minutes and a throughput ranging between 0 and 173 Mbps. The traces are organized in CSV files, and each of them contains general information such as the timestamp, coordinates and speed, technical information about the mobile link such as the type of network (4G/LTE, 3G/HSPA+, etc.), SNR and RSRP, and finally the measured throughput for both downlink and uplink.

From this dataset, we extracted two 60-second sequences that show interesting patterns. We called these two patterns `lte` and `hspa+`, from the technologies that were in use during the 60-second intervals. Both traces were captured on a moving train. The `lte` pattern shows a relatively high bandwidth, although varying between about 2 and 15 Mbps. The `hspa+` pattern instead ranges from about 1 Mbps to 6 Mbps.

The patterns inspired by the **Twitch's Grand Challenge** are particularly useful when testing low-latency scenarios, since they are purposely designed to hinder adaptation algorithms. Two patterns used in our testbed were inspired from this source, specifically the `cascade` pattern, where the bandwidth is slowly reduced over time, and the `spike` pattern, where the bandwidth drops abruptly for several seconds.

3.5 Client backend and headless browser

On the "client" side of the network link there is host H1, which hosts the application that is responsible for launching the browser instance and playing the video stream in a web page. It also collects and records some metrics and events that will be later used for analysis.

For the browser, we chose to use the **Chromium** web browser, on which Google Chrome is based. We made this choice for multiple reasons, one of them being the fact that we can use a library developed by Google to control the browser by code, as we will see shortly. Other reasons are the way HTTP/3 priorities are handled (Section 4.1), the video playback behavior in case of buffer underruns[29] (Section 4.3.2), and the support for the WebCodecs API (Section 5.5.5).

Chromium is installed when building the Docker image through the default Debian 11 (bullseye) repository, which includes an ARM64 build of Chromium. However, by default Chromium only ships with non-proprietary codecs such as VP9 and AV1, and not H.264.[30] To be able to play H.264 video files, an additional package must be installed. This package (`chromium-codecs-ffmpeg-extra`) is not provided by Debian repositories, so the (equivalent) Ubuntu version is used.

Before starting Chromium, the root Certificate Authority must be trusted by the system (within the container) so that the player can download the manifest files and segments without incurring certificate errors. This can be done at startup by downloading the root CA certificate from the CDN server and then installing it system-wide, with a tool such as `certutil`. Chromium will then use the certificates database of the system and therefore trust the local CDN domain, as if the certificate was not actually self-signed.

The Chromium browser is launched as a headless[31] instance through Google's **Puppeteer**, a popular Node.js module to control Chromium through code.[32] For this reason, the backend application was written with TypeScript and is based on Node.js.

There are a few configuration parameters of Puppeteer that are important for our setup:

- Since we already installed the Chromium browser, we must tell Puppeteer not to download it at startup. This can be done with the environment variable `PUPPETEER_SKIP_CHROMIUM_DOWNLOAD` set to `true`. We can then specify where Puppeteer should find the Chromium executable with the variable `PUPPETEER_EXECUTABLE_PATH` (in our case, the value would be `/usr/bin/chromium`).

- When launching Chromium through Puppeteer, we must specify some command line options to force the use of QUIC on port 444, as mentioned above. In practice, this can be done with the

[29] A buffer underrun (or underflow) happens when the buffer is not filled fast enough and thus becomes empty.
[30] `https://www.chromium.org/audio-video/`
[31] Without the GUI.
[32] `https://pptr.dev/`

options `--enable-quic` and `--origin-to-force-quic-on=cdn.local:444`.

It is worth noting that it is not required to explicitly trust the server certificate in Chromium, since the browser already inherits the root CA certificate that we previously installed system-wide, and therefore trusts all the certificates with the CA in the chain.

An alternative way to have Chromium trust the certificate, which can be useful when developing and testing outside the testbed setup, is to use the `--ignore-certificate-errors-spki-list` option to specify the comma-separated base64-encoded SHA-256 hashes (or SPKI fingerprints) of the public key certificates to trust.

The fingerprint of a certificate can be obtained with OpenSSL like this:

```
openssl x509 -noout -pubkey -in cdn.local.pem |
  openssl pkey -pubin -outform der |
  openssl dgst -sha256 -binary |
  base64
```

An example command that launches Google Chrome on macOS, trusting a specific certificate, is the following:

```
open -a "Google Chrome" --args --enable-quic
↪   --origin-to-force-quic-on=cdn.local:444
↪   --ignore-certificate-errors-spki-list=PzvKkGfTAvrQWnmXssTywk7rHhscPwokTCMqtyg=
```

After launching Chromium, the backend application navigates to the frontend user interface, which is exposed on `localhost`. Then it starts the emulation by clicking a button in the HTML page. With Puppeteer, this can be done with the following instructions:

```
const page = await browser.newPage();
await page.goto('http://localhost:3000');
await page.waitForTimeout(2000);
await page.click('button');
```

The backend also takes care of communicating with the frontend, coordinating the emulation, and collecting and storing player metrics for subsequent analysis. Communication with the frontend is done through a `tRPC` API and through `WebSockets`. `tRPC` is a Node.js library for building type-safe APIs with TypeScript. The backend defines the **queries** (read operations) and **mutations** (write operations) that are exposed to the frontend. The frontend then uses the `tRPC` library to perform the calls.

Several `tRPC` mutations were defined, mainly corresponding to the events to which the player listens. We will see these events in more detail in Section 3.6. In addition to the events, at the start of the emulation (when Puppeteer performs the click on the start button), the frontend calls the `startExperiments` operation. The server-side handler of this operation starts looping through the set of experiments and runs them sequentially.

In a couple of cases, the emulation requires **bi-directional communication** with the frontend. An example is the backend requesting the frontend to reset the player at the end of an experiment, or to start a new playback session with a given set of parameters. At the time the testbed was developed, `tRPC`'s support for client subscriptions based on the `WebSocket` API was incomplete. Therefore, we decided to implement a custom JSON-based protocol over web sockets. Each JSON command that is transmitted over the socket contains the field `type`, plus other optional fields.

The two main types of commands transmitted over the `WebSocket` are:

- `reset`, which has the effect of destroying the player and stoping metrics collection.
- `start`, which starts a new playback session with the following parameters:

  - The ABR **protocol** (HLS or DASH).
  - The **URL** of the DASH manifest or the HLS master playlist. This parameter also determines which HTTP version is used, since different HTTP versions are enabled on the different ports.
  - The **minimum bitrate** to use in ABR playback.
  - Whether the **live catchup** feature of players should be enabled.

As mentioned above, the experiments defined in the backend are executed sequentially. Each experiment is defined by the `Experiment` class, which can be subclassed as needed to implement more specific use cases. The default implementation can be customized with the following configuration parameters:

- The **name of the experiment**, which is then used to compose the output file name that will contain the collected data.
- The name of the **network pattern** to use, which is read from a CSV file. The CSV file contains one row for each second in which the emulation will run. Each row specifies the desired bandwidth, RTT and packet loss.
- The **ABR protocol** to use for the experiment (HLS or DASH).
- The **HTTP version** (HTTP/1.1, HTTP/2 or HTTP/3).
- The **minimum bitrate** for ABR playback.
- Whether the **live catchup** feature is enabled.

When an experiment is run with the default `Experiment` implementation, a fixed sequence of operations is executed for each experiment. The duration of the experiment depends on the number of rows contained in the network pattern file. In more detail, the operations that are performed are the following.

- Read the network pattern from the CSV file and parse it.
- Reset the timer (used to assign a timestamp to each event) and empty the list of events from the previous experiment.
- Send the `start` command to the frontend so that it can configure the player and start the playback.
- Loop through the data points of the network pattern and apply the updated network link configuration once a second. The update is performed by calling the `/update` endpoint of the ComNetsEmu script, as seen in Section 3.4. The hostname of the API server is provided through an environment variable.
- At the end of the experiment, the player is reset and the list of collected events is serialized and saved to a JSON file.

3.6 Collected metrics and events

The output of each experiment is a JSON file containing the list of events that were captured during the emulation. One of these events is the `STATUS` event, which is generated by the frontend every 250 milliseconds and contains the current values of several metrics. Specifically:

- The length of the forward **video buffer**, in seconds.
- The length of the forward **audio buffer**, in seconds.

- The estimate of the **live latency**, referred to the segment currently being played.
- The **playback rate**.

Several other events are recorded. In particular:

- `BUFFER_EMPTY`, signaling that the forward buffer for the audio or video track is empty and therefore there is no media data available to continue playing the track.
- `BUFFER_LOADED`, signaling that the buffer for the specified media type now contains some data.
- `PLAYBACK_STALLED`, signaling that playback has stalled and no audio or video is being played. This is different from the buffer being empty, because playback of a specific track (e.g. audio) can in some cases continue even if the other track's buffer is empty.
- `PLAYBACK_RESUMED`, signaling that the playback has resumed;
- `FRAGMENT_LOADED`, representing the successful completion of the load of a segment. This event has some metadata attached to it, specifically:
  - the **URL** of the segment that was loaded;
  - the **media type**, i.e. video or audio;
  - the **start timestamp** of the segment within the video/audio;
  - the **duration** of the segment;
  - the **clock timestamp** of when the loading started and ended;
  - whether the segment is a **filler segment**, as we will see in Section 5.5.2.
- `REPRESENTATION_SWITCH`, signaling that a switch to a different rendition occurred. The metadata of this event are the video and audio bitrates of the new rendition.

3.7 Frontend and player

The frontend is a TypeScript application built with `Vite`, a popular tool for generating frontend bundles. It contains a simple user interface made of an HTML5 `<video>` element and a button to trigger the start of the experiments. When the button is clicked through Puppeteer, the frontend sends a `startExperiments` call to the backend through the `tRPC` client.

At startup, the frontend also creates a `WebSocket` and starts listening for commands from the backend server. As we have seen, the two commands that are transmitted over the `WebSocket` are the ones to start an experiment and to reset the player. Both these commands depend on the ABR protocol that is being used for the specific experiment, since different JavaScript libraries are used for HLS and DASH.

For both HLS and DASH, the adaptive streaming library is initialized with the manifest/master playlist URL and is attached to the HTML5 `<video>` element. Therefore, the player is the default video player provided by the browser and the HLS and DASH libraries use the Media Source Extensions API to push the video and audio data, as previously explained in Section 2.4.4.

3.7.1 `dash.js`

For MPEG-DASH playback, we used the `dash.js` JavaScript library (v4.4.1). `dash.js` is developed by the DASH Industry Forum and is therefore the official reference implementation for DASH.

Creating a new instance of the DASH player and attaching it to the HTML5 media element is simple and looks like this.

```
player = MediaPlayer().create();
player.initialize(element);
player.attachSource(url);
```

Some settings of the `dash.js` instance need to be customized for our setup. This can be done through the `updateSettings` method. For example, the code block in Figure 3.5 shows how to set some important options, in particular:

- `liveDelayFragmentCount`: a number of segments for determining the target latency of the live stream. The player will make decisions so that the requested live delay is respected. The delay can also be specified directly in seconds with the `liveDelay` option.
- `liveCatchup`: defines whether the player should increase the playback rate to catch up with the live stream when it gets behind, for example after rebuffering.
- `minBitrate`: determines which subset of video renditions will be used for adaptive streaming. Only renditions with a bitrate higher than the specified one will be selected.

```
player.updateSettings({
    streaming: {
        delay: {
            liveDelayFragmentCount: 2
        },
        liveCatchup: {
            enabled: liveCatchup
        },
        abr: {
            minBitrate: {
                video: minBitrate,
                audio: -1
            }
        }
    }
});
```

Figure 3.5: Example TypeScript code showing how to update some important `dash.js` settings.

Most of the events that the backend expects (Section 3.6) are directly exposed by the `dash.js` event emitter. In detail, the `BUFFER_EMPTY`, `BUFFER_LOADED`, `PLAYBACK_STALLED`, `PLAYBACK_RESUMED`, `FRAGMENT_LOADED`, and `REPRESENTATION_SWITCH` events are all mapped to the corresponding `dash.js` events (from which they actually take the names).

For the `STATUS` event, a JavaScript interval is created so that every 250 milliseconds the tick function is called. The buffer length (called *buffer level* by `dash.js`), the live latency, and the playback rate are collected and sent with the RPC call. Figure 3.6 shows how the status metrics are collected.

3.7.2 `HLS.js`

For HLS playback, we used the `HLS.js` library (v1.2.3). It is written in TypeScript and transpiled to JavaScript (ECMAScript5).

The most basic code snippet to initialize `HLS.js` and start the playback of an HLS stream looks like the one shown in Figure 3.7. After creating the `Hls` instance, the source is loaded and attached to the HTML5 media element. To start the playback automatically, we must listen to the `MANIFEST_PARSED` event and then call the play method on the HTML5 element, which is in practice our video player.

To customize the settings, an object literal can be passed to the `Hls` constructor. The code block in Figure 3.8 shows how to set the following options:

- `liveSyncDurationCount`: the delay from the live edge expressed in number of segments. If set to 1, it means that playback should start from segment N-1 with N being the last segment in the live playlist. For this reason `liveSyncDurationCount` corresponds to `dash.js`' `liveDelayFragmentCount` minus 1;

```
async tick() {
    if (!player.isReady()) return;
    const videoBuffer = player.getDashMetrics().getCurrentBufferLevel('video');
    const audioBuffer = player.getDashMetrics().getCurrentBufferLevel('audio');
    const latency = player.getCurrentLiveLatency();
    const rate = player.getPlaybackRate();
    await api.sendStatus(videoBuffer, audioBuffer, latency, rate);
}
```

Figure 3.6: Example TypeScript code showing how some important `dash.js` metrics, corresponding to the `STATUS` event, can be collected.

```
hls = new Hls();
hls.loadSource(url);
hls.on(Hls.Events.MANIFEST_PARSED, async () => {
    element.muted = true;
    await element.play();
});
hls.attachMedia(element);
```

Figure 3.7: TypeScript code snippet showing how `HLS.js` can be initialized and attached to a media element.

- `maxLiveSyncPlaybackRate`: similar to `dash.js`' `liveCatchup`, but the value is the maximum playback rate that can be used to catch up to the target latency of the live stream. A value of 1 means disabling live catchup;
- `minAutoBitrate`: equivalent to `dash.js`' `minBitrate`;

```
hls = new Hls({
    liveSyncDurationCount: 1,
    minAutoBitrate: minBitrate,
    maxLiveSyncPlaybackRate: liveCatchup ? 1.5 : 1,
});
```

Figure 3.8: TypeScript code showing how `HLS.js` can be initialized with custom options.

Collecting events and metrics with `HLS.js` is slightly more complex than with `dash.js`, because most events that we need are not directly exposed by the `Hls` event emitter. In practice, the only events that can be directly mapped are `FRAGMENT_LOADED` and `REPRESENTATION_SWITCH`, which respectively correspond to the events `FRAG_LOADED` and `LEVEL_SWITCHING` in `HLS.js`.

For the `STATUS` event, every 250 milliseconds a tick function is called. While the latency and playback rate are exposed as properties of the `Hls` instance, the video and audio buffer lengths must be calculated based on the current player state. To do this, we use an internal `HLS.js` function from the `BufferHelper` class, which calculates the range of the buffered media based on the MSE `SourceBuffer` (Section 2.4.4), the current time and a tolerance value (that lets us skip "holes" in the buffer that are very small). For example, the code block in Figure 3.9 shows how we can calculate the forward buffer length for the video track.

Since `HLS.js` does not directly expose the other events that we need, we must manually implement the code that infers them by observing the state of the player. For instance, to understand whether

```
const {len} = BufferHelper.bufferInfo(
    this.bufferController.tracks.video.buffer, // the source buffer
    this.element.currentTime,
    0.25 // the maximum hole duration
);
```

Figure 3.9: Code snippet showing how we can obtain the length of the forward buffer from `HLS.js`.

```
// Playback resumed (readyState >= HAVE_FUTURE_DATA)
if (this.isStalled && this.element.readyState >= 3) {
    this.isStalled = false;
    await api.sendPlaybackResumedEvent();
}
// Playback stalled (readyState < HAVE_FUTURE_DATA)
else if (!this.isStalled && this.element.readyState < 3) {
    this.isStalled = true;
    await api.sendPlaybackStalledEvent();
}
```

Figure 3.10: TypeScript code showing how playback stall detection can be implemented.

the video buffer is empty we look at the value calculated above and decide that the buffer is empty when the length goes below the threshold of 300 milliseconds (which is more or less the same that is used by `dash.js` internally). We then send the `BUFFER_EMPTY` event to the backend. Similarly, this can be done for the `BUFFER_LOADED` event, for both audio and video.

A similar approach can be used for the `PLAYBACK_STALLED` and `PLAYBACK_RESUMED` events, in this case observing the `readyState` property of the HTML5 media element. The value of this property can range between 0 and 5 and tells us how much media data is ready to be played. For example, a value of 2 (`HAVE_CURRENT_DATA`) means that the player has enough data to play only the current frame, while a value of 3 (`HAVE_FUTURE_DATA`) means that the source buffer contains data to play at least some future frames.[33] When the playback is stalled, the `readyState` of the media element will transition from 3 to 2 (or lower), allowing us to detect playback stalls (`PLAYBACK_STALLED` event). The inverse strategy can be used to detect when we should generate the `PLAYBACK_RESUMED` event. Figure 3.10 shows how this strategy can be implemented.

3.8 The glue: the ComNetsEmu script

As mentioned above, all the components of the testbed are coordinated by a ComNetsEmu script written in Python. When launched, the script performs the following operations:

- Create a `ContainerNet`, that is, a network with support for containers, a `VNFManager`, i.e. a Virtual Network Functions manager that allows to launch the containerized applications, and the SDN controller.
- Add the three hosts with hostnames `h1`, `h2`, and `h3`, with static private IPs assigned to them.
- Add the two switches, `s1` and `s2`, and create the links, as seen in Section 3.4.
- Start the network and perform a ping between `h1` and `h2` to check the connectivity between the two switches.

[33] `https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/readyState`

- Start the API server (Section 3.4) that listens on the `docker0` interface so that it can be accessed by the containerized applications.
- Update the `/etc/hosts` file of `h1` so that it can reach `h2` using the domain name `cdn.local`.
- Add the application containers to the VNF manager, making sure to pass the required environment variables and setting up the volume mounts (for example, the live source and the CDN containers must share the folder where the segment data is written/read).
- Wait for the API server to emit a stop event (triggered when the `/stop` endpoint is called) and then stop all the containers and the network; the event is propagated through different modules with an instance of the `threading.Event` Python class.

3.9 Analysis and visualization with R

After each experiment is run in an emulation, the collected events are serialized to a JSON file as an array. Each event is characterized by the timestamp, the type, and the optional metadata. This data is then used to generate some plots and indicators that help us to understand how the playback session performed in terms of quality of experience.

To parse, analyze, and visualize the data, we decided to use R, the programming language for data science. All plots were generated with the `ggplot2` library, but we also used `dplyr` and other packages from the `tidyverse` to prepare the data for visualization.[34]

The visualization mainly consists of 3 plots:

- The **buffer health plot**, which shows how the length of the client buffer varies over time, for both the video and audio tracks. This plot also shows the network bandwidth available at any given moment and the current total bitrate of the media. It also highlights buffer empty, buffer load, playback stalled and playback resumed events.
- The **live latency plot**, which shows how the live latency varies during the playback session. This plot also shows the playback rate (useful when evaluating the live catchup scenario) and the buffer/playback events mentioned above.
- The **waterfall diagram** of the requests, showing when the individual segments were loaded during the playback session and how long the loading took.

A more detailed explanation of these plots will be provided in Chapter 4 when analyzing the results of some experiments.

[34] `https://www.tidyverse.org/packages/`

Chapter 4

Quality of experience evaluation

In this section, we will use the testbed presented in the previous chapter to execute some **experiments** and highlight some of the issues of current streaming technologies in low-latency scenarios. We will present how the data collected through the testbed is visualized and show some of the most important results and observations.

The results of the evaluation will be the basis for the improvements proposed in Chapter 5.

4.1 Different browsers behaviors

An important thing to keep in mind when evaluating the performance of streaming protocols is that different browsers have different implementations of the network protocols and can therefore show different behaviors. One of these differences relates to how **HTTP priorities** are handled.

4.1.1 HTTP priorities handling in Chromium and Firefox

According to our tests, Chromium (v107) allows to specify the `Priority` header in requests sent with the `fetch` API or with `XMLHttpRequest`. The header is in fact passed as is to the HTTP server and is correctly interpreted by servers supporting the extensible priorities specification (Section 2.5.3). However, the browser also sends a `PRIORITY_UPDATE` frame after sending the HTTP request, overriding the priority specified through the header. The `PRIORITY_UPDATE` frame is sent as part of the prioritization strategy implemented by the browser, which applies its own heuristics to determine which requests should be prioritized, both internally in the browser and at the network level.

The **Priority Hints** feature provides a way to control the priority that the browser gives to individual HTTP requests. The Priority Hints specification is still a W3C draft, but Chromium already implements it. The priority values defined by the specification are `high` and `low`.[35] By default, `fetch` API requests have a priority of `high`, which corresponds to HTTP/3 priority `1` (where `0` means highest priority), transmitted in the `PRIORITY_UPDATE` frame. If the `fetch` priority is set to `low`, it turns out that Chromium does not send the `PRIORITY_UPDATE` frame at all, effectively leaving the priority value at the default value of `3`. As a consequence, in practice, setting the fetch priority to `low` enables setting a custom priority through the `Priority` header. Figure 4.1 shows some examples of how priorities are currently handled in Chromium.

In summary, **in Chromium it is possible to set custom priority values** by using the `Priority` header, the Priority Hints feature, or a combination of the two.

In Firefox (v107), the behavior is different. First, Firefox does not support Priority Hints. It also appears that it does not use the `PRIORITY_UPDATE` frame, instead adding the `Priority` header internally to outgoing requests. By default, the urgency is set to `4` for `fetch` requests. However, manually setting the `Priority` header does not seem to have any effect, as the header is always overridden by Firefox. In practice, this means that **until Firefox adds support for Priority Hints there does not seem to be a way to send a custom priority**.

[35] `https://wicg.github.io/priority-hints/`

```
await fetch('/');
// Priority hint   => default => high
// Priority header => not sent
// PRIORITY_UPDATE => sent with urgency 1

await fetch('/', { headers: { Priority: 'u=2' } });
// Priority hint   => default => high
// Priority header => sent with urgency 2
// PRIORITY_UPDATE => sent with urgency 1

await fetch('/', { priority: 'low' });
// Priority hint   => low
// Priority header => not sent
// PRIORITY_UPDATE => not sent, default to urgency 3

await fetch('/', { priority: 'low', headers: { Priority: 'u=2' } });
// Priority hint   => low
// Priority header => sent with urgency 2
// PRIORITY_UPDATE => not sent
```

Figure 4.1: Summary of the observed behaviors when trying to set custom HTTP/3 priorities in Chromium.

4.1.2 The issue with CORS

A problem that might arise when using the `Priority` header, common to all browsers, is related to how **cross-origin requests** are handled.

A cross-origin request is a request generated by a web page that has a destination whose domain, scheme, or port do not match those of the current web page. For security reasons, browsers require that the server explicitly authorizes the origin to execute cross-origin requests when they might have undesired effects on user data. This is done through CORS, which consists in sending an additional preflight request before the actual HTTP request so that the server can specify which requests are allowed (through response headers).

The presence of the `Priority` header triggers a preflight request before each HTTP request if the request is cross-origin (which is often the case when using a CDN). Unfortunately, this additional request can introduce significant latency.

4.1.3 The h2o patch

To work around the issue of not being able to specify the HTTP/3 priority in some cases, and more importantly the CORS issue, we implemented a patch to `h2o` so that the priority can be forced with a query string parameter.[36] So, for example, a request like `GET /segment.m4s?priority=2` would give the request an urgency value of 2.

4.2 Experiments and results with non-ABR setup

As a first experiment with our testbed, we investigated how video streaming behaves when the network bandwidth is unstable and the bitrate is fixed, i.e. no bitrate adaptation is carried out. As we shall see, although this situation is not realistic, it allows us to better observe the different behaviors between HTTP versions.

We based this first analysis on **MPEG-DASH** with the `dash.js` library, with a **target live latency of 4 seconds**, corresponding to 2 segments. The bitrate of the video rendition is the highest one, i.e. 3.5 Mbps.

[36] `https://github.com/matteocontrini/h2o/commit/bf307ef`

For this setup, we used network patterns from the 4G dataset. Specifically, the `lte` and `hspa+` patterns, as seen in Section 3.4.1. The average RTT is set to 80 ms with a jitter of 15 ms.

For each of these two patterns, we tested how the system behaves with HTTP/1.1, HTTP/2 and HTTP/3. For HTTP/3, we also tested with the live catchup feature enabled. In practice, this means that 8 experiments were defined, as shown in Figure 4.2.

```
const experiments = [
  new Experiment('lte_h1', 'lte', ABRProtocol.DASH, HttpVersion.HTTP1_1, 3000),
  new Experiment('lte_h2', 'lte', ABRProtocol.DASH, HttpVersion.HTTP2, 3000),
  new Experiment('lte_h3', 'lte', ABRProtocol.DASH, HttpVersion.HTTP3, 3000),
  new Experiment('hspa+_h1', 'hspa+', ABRProtocol.DASH, HttpVersion.HTTP1_1, 3000),
  new Experiment('hspa+_h2', 'hspa+', ABRProtocol.DASH, HttpVersion.HTTP2, 3000),
  new Experiment('hspa+_h3', 'hspa+', ABRProtocol.DASH, HttpVersion.HTTP3, 3000),
  new Experiment('lte_catchup', 'lte', ABRProtocol.DASH, HttpVersion.HTTP3, 3000, true),
  new Experiment('hspa+_catchup', 'hspa+', ABRProtocol.DASH, HttpVersion.HTTP3, 3000, true)
];
```

Figure 4.2: TypeScript code showing how the list of experiments is defined in the testbed.

4.2.1 Suboptimal behavior over HTTP/3

As a first step, we will look at the results of the experiment named `lte_h3`, which uses DASH over HTTP/3 with the `lte` dataset.

The **buffer health plot** is shown in Figure 4.3. In detail, the elements of the plot have the following meaning:

- The gray area in the background represents the **network bandwidth**, with the Mbps scale shown on the right side. As we can see, it varies between about 2 and 16 Mbps over the duration of the experiments, which is approximately 60 seconds with this network pattern.
- The orange line represents the **bitrate of the media**, calculated as the sum of the video and audio bitrates. In this case, it is constant at about 3.5 Mbps.
- The black line indicates the **size of the video buffer in seconds**. The scale is shown to the left of the panel.
- The purple line instead refers to the **audio buffer size**.
- The vertical dotted lines refer to the **buffer events**. The red lines correspond to empty buffer events (for any track), while the green lines indicate buffer loaded.
- The red areas between the dotted lines represent the **playback stalls**. The left border of a red area corresponds to a playback stalled event, while the right border corresponds to playback resumption.

What we can observe from the plot is that for the first 30 seconds the bandwidth is generally above the bitrate of the media, and therefore the buffer can be filled properly most of the time. Another observation that we can already make is that the audio buffer is sometimes filled with more seconds of data than the video buffer. This happens because **the video and audio tracks are unmuxed and can therefore be downloaded independently** by the player/browser. Depending on the HTTP response scheduling, the audio segment might arrive before the video segment, or vice versa.

In the first half of the experiment, we can also notice a couple of cases where the player struggles to fill the video buffer in time. For example, just after second 20 the video buffer actually becomes empty for a very short period of time. As we will see later, in practice this kind of event does not necessarily produce a playback stall due to the way Chromium handles empty buffers.

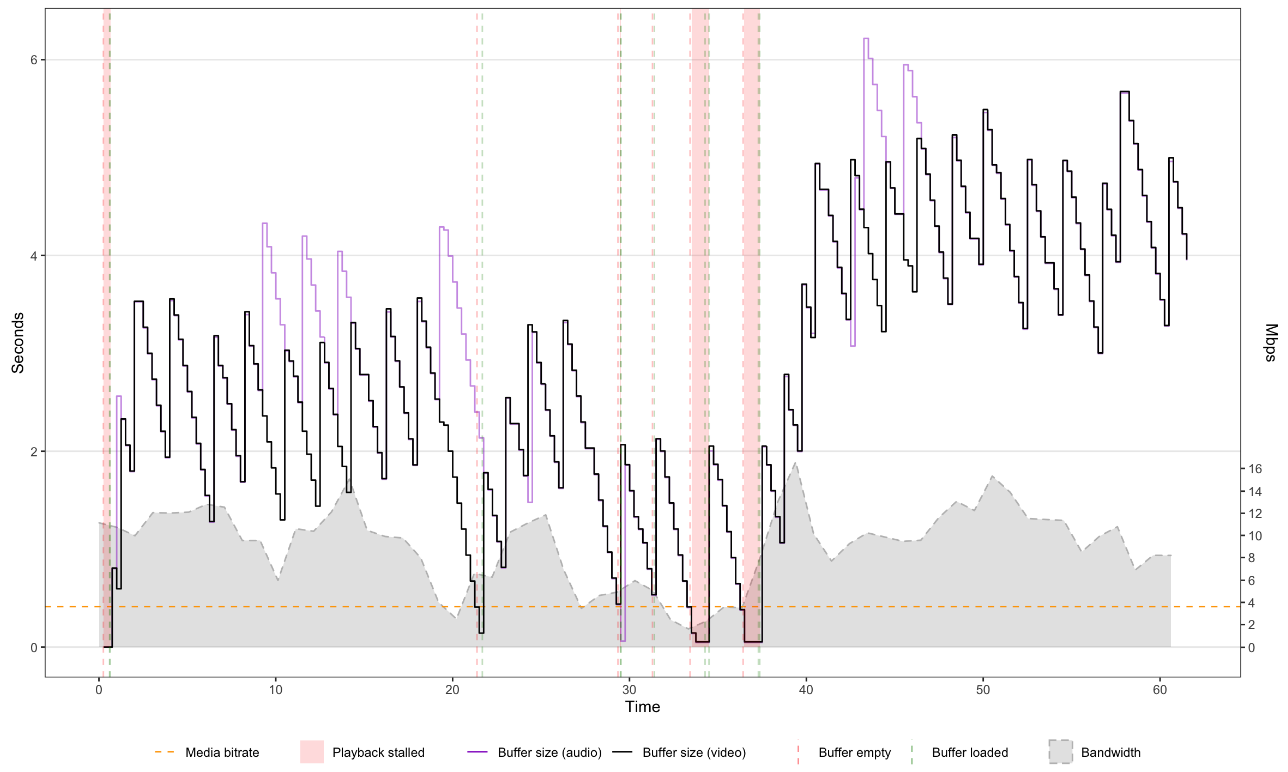


Figure 4.3: Buffer health plot for DASH with the `lte` network pattern over HTTP/3.

At about second 30, the bandwidth starts to become very limited and goes below 3.5 Mbps for a few seconds. The video and audio buffers quickly deplete, empty buffer events are emitted, and the playback stalls. This happens twice, as can be seen by the two red areas.

When the network bandwidth grows again, the buffers are filled. One thing to note is that the buffer length now contains about one segment more on average (it is in fact closer to 4 seconds than to 2). This happens because the delay introduced by the stall caused the playback to get behind with respect to the edge of the playlist, and therefore there is now an additional segment that can be loaded in advance.

The effect of this behavior can be better seen in the **live latency plot**, shown in Figure 4.4. In correspondence of the playback stalls, the live latency grows from about 4.5 to 5.5 seconds, and then to 6.5 seconds (in total, a 2-second segment more).

Finally, if we observe the **waterfall diagram** (Figure 4.5), an interesting and perhaps unexpected behavior can be observed. Intuitively, we would expect requests for audio segments to always take less time than video segments. In fact, while a single video segment can be as large as 6-700 KiB, the audio segment is usually about 30 KiB in size. This means that **audio is approximately between 10 and 20 times smaller than video**. However, the waterfall shows that, while sometimes the loading of the audio segment takes very little time, **the loading of the audio segments often takes as long as the video**.

This behavior hints at a suboptimal scheduling strategy of HTTP requests over the QUIC connection, so we decided to investigate. We used the **NetLog dump** feature of Chromium (Section 2.5.3) to generate an export of network traffic, which also contains detailed information about QUIC connections and streams. We then analyzed the dump file with `qvis`.

One of the visualization tools provided by `qvis` is the multiplexing view, where the multiplexing of the streams at the QUIC level is represented both as a flow/timeline and as a waterfall (Figure 4.6). The flow representation shows the sequence of QUIC packets that the client received. Each packet is colored with a different color depending on the stream to which it belongs. The waterfall view shows the same data as a waterfall chart, where each stream has its own row, and the horizontal segments represent the time intervals in which the stream was actively being transferred.

In our case, what we can clearly see in the multiplexing diagrams, especially in the waterfall, is

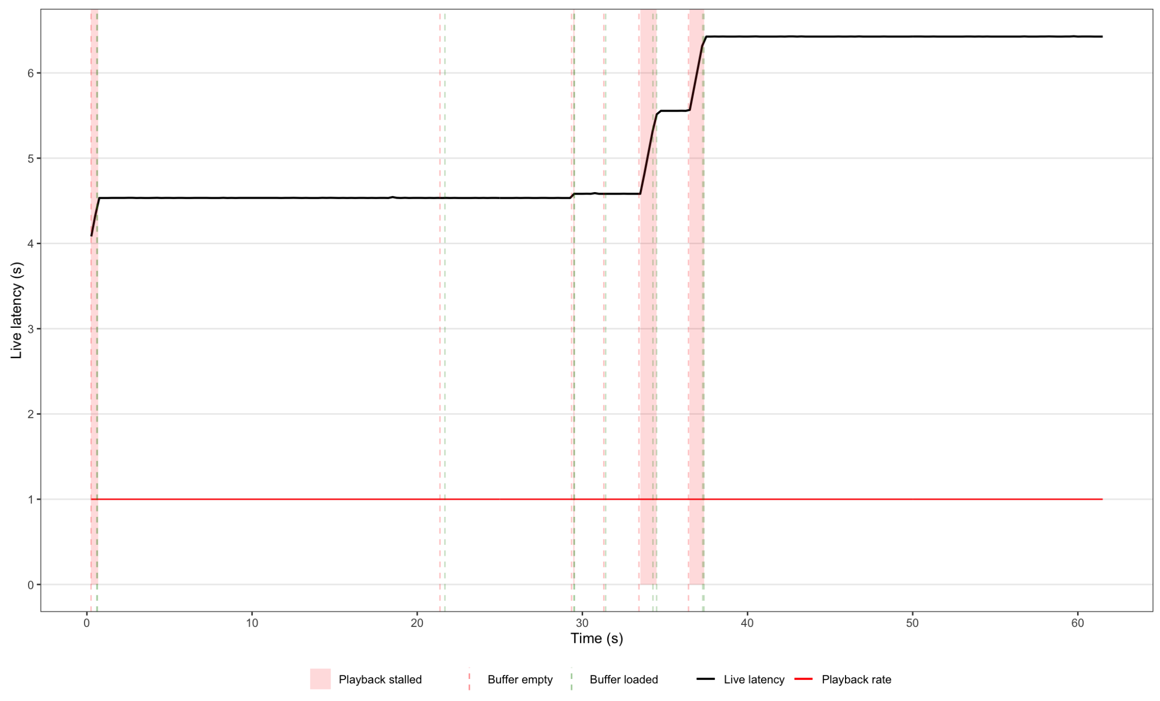


Figure 4.4: Live latency plot for DASH with the `lte` network pattern over HTTP/3.

that the streams are transmitted almost sequentially. In fact, there is almost no preemption, and most streams get to use the entire channel for transmission until all data is received.

However, an interesting fact to observe is that the data flow is not entirely sequential. As Figure 4.7 shows, there is a recurring pattern of streams/requests that start and are immediately suspended to give bandwidth to another stream on the QUIC connection.

This behavior explains what we were seeing in the segments requests waterfall in Figure 4.5. Requests for audio and video segments often start at the same time; however, the transfer of the audio segment is almost immediately preempted to give priority to the video segment. The audio segment transfer is then completed after the video segment is completely downloaded.

This behavior is not ideal, because an audio segment is always much smaller than a video segment and could therefore benefit from being received by the client earlier, as we will see in the next section. The next section will also compare how this same setup performs with HTTP/2 and HTTP/1.1, showing that the scheduling scheme that was observed seems to be specific to HTTP/3, or at least to the HTTP/3 implementation that we are relying on in this testbed.

The takeaway result of this analysis is that **we cannot rely on HTTP/3 implementations to always know the best strategy for request prioritization**. Instead, it is also the client/browser's responsibility to implement optimizations so that the prioritization strategy is meaningful.

The prioritization strategy could derive from browser heuristics that are applied globally for all websites, as we mentioned in Section 2.5.2, but it could also be explicitly requested by developers with the use of priorities, as we explained in Section 4.1.1.

4.2.2 Comparison across HTTP versions

As we have seen, the emulation over HTTP/3 shows that priorities are not consistent, even in the same run. We will now see how HTTP/2 and HTTP/1.1 behave and then compare the results.

Let us start from **HTTP/2**. Figure 4.8a shows the buffer health plot for an experiment execution based on HTTP/2. The rest of the settings remained the same. As we can easily see, the behavior is quite different from HTTP/3. First, the audio buffer is always larger than the video. This means that the audio segments are consistently loaded before the video buffer. The waterfall in Figure 4.8b confirms that this is because the audio segments are loaded much faster, as we would expect, and the loading of the video segments does not delay the audio. This behavior probably derives from a different prioritization strategy, implemented by the browser or driven by the HTTP/2 server implementation.

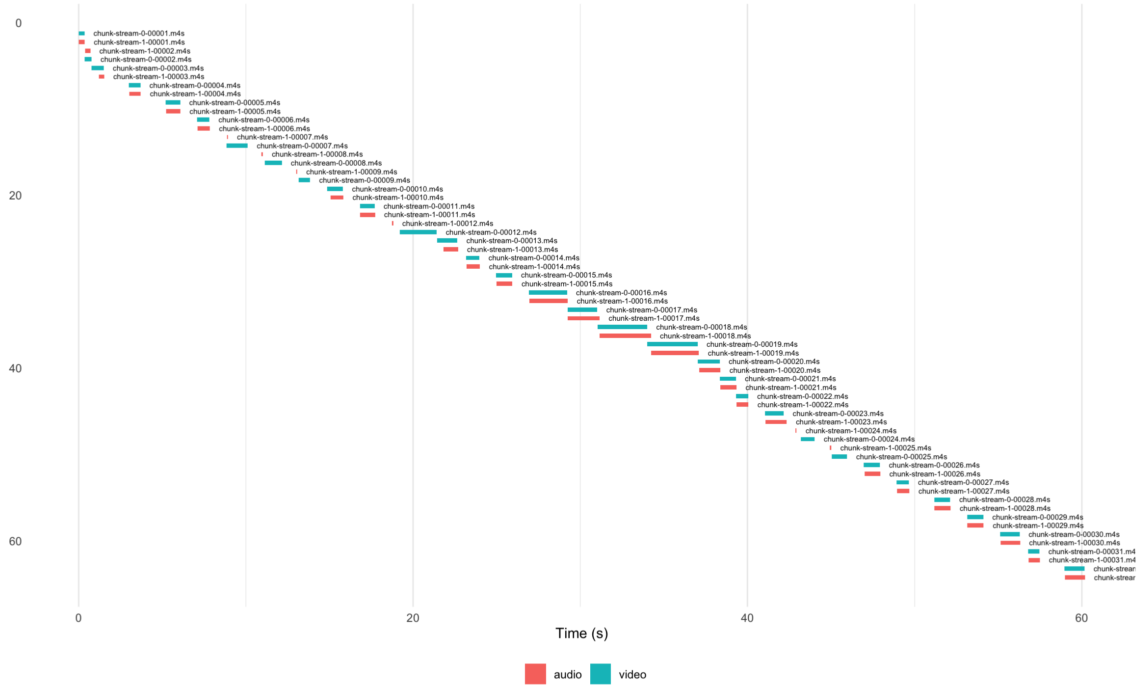


Figure 4.5: Waterfall diagram for DASH with the `lte` network pattern over HTTP/3.

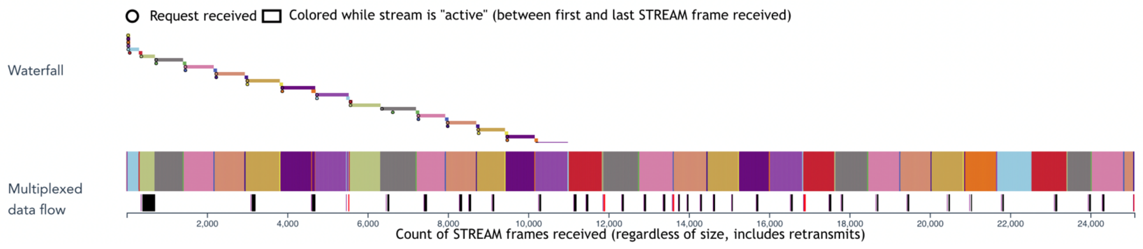


Figure 4.6: `qvis` multiplexing diagrams for the MPEG-DASH over HTTP/3 experiment.

The other thing that we can observe is that there are no playback stalls, apart from the initial loading, even if the video buffer is almost empty for several seconds in the middle of the emulation. This behavior derives from the fact that the browser is capable of playing the audio data even if the video is not available, for up to 3 seconds. This particular **buffer underflow behavior** is specific to Chromium and is officially documented.[37] Obviously, it can only work when audio and video data is appended to the MSE `SourceBuffer` independently; therefore, in practice, it can only happen when using unmuxed video and audio tracks, as is in our case.

Although Chromium's underflow behavior might not be the expected experience when streaming on-demand content, since it has the effect of not rendering the video for a few seconds, it seems instead to be particularly suitable for live streaming. In fact, in this way **short time intervals where video data is not available do not cause a stall**, and, therefore, **do not increase the latency of the live stream**. While the video is frozen due to a slow loading segment, **the audio will continue to play**.

Moving on to **HTTP/1.1**, the behavior that we observed is somewhat similar to HTTP/2. As Figure 4.9a shows, the audio buffer is virtually always longer than the video buffer. One thing to note is that there is a playback stall at about 38 seconds, even if the audio buffer contains several seconds of data. This is due to the limitations of the underflow tolerance we have just introduced: Chromium only allows the video track to have missing data for about 3 seconds, after which the playback will stall. We will address this limitation in Chapter 5.

The waterfall diagram for HTTP/1.1, shown in 4.9b, is perhaps even more interesting. It shows

[37] `https://www.chromium.org/audio-video/#how-the-does-buffering-work`

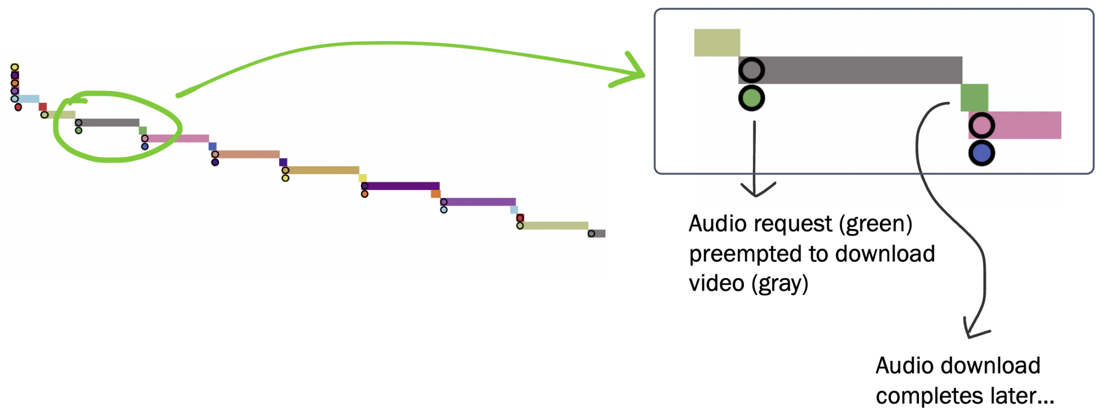
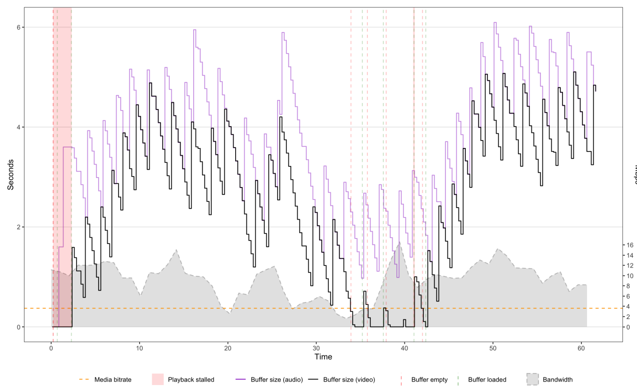
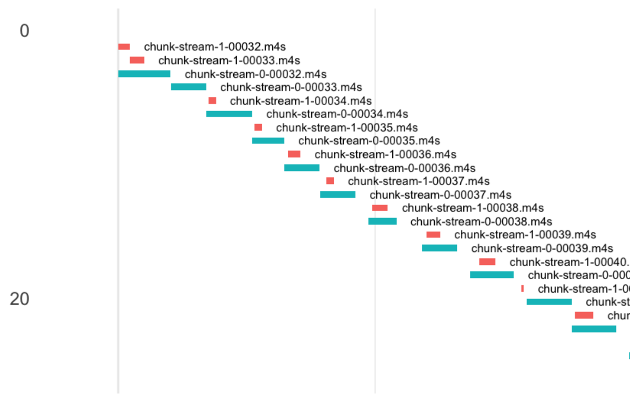

Audio request (green) preempted to download video (gray)
Audio download completes later...

Figure 4.7: `qvis` waterfall diagram showing unexpected prioritization of video segments requests over audio with HTTP/3.



(a) Buffer health plot.



(b) Waterfall diagram.

Figure 4.8: DASH experiment over HTTP/2 with the `lte` network pattern.

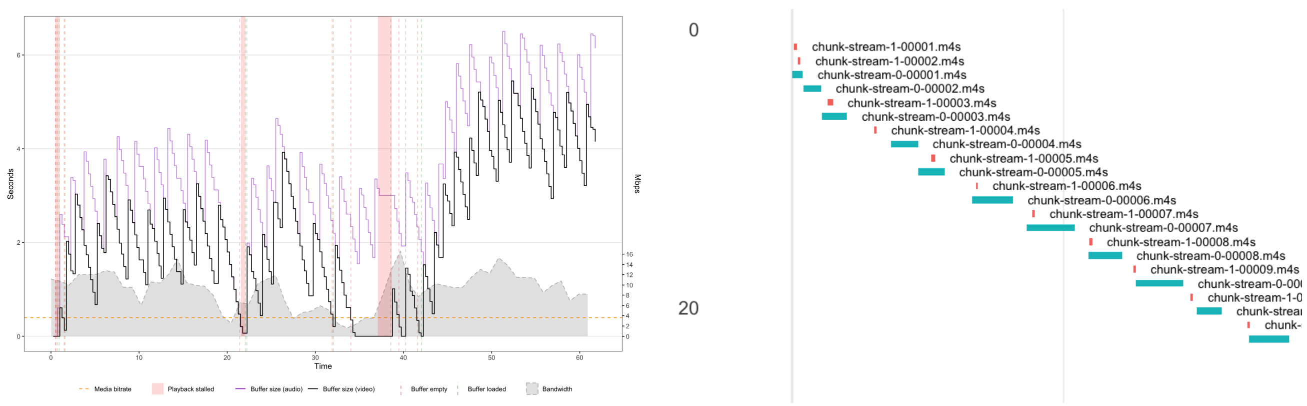
that the download of the audio segments, represented in red, is consistently completed in a very short amount of time. Intuitively, the explanation for this phenomenon is that HTTP/1.1 can use multiple independent TCP connections, not performing multiplexing at all. In this way, requests can be distributed among the connections and benefit from a "dedicated" transmission channel.

These results are surprising. We are seeing that HTTP/2 and HTTP/1.1 appear to offer better scheduling of requests. Incidentally, this also translates into a live streaming experience that has fewer "hard" stalls and where the latency does not increase.

4.2.3 The need for adaptive bitrate streaming

The experiments mentioned above were conducted with the `lte` network pattern, where the network bandwidth is most of the time higher than the media bitrate. In the `hspa+` pattern, the situation is different: the bandwidth is just slightly above the media bitrate and often goes below it. This is pretty clear from Figure 4.10a, where there are many stalls in all situations where the bandwidth is not enough. The consequence is that the live latency increases during the playback stall periods, reaching 23 seconds as shown in Figure 4.10b.

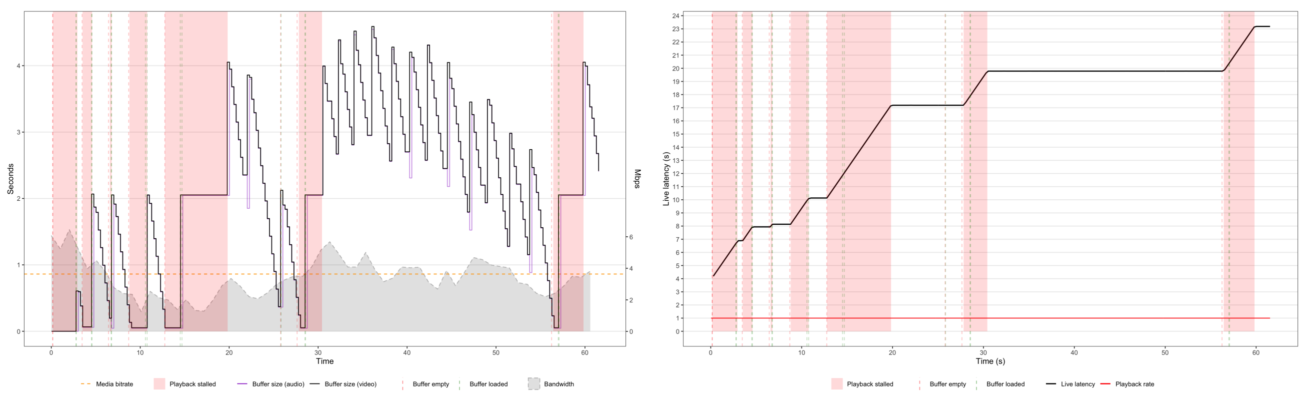
Adaptive bitrate streaming is needed for this reason: when the adaptation algorithm estimates that there is not enough bandwidth to continue the playback at the current bitrate, it will make the decision to switch to a lower bitrate. In the next sections, we will see how the system performs when adaptive bitrate streaming is enabled.



(a) Buffer health plot.

(b) Waterfall diagram.

Figure 4.9: DASH experiment over HTTP/1 with the `lte` network pattern.



(a) Buffer health plot.

(b) Live latency plot.

Figure 4.10: DASH experiment over HTTP/3 with the `hspa+` network pattern.

4.3 Experiments and results in an ABR setup

The testbed that we introduced in Chapter 3 is already capable of running experiments on an adaptive live stream. In fact, if we do not specify the minimum bitrate in the experiment configuration, the playback will use ABR. We tested this setup with both `dash.js` and `HLS.js`.

4.3.1 Suboptimal behavior of the default `dash.js` configuration with live

An experiment we ran involved using `dash.js` with ABR enabled and on the `lte` network pattern. As a reminder, the bitrate ladder we are using in this experiment is made of four resolutions (720p, 540p, 360p, 270p) with video bitrates of 3.5 Mbps, 2.5 Mbps, 1.5 Mbps and 0.8 Mbps.

The buffer health plot in Figure 4.11 shows that with this configuration buffer underflows and playback stalls are essentially eliminated, compared to Figure 4.3 (non-ABR case) where there were multiple stalls. This is due to the fact that the default configuration of `dash.js` is relatively aggressive in downswitching the resolution/bitrate. Therefore, it is able to quickly react to the worsening network conditions at about 30 seconds, and lower the bitrate to 2.5 and then 1.5 Mbps. A similar result (not shown here) is obtained with the `hspa+` pattern.

There is, however, an evident disadvantage to this aggressive approach, which can be seen between second 40 and second 60 in the plot. The bandwidth is always above 8 Mbps in that interval of the experiment, and thus there should be more than enough bandwidth to keep the stream at the highest quality and bitrate (3.5 Mbps).

Instead, what we observe is that the adaptation algorithm keeps switching quality in a way that does not seem to make much sense. At second 40, the bitrate is raised to 3.5 Mbps but almost immediately switched down to 2.5 Mbps. The algorithm then tries to raise the bitrate again and immediately falls to 1.5 Mbps. This oscillating pattern is repeated a few times until the experiment

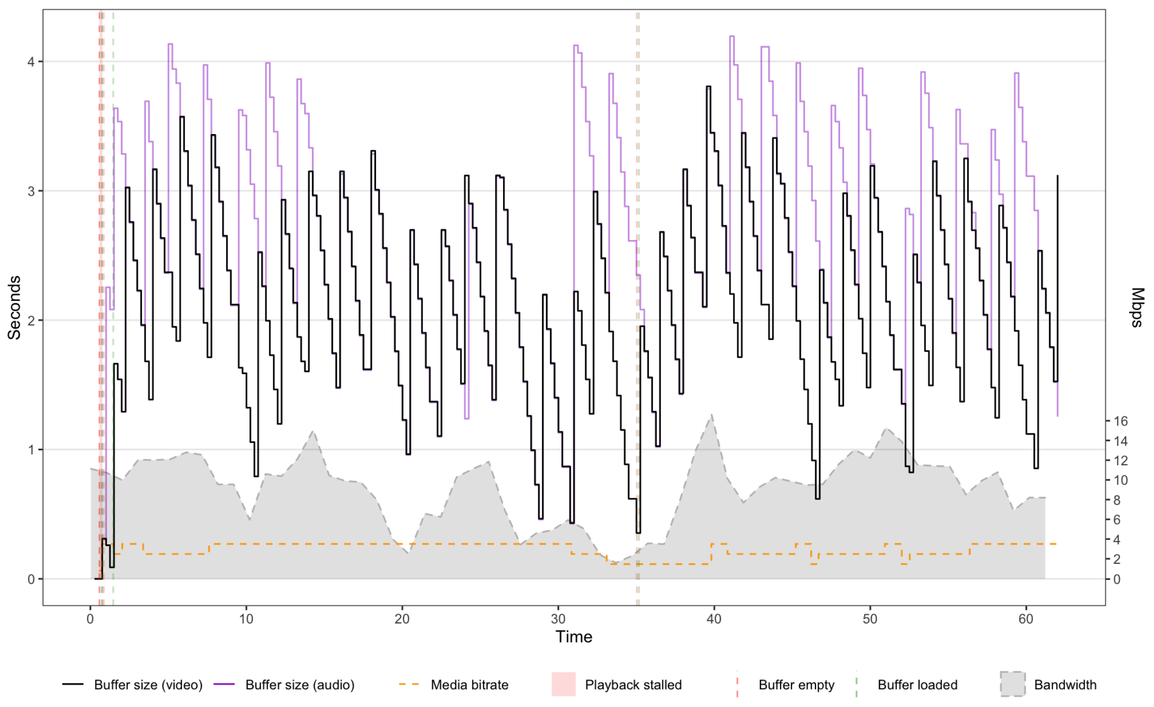


Figure 4.11: Buffer health plot of an ABR DASH experiment with default configuration over HTTP/3 with the `lte` network pattern.

is over.

We observed that this behavior is consistent between different runs of the same experiment and is therefore something that should be investigated.

4.3.2 Other network patterns and experiments with `HLS.js`

To find a situation where the adaptation algorithm does not behave well and leads to playback stalls, we relied on new network patterns inspired by Twitch's dataset for low-latency scenarios, introduced in Section 3.4.1. Specifically, for this analysis we will take into consideration the `spike` pattern, which contains an abrupt drop in bandwidth that lasts for a few seconds. The duration of the experiments based on this pattern is about 30 seconds.

At this point of the analysis, we also had a complete implementation of the integration with `HLS.js` in the testbed, and therefore we ran the experiments with both `dash.js` and `HLS.js`.

Figure 4.12 shows the buffer health plot of an experiment run with `HLS.js` and the `spike` network pattern over HTTP/3. As can be seen, when the bandwidth suddenly drops from about 4 to 1 Mbps, the adaptation algorithm struggles to react in time, causing a couple of playback stalls and thus increasing latency. An observation we could make is that if the audio had priority over video, the playback stalls would be avoided since the stalls are shorter than 3 seconds. Section 5.3 will show how this can actually be implemented.

Finally, we also tested the behavior of the system when the **live catchup** feature is enabled. The live catchup feature increases the playback rate when the measured live latency is greater than the target one, up to a specified limit. In our tests, we used a maximum speed up of 1.5x. The results (not shown here) show that the live catchup indeed reduces the average latency, however seemingly causing additional playback stalls. The other obvious disadvantage is that for many types of video content having an increased playback rate is not an ideal experience.

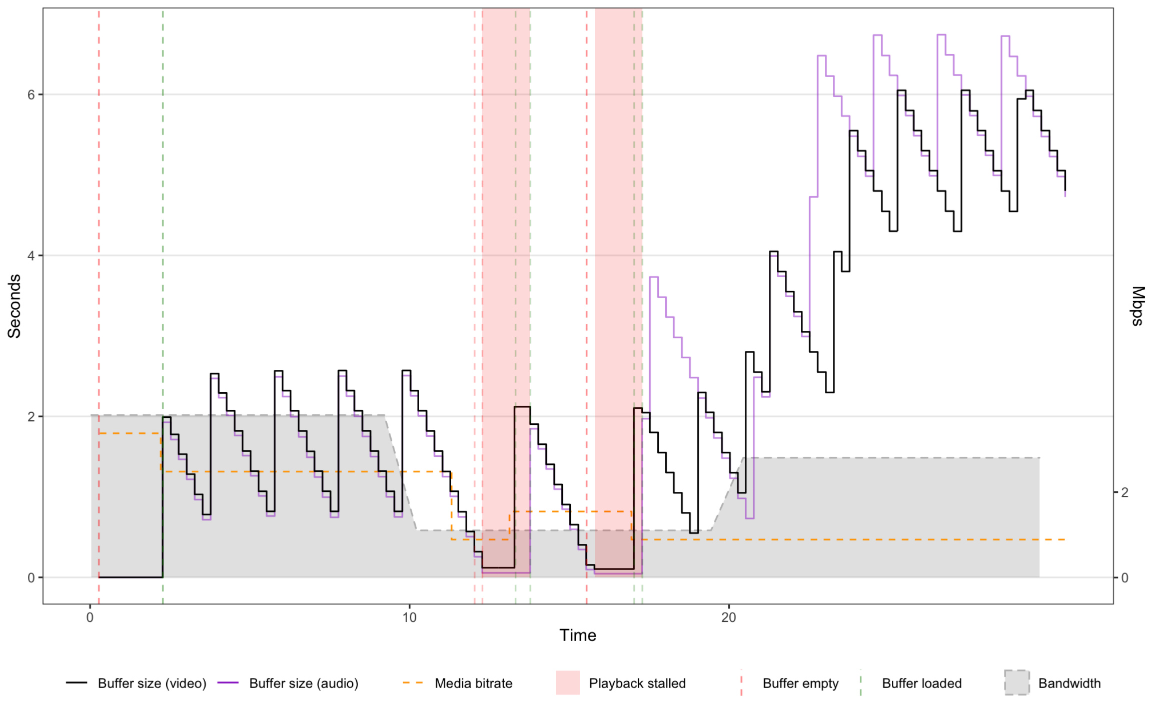


Figure 4.12: HLS experiment over HTTP/3 with the `spike` network pattern.

Chapter 5

Proposed improvements and results

In this chapter, we will present some **improvements** to live streaming protocols and libraries that we believe produce a **better overall quality of experience** in video streaming. The proposed improvements are based on the findings presented in the previous chapter.

Specifically, we will first dive into the adaptation algorithms of `dash.js` to assess that the default configuration is not suitable for low-latency live streaming. Then, we will propose a better configuration which also involves writing a custom ABR rule.

We will then look at how we can take advantage of HTTP/3 features, such as priorities and stream resetting, to implement a new live streaming approach where video playback stalls are avoided if the video buffer is temporarily empty. We will do so with a combination of HTTP prioritization and on-the-fly generation of filler segments on the client side, thanks to WebCodecs API and WebAssembly. For this part, we will modify the `HLS.js` library.

Finally, we will show how the proposed solution produces fewer playback stalls and avoids the increase of live latency in the case of network congestion, with an overall smoother live streaming user experience.

5.1 `dash.js` bitrate adaptation algorithm improvements

In Section 4.3.1 we have shown how in some cases the default configuration of `dash.js` makes the video quality oscillate too frequently with a low-latency live stream. Specifically, Figure 4.11 represents the buffer health plot, where we can see that in the second half of the experiment the media bitrate chosen by `dash.js` frequently changes and most of the time stays at a lower value than it should, even if the network bandwidth is consistently above 8 Mbps.

The decision on which bitrate should be requested next is the task of the **rate adaptation algorithm**, which every library typically implements differently. There are in fact many approaches to solving the problem and there is no clear consensus on which is the best.

The core idea of adaptation algorithms is to choose the best quality level so that the bitrate is lower than the available network bandwidth. There are two main approaches to make this decision:

- **Throughput-based**: the most intuitive approach, consisting of continuously calculating an estimate of the network bandwidth and then selecting the highest bitrate in the ladder which is lower than the bandwidth estimate.
- **Buffer-based**: an alternative approach that observes the occupancy of the buffer to determine whether the decision to increase/dicrease the bitrate should be made. This approach avoids the need to keep a bandwidth estimate.

`dash.js` uses a combination of these two approaches with a strategy called `DYNAMIC`, which was first introduced in [28]. The `DYNAMIC` strategy switches between a throughput-based approach, named `THROUGHPUT`, and a buffer-based approach, `BOLA`.[29] The idea of switching between the two is based on the fact that each algorithm works best in different situations. In particular, the throughput-based algorithm performs better when the buffer is low or empty, while `BOLA` (buffer-based) performs better when the buffer level is sufficiently large.

The algorithm that switches between THROUGHPUT and BOLA is therefore based on a threshold that is applied on the buffer level (for each media track). The following code snippet taken from the dash.js source code (AbrController class) shows how the decision on whether to use BOLA is made:

```
1  const useBufferABR = isUsingBufferOccupancyAbrDict[mediaType];
2  const newUseBufferABR = bufferLevel > (useBufferABR ? switchOffThreshold :
   ↪  switchOnThreshold);
3  isUsingBufferOccupancyAbrDict[mediaType] = newUseBufferABR;
4
5  if (newUseBufferABR !== useBufferABR) {
6      if (newUseBufferABR) {
7          logger.info('[' + mediaType + '] switching from throughput to buffer
           ↪  occupancy ABR rule');
8      } else {
9          logger.info('[' + mediaType + '] switching from buffer occupancy to
           ↪  throughput ABR rule');
10     }
11 }
```

The relevant part is line 2, where the new decision on whether buffer-based ABR should be used is made based on whether the current buffer level (in seconds) is greater than a threshold.

The switchOffThreshold/switchOnThreshold are calculated from the value of a property called stableBufferTime. Basically, when the buffer level exceeds a threshold value that is considered stable, the DYNAMIC algorithm switches to the buffer-based approach (BOLA), while going back to THROUGHPUT when the buffer level is less than half of the stableBufferTime.

```
const switchOnThreshold = stableBufferTime;
const switchOffThreshold = 0.5 * stableBufferTime;
```

The aim is to avoid continuous oscillations between the two strategies, although this does not work well in practice according to our tests. In fact, in the case of a live stream, stableBufferTime is assigned to the live delay target (the configuration parameter we introduced in Section 3.7.1). So, for example, if the live delay is 4 seconds, this means that the DYNAMIC algorithm will switch from THROUGHPUT to BOLA when the buffer level goes above 4 seconds, and will switch back to THROUGHPUT when the buffer level goes below 2 seconds.

The problem is that with such a low target live latency, the buffer level tends to oscillate quite a lot and often goes below the threshold of 2 seconds, as can be seen in Figure 4.11 as an example. In practice, this means that the default DYNAMIC ABR strategy of dash.js will switch between the two approaches very often. This behavior is probably undesirable, as it does not leave time for the algorithm to make proper decisions.

This behavior is also in contrast with the threshold used by the paper that introduced the DYNAMIC strategy and tested its performance. In fact, in [28] the threshold value is fixed at 10 seconds. Therefore, following the recommendation of [28] for low-latency streams we decided to disable the DYNAMIC strategy and force the THROUGHPUT one. This can be done in dash.js through the settings:

```
player.updateSettings({
    streaming: {
        abr: {
            ABRStrategy: 'abrThroughput'
        }
    }
});
```

While we were looking at the internals of `dash.js` to investigate the issue with `DYNAMIC`, we discovered that the library includes support for **additional ABR rules** that should help in cases when the ABR strategy is not enough to make proper decisions about the next bitrate. These rules are by default:

- `DroppedFramesRule`: measures the dropped frames during playback. If they exceed 15%, the rule switches to a lower bitrate.
- `SwitchHistoryRule`: observes the switch history and avoids switches to higher bitrates that caused quality drops in the past, according to the switch history.
- `AbandonRequestsRule`: calculates whether the loading of a segment should be canceled because it is estimated that it will not be loaded in time for playback, only if there is a lower bitrate that could instead be loaded in the remaining time. This rule acts as an emergency switch down.
- `InsufficientBufferRule`: this rule puts an upper limit on the bitrate, depending on the current buffer health.

In particular, the `InsufficientBufferRule` was found to be the most aggressive in lowering the bitrate even when it was not needed. This happens because it bases the decision on the buffer level, which, as we have seen, is always quite small in a low-latency scenario. More in detail, the `InsufficientBufferRule` listens to the `BUFFER_EMPTY` event and requests a switch to the minimum bitrate when that happens. Otherwise, the rule calculates the maximum bitrate (to be used as the limit) depending on the throughput, so that a whole segment can be downloaded before the buffer runs out. The relevant lines of the rule that perform this computation are the following:

```
1 const throughput = throughputHistory.getAverageThroughput(mediaType, isDynamic);
2 const bufferLevel = dashMetrics.getCurrentBufferLevel(mediaType);
3 const fragmentDuration = representationInfo.fragmentDuration;
4 const bitrate = throughput * (bufferLevel / fragmentDuration) *
  ↪  INSUFFICIENT_BUFFER_SAFETY_FACTOR;
```

Line 4 is where the upper limit is calculated. Since `(bufferLevel / fragmentDuration)` can be seen as the number/fraction of segments currently in the buffer, the maximum bitrate is calculated as a multiplication between the current throughput estimate and the number of segments in the buffer. However, there is a safety factor of 0.5 that effectively halves the maximum bitrate. When the bandwidth estimate is conservative (or has still to adapt to the new network conditions) or the buffer level is kept low on purpose to achieve lower latency, this rule tends to limit the video bitrate too aggressively.

For these reasons, we decided to disable the `InsufficientBufferRule` and instead create a **custom rule** that acts only when the buffer is actually empty. In this way, we avoid overreactions of the algorithm deriving from the low latency. The rule can be disabled as follows:

```
player.updateSettings({
    streaming: {
        abr: {
            ABRStrategy: 'abrThroughput'
            additionalAbrRules: {
                insufficientBufferRule: false
            }
        }
    }
});
```

The new custom rule can instead be added using the method shown in the following line, without having to modify and rebuild the library.

```
player.addABRCustomRule('qualitySwitchRules', 'BufferEmptyRule', BufferEmptyRule);
```

The implementation of the new `BufferEmptyRule` is similar to the original, but removes the maximum bitrate cap. Instead, it only forces a switch to the minimum bitrate when the buffer is empty. The following (simplified) code block shows how the check is performed.

```
if (currentBufferState.state === MetricsConstants.BUFFER_EMPTY) {
    logger.warning('[' + mediaType + '] BufferEmptyRule: switch to index 0.');
    switchRequest.quality = 0;
    switchRequest.reason = 'BufferEmptyRule: Buffer is empty';
}
```

This tweaked configuration, consisting in forcing the `THROUGHPUT` strategy and modifying the additional rules, produces different results compared to previous experiments. For example, the experiment with the `lte` dataset no longer shows the issue of oscillating bitrate, as shown in Figure 5.1.

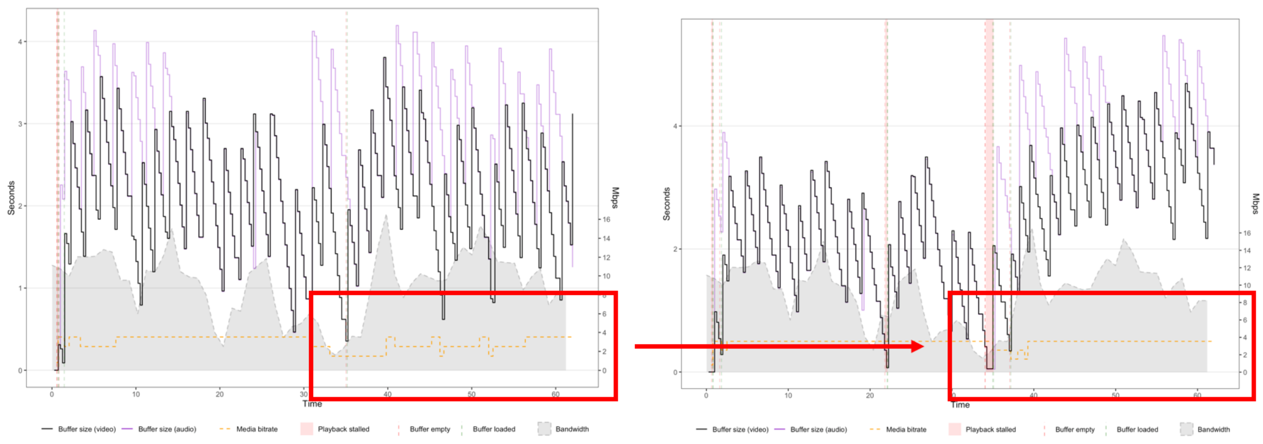


Figure 5.1: Buffer health plots showing the difference between the default `dash.js` ABR configuration and the tweaked configuration.

5.2 Switching to `HLS.js`

During the development of these improvements to `dash.js`, we had the opportunity to take a deeper look at the internals and architecture of `dash.js`. Unfortunately, we have found that most parts of the code are poorly documented and sometimes do not even have proper TypeScript types definitions. Also, our impression was that it was harder to understand how to extend the library and add new components. One of the reasons is that the library is using plain JavaScript, making it harder to have proper IDE support and type safety.

This led us to switch to `HLS.js` for the rest of our work. Unlike `dash.js`, `HLS.js` is written in TypeScript and is generally better documented. For example, the general architecture and design of the library is explained in detail in the official documentation.[38]

Unlike `dash.js`, `HLS.js` does not use a hybrid approach for the adaptation algorithm. Instead, it uses a standard **Exponential Weighted Moving Average** (EWMA) bandwidth estimator, which tracks bandwidth samples to produce an estimate of the available network bandwidth. This approach is used for both on-demand and live content, although with different configuration parameters for the EWMA computation.

[38] `https://github.com/video-dev/hls.js/blob/master/docs/design.md`

In addition to the bandwidth estimate, `HLS.js` also has an **abandon rule** similar to the one of `dash.js`. This rule is implemented in the `_abandonRulesCheck` method of the `AbrController` class and is periodically called by a timer. The method calculates the download rate of the current segment and estimates whether the segment will be loaded quickly enough to prevent buffer underflow, based on the estimate of bandwidth. If necessary, the rule will then find a quality level whose bitrate allows the segment to be loaded before buffer starvation, with a safety factor of 80%. If no level matches this condition, the lower bitrate is chosen.

While we were inspecting the source code that makes the request abandoning feature possible, we found a bug that prevented the check from running in some cases. We therefore reported the issue to the `HLS.js` developers and a fix is scheduled to be released in early 2023 among other improvements, to the abandon logic.[39]

5.3 Adding priority

As we have seen in Section 4.3.2, using HLS with the `spike` pattern tends to produce playback stalls when the bandwidth abruptly drops. This was evident in Figure 4.12. In this section, we will look at how **adding priority** to some requests helps reducing the issue.

The easiest way to manipulate how `HLS.js` sends HTTP requests is to use the `xhrSetup` or `fetchSetup` configuration options. They can be used to manipulate how HTTP requests are sent respectively with `XMLHttpRequest` or the `fetch` API, depending on how `HLS.js` is configured. In our case we use the default XHR loader, so we need to use the `xhrSetup` configuration option.

The `xhrSetup` option takes a function that is called just before sending the HTTP request. In practice, it allows to override how the request is created. In our case, to set the HTTP/3 priority we need to either set the `Priority` header or the `priority` query string parameter. In fact, in Section 4.1.1 we introduced a patch to the `h2o` web server that lets us set the priority with the query string parameter.

Since we are interested in **giving priority only to the audio segments requests**, we need to distinguish which requests correspond to those segments. Doing so in the `xhrSetup` function does not give a proper way to distinguish the type of request so we must resort to looking at the format of the URL. Since the chunk files contain the ID of the stream (i.e. audio or video), we can implement priority for audio segments as shown in Figure 5.2.

```
const hls = new Hls({
    xhrSetup: (xhr, url) => {
        if (url.includes('chunk-stream-0')) {
            url = url + '?priority=1';
        } else {
            url = url + '?priority=2';
        }
        xhr.open('GET', url, true);
    }
});
```

Figure 5.2: TypeScript code showing how to configure the XHR request and specify the priority for audio requests. In this case, we are giving the chunks for stream `1` an HTTP/3 priority of `1`, while all the other files have priority `2` (still higher than the default of `3`).

A better way is to override the `Hls` **loader** responsible for sending HTTP requests, to modify the URL of the segment based on its type, as shown in Figure 5.3. This is slightly more complex to implement, but we can still avoid to reimplement the whole loader implementation, taking advantage of inheritance.

[39] `https://github.com/video-dev/hls.js/issues/5094`

```
class CustomLoader extends Hls.DefaultConfig.loader {
    constructor(config: HlsConfig) {
        super(config);
        const load = this.load.bind(this);
        this.load = (context: FragmentLoaderContext, config: LoaderConfiguration,
        ↪  callbacks: LoaderCallbacks<LoaderContext>) => {
            if (context.frag.type === 'audio') {
                context.url += '?priority=1';
            } else {
                context.url += '?priority=2';
            }
            load(context, config, callbacks);
        }
    }
}
```

Figure 5.3: TypeScript code showing how a custom fragment loader for `HLS.js` can be implemented to override the priority for specific HTTP requests.

An instance of the custom fragment loader can then be passed as an option when creating the `Hls` instance:

```
const hls = new Hls({
    fLoader: CustomLoader as FragmentLoaderConstructor,
});
```

After testing this configuration that gives priority to audio, we quickly discovered that the results were not what we expected. In particular, there were still playback stalls, and in some cases the audio buffer became empty. However, this should not happen since there is always enough bandwidth to transfer the audio segments, which are very small and require just a few hundreds of kbps of bandwidth.

Looking at the logs of `HLS.js` in debug mode, we discovered that the audio media playlist took several seconds to load. Without an updated playlist, HLS does not know which segment should be requested next. This is different from DASH, where `SegmentTemplate` allows the client to generate the URLs of the segments without having to update the manifest (Section 2.4.2).

Consequently, if we want to give priority to audio, we also need to give priority to the corresponding media playlist, in addition to the segments. This can be done by overriding the `pLoader` (playlist loader) or the generic `loader`. In the second case, the implementation of the `load` method would be something like the following:

```
const fragmentContext = context as FragmentLoaderContext;
const playlistContext = context as PlaylistLoaderContext;

if (fragmentContext.frag && fragmentContext.frag.type === 'audio' ||
    playlistContext.url && playlistContext.type === 'audioTrack') {
    context.url += '?priority=1';
} else {
    context.url += '?priority=2';
}
```

Figure 5.4 shows how this approach affects the buffer health plot. Compared to Figure 4.12, we see that there are **fewer playback stalls**, even if the video buffer is often empty. Obviously, there is some unpredictability in these experiments, so different runs might show slightly different behaviors. However, the priority to audio is now assigned consistently and better resembles the behavior of HTTP/2 and HTTP/1.1 as seen earlier (Section 4.2.2).

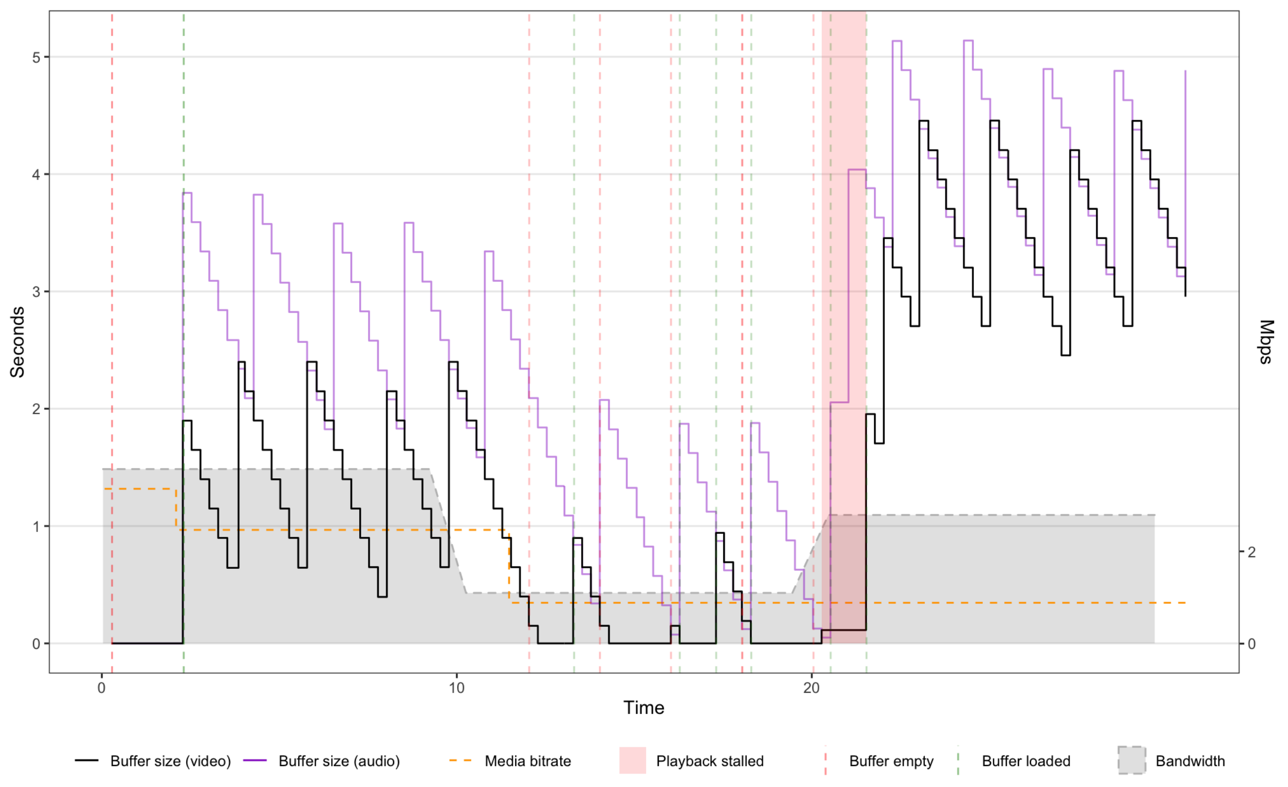


Figure 5.4: Buffer health plot for an HLS experiment with the `spike` pattern giving priority to the audio.

Clearly, when the video buffer is empty the video will appear frozen on screen while the buffer recovers, although the audio will keep playing. Unlike the situation where no priority is specified, **the live latency of the stream will not increase**.

5.4 A new and different live user experience

The behavior that we have shown in the previous section derives from the Chromium implementation of the handling of buffer underflows. This means, for example, that in other browsers when one of the two buffers is empty the playback will immediately stall.

Another limitation is that in Chromium the playback stall is avoided only for three seconds. Ideally, we would like to have more control on how and when this happens, on all browsers.

The final objective is to build **a user experience with fewer playback stalls**. Usually, when a live stream struggles to adapt to network conditions the viewer is used to see both audio and video stuck for a while during rebuffering. When this happens, latency tends to grow and playback lags behind the edge of the live playlist. As a consequence, users will perceive the action late, possibly by even tens of seconds.

This kind of experience is **different in comparison to other broadcasting systems**. For example, digital terrestrial and satellite do not have the concept of buffering: if the video signal is not good enough, video and/or audio are temporarily unavailable while the signal recovers, but no latency is introduced, i.e. viewers always see the live action with a fixed delay independently from the signal quality.

Adaptive streaming research has been mostly focused on avoiding empty buffers as much as possible, although in low-latency scenarios entirely avoiding rebuffering is a very hard task. Instead, we can try to implement a solution that tries to resemble the linear TV experience that viewers are already used to. Although subjective, it would likely result in a better Quality of Experience (QoE).

5.4.1 Related works

There are a few proposals that in the past few years attempted to provide an alternative solution to the problem of video transmission over unreliable networks.

In 2019, the authors of [33] proposed an approach called **frame discarding**. The idea was to exploit HTTP/2 streams to transmit each individual video frame independently. When network conditions become adverse, some carefully selected video frames are discarded and their download is aborted through the stream resetting feature of HTTP/2, saving bandwidth and at the same time preserving playback continuity.

In 2021, Facebook presented a draft for **Reliable (Unreliable) Streaming Protocol** (RUSH), which uses the QUIC protocol to provide a replacement for RTMP for video ingestion on unreliable networks. Video frames are transmitted in independent QUIC streams, providing a way to stop retransmissions if a video frame is not fully delivered on time.[24]

In 2022, Twitch, the video streaming platform, submitted a draft RFC for **Warp**, a new protocol for segmented media delivery over QUIC. Warp transmits one video segment per QUIC stream and gives priority to newer video segments and to the audio track. Video segments that cannot be delivered in time are reset at the QUIC stream level to free bandwidth, an approach that Twitch calls **segment truncation**. Warp can be used for video delivery in the browser thanks to WebTransport, an abstraction of QUIC, although at the time of writing the draft proposal does not include adaptive bitrate support.[14]

Finally, a new IETF working group, **Media Over QUIC**, is evaluating how to consolidate video streaming protocols that work over QUIC in a single standardized solution.[40]

5.5 Implementation

To our knowledge, the solutions mentioned in Section 5.4.1 are the only attempts that have been publicly proposed to tackle the issue of video streaming from a different perspective.

In this work, we make an attempt to implement an approach similar to the one proposed by Warp, but over HTTP and using an existing ABR protocol. Specifically, we modify the `HLS.js` library and take advantage of new HTTP features made available by HTTP/3, something that has not been tried before, to our knowledge.

In practice, implementing the solution means tackling the following points:

- **Prioritization of requests**: as we have already seen in depth, audio segments and playlists should be prioritized over video.
- **Segment dropping**: when it is too late for a video segment to be useful, we can cancel its download to free bandwidth.
- **Filler segments**: when a segment is dropped, we need to provide a filler or placeholder to the source buffer. This "fake" segment should be generated on the client side in the browser.

5.5.1 Segment dropping

In a live stream, when a video segment has no chance of being loaded in time for playback, we can decide that there is no point in continuing the download and instead discard it. In this way, we free bandwidth that can be used to load the next segment.

In HTTP/3, this can be done by aborting the HTTP request, an action that in practice resets the underlying QUIC stream. Resetting a stream means sending a frame of type `RESET_STREAM` and immediately closing the stream, also stopping the transmission of the data, if any.

In `HLS.js`, video segments are downloaded sequentially, one at a time. Therefore, we can always look at the current segment and estimate whether the segment will not arrive on time and cancel the download if needed. This is somewhat similar to what the `_abandonRulesCheck` does, which we mentioned in Section 5.2. The `_abandonRulesCheck` method is called by a JavaScript interval every 100 milliseconds and evaluates whether the current segment is taking too long to load. If so, it will abort the load of the segment.

[40]https://datatracker.ietf.org/wg/moq/about/

For our scenario, we needed to implement something similar. As a first attempt, we implemented a simpler mechanism that checks whether we are about to reach the end of the video buffer. We therefore modified the `HLS.js` source code and specifically the `AbrController`. We added a new periodic timer for a function named `fillerCheck`, which calculates how much time is left in the buffer, as shown in the following code block.

```
const bufferInfo = BufferHelper.bufferInfo(
    media,
    media.currentTime,
    config.maxBufferHole
);

const bufferLength = bufferInfo.len;
const nextFragmentOffset = frag.start - bufferInfo.end;
```

The `bufferLength` variable is self-explanatory: it corresponds to the seconds of media data currently in the buffer between the current video timestamp and the end of the buffer. Obviously, if no data is appended to the buffer the length will decrease at every tick of the timer. The `nextFragmentOffset` is the time difference between the end of the buffer and the start of the fragment. This is used to understand whether the fragment currently being loaded is actually the one immediately following the end of the buffer. Ideally, the difference should be zero, but timestamps are not always that precise so we use a small tolerance (`maxBufferHole`).

When the above condition is met and the `bufferLength` is less than a threshold, we make the decision to abort the segment load. The threshold is set by default to 200 milliseconds (considering the fact that the tick interval is 100 milliseconds), but it is configurable. Another thing that the implementation needs to consider and handle is that the segment loading might be aborted by other rules, for example the `_abandonRulesCheck`.

Upon making the decision to abort the segment loading, we cancel the actual download of the segment file. Since we are in the `AbrController` and have access to the `Fragment` being loaded, we can directly interact with the `Loader` instance through the `loader` property. We therefore modified the `Loader` interface and defined a new method that mainly performs two operations:

- Cancel the HTTP request, and thus reset the underlying QUIC stream in the case of HTTP/3.
- Emit an event so that other actions can take place, as we will see in the next section.

5.5.2 Generating the filler segment

When the segment dropping code makes the decision to discard a video segment and interrupt its loading, the buffer must still be filled with some placeholder data in order not to interrupt playback. We call this “fake” segment a **filler segment**.

To reproduce the Chromium underflow behavior, which consists of showing the last rendered frame on the video player while the playback continues with audio, we would need to **generate a video segment on the fly in the browser**. In practice, this means encoding a 2-second video fragment containing a still picture, corresponding to the video frame.

As a first implementation to confirm the feasibility of the solution, we implemented a simpler solution that uses an empty frame with a solid color background. Since H.264 does not necessarily contain timing data, in this way the segment can be generated statically beforehand and reused multiple times. It can even be hardcoded in the client code.

However, the H.264 bitstream alone is not enough to play the segment. In fact, the Media Source Extensions API expects media data to be muxed in the fragmented MP4 format. Consequently, the H.264 data must be inserted into the MP4 container with the correct timing information.

Moreover, since the filler segment is generated with different encoding parameters with respect to the main stream, there is a discontinuity that the video decoder must be aware of. In `fMP4`, this signaling of the new parameters is done through initialization segments, that is, segments that contain at least the ftyp and moov boxes. This information is shared by all the segments following the init segment, until a new one is found.

Once both the init segment and the segment data are available, they are appended to the SourceBuffer so that the video segment can be played.

5.5.3 Muxing in the browser

As mentioned above, once we have the H.264 bitstream of the video frame we have to **mux it into the fMP4 container** to produce the segment. The segment does not need to contain 25 frames per second, but it can actually contain only one video frame with a duration of 2 seconds (and the correct start timestamp).

The browser APIs do not provide a way to mux or transmux video data, therefore we had to rely on other libraries. An important fact that affects the implementation is that we are dealing with the fragmented MP4 container format and not simply MP4. In fact, not all libraries support the fMP4 format.

After investigating several possible options, we came to the conclusion that there was no complete JavaScript library that implements fMP4 muxing in the browser. In fact, fMP4 muxing is usually performed directly by the libraries that use the MSE API, such as HLS.js when transmuxing from MPEG-2 TS to fMP4.

Instead of delving into a manual JavaScript implementation of fMP4 fragments muxing from the raw H.264 bitstream, we decided to experiment with an approach based on WebAssembly, which turned out to be successful.

WebAssembly (WASM) is a low-level binary format that can be used as a build target of many programming languages. It is meant to provide an efficient alternative to JavaScript: a WASM binary file can be embedded and executed in a web page, effectively allowing the execution of code written in any supported programming language in the browser, at close-to-native performance.

In our case, we used WASM to run a Go application that performs the muxing of the video data in the browser. The Go programming language provides out-of-the-box support for building an application to WASM. For example, to build a main.go file into a WASM binary named main.wasm one would use the following command:

```
GOOS=js GOARCH=wasm go build -o main.wasm main.go
```

The Go code built in the WASM format can then be executed in the browser thanks to a JavaScript utility provided by Go and included with every Go installation, wasm_exec.js. In the web page, a simple code snippet such as the following loads the WASM binary file and executes its content.

```
const go = new Go();
WebAssembly.instantiateStreaming(fetch('/main.wasm'), go.importObject)
    .then((result) => {
        go.run(result.instance);
    });
```

Within the Go application entrypoint, namely the main.go file, we can directly interact with the Document Object Model (DOM) of the web page. For example, we can access the window object and manipulate the page contents directly. Alternatively, we can expose functions that can then be called by the JavaScript code by setting them as global variables in the window object.

For example, in our case we expose two functions, createInit and createFragment, as shown in Figure 5.5. These functions are then called by the TypeScript code when generating the filler segment data.

Both functions take the H.264 coded data as input, which is then muxed into an fMP4 container. To perform the muxing, we used mp4ff, a library to parse and write MP4 files.[41] The input H.264 data must be in the Annex B format with start codes (see Section 2.1.4).

[41] https://github.com/edgeware/mp4ff

```go
//go:build js && wasm
package main
import "syscall/js"

func main() {
	done := make(chan struct{}, 0)
	global := js.Global()
	global.Set("createInit", js.FuncOf(jsCreateInit))
	global.Set("createFragment", js.FuncOf(jsCreateFragment))
	<-done
}
```

Figure 5.5: Snippet of a Go application to expose two functions as WebAssembly functions.

For the initialization segment, we must create an `fMP4` file containing a `moov` atom that defines some characteristics of the video tracks. In this case, we only have one video track in the H.264 format, whose decoding information is written in a box in the `avcC` format (see Section 2.3.2).

To build the H.264 descriptor, we must first extract two NAL units from the H.264 bitstream, specifically the **Sequence Parameter Set** (SPS) and the **Picture Parameter Set** (PPS). These units contain information such as the width and height of the frame and other encoding parameters. Extracting them means parsing the NAL units from the H.264 bitstream and identifying which are the SPS and PPS units, as shown in Figure 5.6.

```go
func ExtractSpsPps(bitstream []byte) ([]byte, []byte) {
	nalus := avc.ExtractNalusFromByteStream(bitstream)
	var sps []byte
	var pps []byte
	for _, nalu := range nalus {
		switch avc.GetNaluType(nalu[0]) {
		case avc.NALU_SPS:
			sps = nalu
		case avc.NALU_PPS:
			pps = nalu
		}
	}
	return sps, pps
}
```

Figure 5.6: Go function showing how to the SPS and PPS NAL units can be extracted from an H.264 bitstream in Annex B format, with `mp4ff`.

The initialization segment can then be built by adding a `moov` box, which contains a single `trak` with the `avcC` information. In this step, we also need to assign a time scale to the video track. The time scale represents the number of time units per second, and is then used as a basis for assigning a timestamp to the specific video frame. For example, if we set the time scale to 12800, a typical value, it means that in a 25 fps video file each frame will have a timestamp that is a multiple of 512.

The WASM `createFragment` function is instead used to create the actual media segment, or `fMP4` fragment. The inputs of this function are not only the H.264 bitstream, but also the start timestamp of the segment and its duration, both expressed in seconds.

In the actual implementation, we must create another fragmented MP4 file instance and only add one fragment to it. Since the Annex B packetization format is incompatible with MP4, we must first convert the byte stream to packets. The library mp4ff provides us with a helper function to do this.

We then need to add the video frames to the segment. In our case, we have a single frame which we will put in an MP4 sample with a duration of 2 seconds. Note that both the timestamp and the duration must be expressed in terms of the time scale, as mentioned before. Figure 5.7 shows a simplified piece of code that creates the sample and adds it to the fragment.

```
bitstream = avc.ConvertByteStreamToNaluSample(bitstream)

sample := mp4.FullSample{
        Sample: mp4.Sample{
                Flags:                 mp4.SyncSampleFlags,
                Dur:                   uint32(TimeScale * duration),
                Size:                  uint32(len(bitstream)),
                CompositionTimeOffset: 0,
        },
        DecodeTime: uint64(TimeScale * timestamp),
        Data:       bitstream,
}

frag.AddFullSample(sample)
```

Figure 5.7: Snippet from the Go WASM application showing how to create an fMP4 sample from the input H.264 bitstream.

5.5.4 Discontinuity handling in HLS.js

As we have seen before, when the modified AbrController in HLS.js decides that the segment loading should be aborted and the segment replaced with a filler segment, the loader implementation will invoke a callback.

This callback is handled in the FragmentLoader class of HLS.js, which is an abstraction that coordinates the lifecycle of the actual loaders, which are created whenever the StreamController decides that a new segment must be loaded. The FragmentLoader is therefore the place where the WASM functions createInit and createFragment are called to produce the new fragment.

Once the generated fragment data is ready, the FragmentLoader will signal to the StreamController that the segment loading has been completed. The data will then be passed to the **transmuxer**, which is again an abstraction that coordinates the demuxer and remuxer implementations. In this case, the data is already in the fMP4 format so the transmuxer is a passthrough remuxer.

One important detail that is crucial for the solution to work is making sure that the remuxer knows that there was a discontinuity, and therefore emit the new initialization segment. The transmuxer bases this decision on the following logic, which is a snippet from the TransmuxerInterface class:

```
const initSegmentChange !(
    lastFrag && frag.initSegment?.url === lastFrag.initSegment?.url
);
```

Basically, the transmuxer relies on the URL of the init segment to decide whether it changed with respect to the previous fragment. Therefore, when in the FramentLoader we assign the initialization data to the fragment we must also change its URL. In our implementation, we simply replace the URL with a random string, since it is not actually used anywhere.

```
const url = Math.random().toString();
frag.initSegment = new Fragment(PlaylistLevelType.MAIN, url);
frag.initSegment.data = initData;
```

In this way, every time there is a filler segment, the transmuxer session will be reset and a new initialization segment will be emitted and appended to the MSE `SourceBuffer`. The same will happen with the following segment.

5.5.5 Encoding the filler segment with WebCodecs

In Section 5.5.2, we explained how the initial implementation only used a static filler segment with no actual video content. However, it is possible to actually encode the video segment in the browser through JavaScript code, thanks to the **WebCodecs API**, and make its content dynamic.

WebCodecs is a set of browser APIs that provide high-level methods to **encode and decode audio and video content in the browser**. The specification is still a draft, but is already implemented and enabled by default in Chromium, and therefore all Chromium-based browsers, since the end of 2021. These browsers represent more than 70% of the global web users, at the time of writing. Safari also enabled WebCodecs support by default in the Technical Previews of the browser in November 2022.[42] Firefox still does not have support for WebCodecs but is in the process of implementing it.[43]

With WebCodecs, we can encode video frames and **produce a raw bitstream** as an output. The codecs that can be used depend on which encoders are included and distributed with the browser build. For example, Google Chrome always ships with an H.264 encoder. WebCodecs only takes care of encoding and decoding media data but does not handle muxing or demuxing.

There are several classes that are part of WebCodecs. For our use case, we are interested in the `VideoEncoder` and `VideoFrame` classes. Individual video frames can be created using the `VideoFrame` constructor, from multiple types of sources, and are then fed into an instance of the `VideoEncoder`, one at a time. The outputs of the `VideoEncoder` are `EncodedVideoChunk` instances that together represent the encoded video.[9]

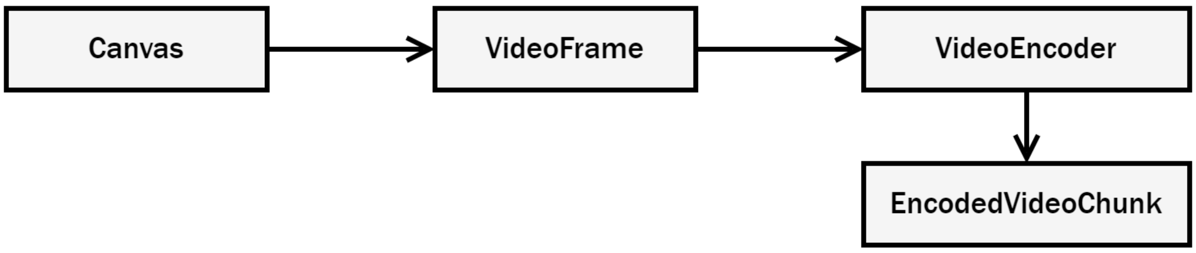


Figure 5.8: The path to encode a video file with WebCodecs, from a `Canvas` to the encoded chunks.

In our implementation, the `VideoFrame` is constructed from an HTML `canvas`, which is an element with a predefined size on which an image can be drawn. Once we have the canvas, the `VideoFrame` is created like this:

```
const frame = new VideoFrame(canvas, {
    timestamp: frag.start,
});
```

To encode the frame, we first need to create a `VideoEncoder` instance specifying a codec configuration supported by the browser. Unfortunately, it is not possible to obtain a list of supported configurations; therefore, a possible strategy is to make multiple attempts until a working configuration is found. The async static method `VideoEncoder.isConfigSupported(config)` lets developers check whether the input configuration is supported.

The code example in Figure 5.9 shows how a `VideoEncoder` instance can be configured. In this case, the following options are specified:

[42]`https://caniuse.com/webcodecs`
[43]`https://bugzilla.mozilla.org/show_bug.cgi?id=1746557`

```
const codec = 'avc1.64001f';
const width = 1280;
const height = 720;
const birate = 3_000_000;
const framerate = 25;

const config = {
    codec,
    width,
    height,
    bitrate,
    framerate,
    avc: { format: 'annexb' as AvcBitstreamFormat },
};

const encoder = new VideoEncoder(init);
encoder.configure(config);
```

Figure 5.9: TypeScript code snippet that initializes and configures and instance of the WebCodecs `VideoEncoder`.

- **Codec**: a string that defines which codec should be used for encoding.
  - The characters before the dot define the name of the codec, as specified by the WebCodecs Codec Registry. For example, in `avc1.64001f` the string `avc1` refers to AVC/H.264.[44]
  - The suffix of the string is specific to the coding format and contains information such as the profile, level, bit depth, etc., that are needed by the decoder. In the case of H.264, the six characters are in hexadecimal format: the first two characters refer to the profile (in the example, `64` corresponds to `High`), while the last two declare the level (in the example, `1f` corresponds to Level 3.1).
- **Width and height**: the resolution of the output video.
- **Bitrate**: the target average bitrate to use for encoding.
- **Framerate**: the number of frames per second of the output video.
- **AVC** contains H.264-specific options. In this case, we are asking the encoder to return a stream in the Annex B format.

There are also options to control whether **hardware acceleration** should be preferred, if available, and whether to **optimize for low latency**.

Encoding complex video sequences with `VideoEncoder` requires careful considerations on the rate with which frames should be provided to the encoder, but in our case we just need to encode a single frame, so the implementation is simpler. The encoding of a frame can be started with the following function call:

```
encoder.encode(frame);
```

In these code samples, error handling and proper management of the resources lifecycle are omitted for brevity.

[44]`https://www.w3.org/TR/2022/DRY-webcodecs-codec-registry-20221013/`

When a newly encoded video chunk is available, a callback defined as a property of the `init` object passed to the `VideoEncoder` is invoked. All the data required by the decoder, including the SPS and PPS metadata, are included in the provided bitstream in the Annex B format.

As mentioned above, the video frame can be created starting from a 2D `canvas` that can contain any picture. If we want to create a video segment made of a still picture of the last frame shown on the screen, we can do that with the `canvas` API. The following code block shows how we can draw the current image rendered by a `<video>` element on a `canvas`.

```
const canvas = document.createElement('canvas');
const ctx = canvas.getContext('2d');
ctx.drawImage(video, 0, 0, canvas.width, canvas.height);
```

Note that the source image resolution might not correspond to the one of the `canvas`. For this reason, we must specify the destination dimensions when calling the `drawImage` function, so that the picture is scaled correctly.

Also, while the filler segment could in theory be of any resolution independently from the current ABR level, in our implementation we try to match the expected resolution.

5.5.6 Changes to the testbed

In the previous sections, we introduced how an alternative user experience for live streaming can be built, taking advantage of multiple techniques such as giving priority to audio segments, dropping segments and generating filler segments on the fly. To **test and integrate the solution with the testbed**, we had to make some changes to it.

First, we swapped the `HLS.js` implementation with our modified version. Since we are building the application with `Vite`, which bundles the dependencies declared in the `package.json` file, we can replace the version of the `HLS.js` library with a local module or the path to a remote repository. In our case, we chose to fork the `hls.js` repository on GitHub so that `npm` can pull the modified library from there.

For WebAssembly to work, the initialization code mentioned in Section 5.5.3 must be executed on the web page. This initialization code can be included in two ways: through the TypeScript code of the `HLS.js` library, or directly in the HTML page of the testbed frontend. In the second case, we must modify the testbed to include the WebAssembly initialization.

If audio prioritization is not implemented directly in the modified version of `HLS.js`, we must modify the `Hls` instance creation to override the HTTP loader, as seen in Section 5.3. Finally, for a proper analysis of the performance of the solution it would be useful to know when a segment was dropped and replaced with a filler. To do this, the statistics collector on both the frontend and the backend need to be updated to collect this information.

5.6 Experimental results

In this section, we analyze some results that we obtained with the modified testbed and observe whether the solution indeed produces a better user experience.

Figure 5.10 shows the buffer health plot for an experiment ran with the modified `HLS.js` library. The protocol is HTTP/3 so that priorities can be used. The network pattern is `spike`, the one that in previous sections showed bad performance.

As we can see, **the playback stalls have been completely removed**. Playback is not interrupted because there is always some audio in the buffer to play. The dark areas refer to the segments that have been dropped and replaced with a filler segment.

In Figure 5.11 we can also see the live latency and the waterfall. It is clear that with this solution the live latency stays stable even during the congestion periods. In the waterfall diagram, we can see which segments are discarded, represented in black.

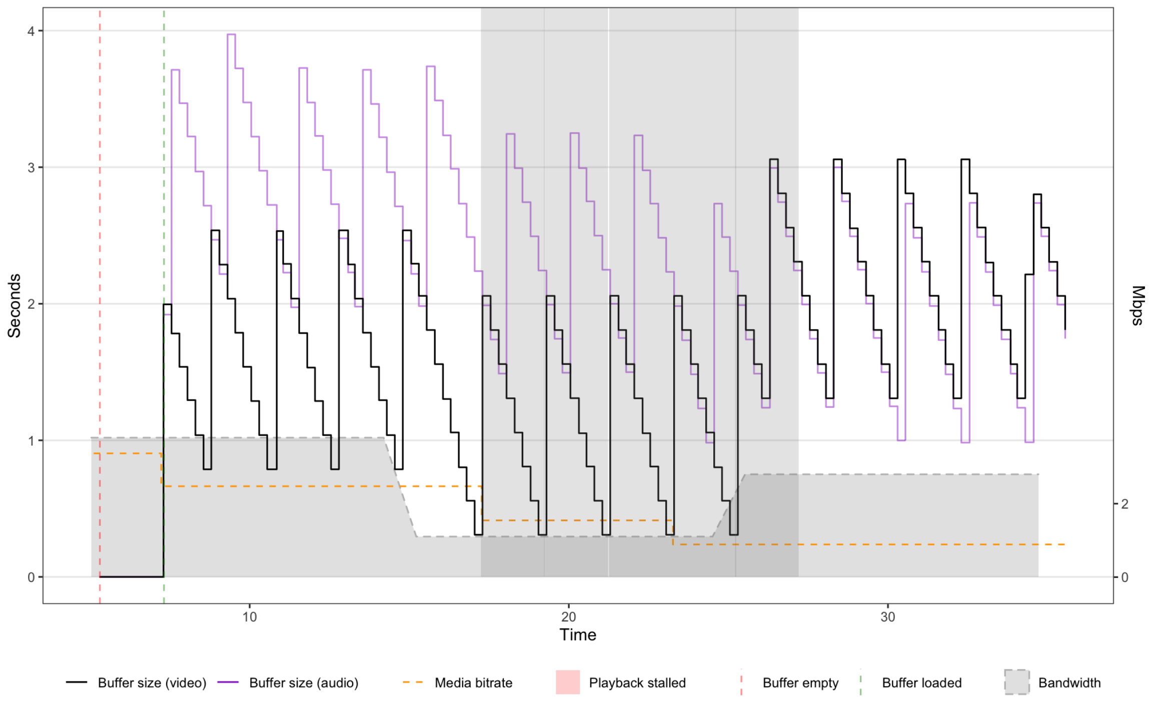
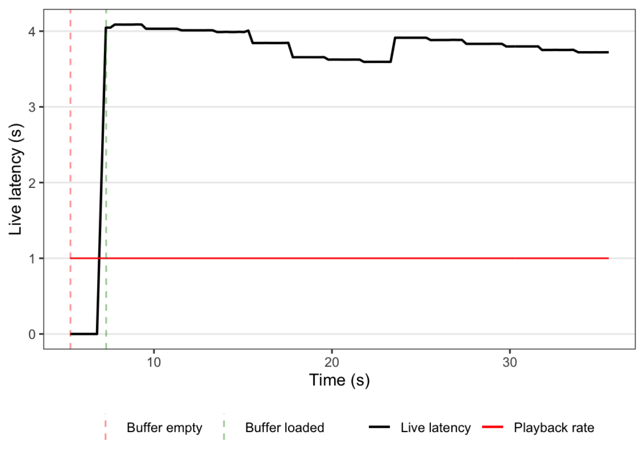
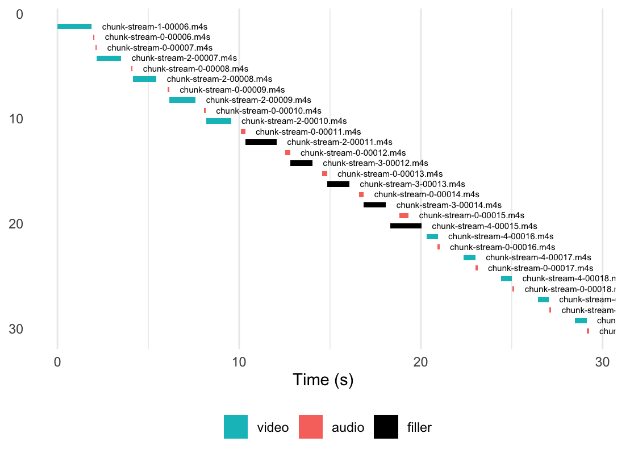


Figure 5.10: Buffer health plot for an experiment using the modified `HLS.js` library. The protocol is HTTP/3 and priority is given to audio. The network pattern is `spike`. Outdated segments are dropped and replaced by the filler segment (dark areas).



(a) Live latency plot.



(b) Waterfall diagram.

Figure 5.11: Additional plots for the experiment with modified `HLS.js` version over HTTP/3 with the `spike` network pattern.

Note that the bandwidth drop in the `spike` pattern is quite aggressive and in general shows very low bandwidth, so the fact that five consecutive segments are dropped is somehow expected. As mentioned in previous sections, running the experiment multiple times results in slightly different results. However, there is still room for improvement. For example, the discarding of the segment should free up the bandwidth needed for loading the next segment, but we see that this is not the case with this particular configuration. In the final chapter, we will propose some ideas that might be worth investigating in the future to further improve the solution.

Chapter 6

Conclusions

In this thesis, we focused on the topic of **live video streaming over HTTP**, specifically low-latency streaming. We analyzed how existing protocols behave in realistic scenarios and then proposed some improvements that we believe produce an overall better user experience.

To properly analyze how popular streaming technologies work, we built a custom **testbed** that makes it possible to execute emulated experiments in a predictable environment. The testbed is based on the ComNetsEmu emulator and takes advantage of a headless Chromium instance to render a video player in a web page. We implemented support for the two main live streaming protocols, namely **Apple HLS** and **MPEG-DASH**, through the JavaScript libraries `HLS.js` and `dash.js`. An important feature of the testbed is the ability to apply custom network patterns, which lets us emulate networks with unstable throughput, latency, or reliability in general.

With this testbed, we were able to collect important events and metrics related to the playback of low-latency streams. This data was then analyzed and plotted, allowing us to better observe and compare the behavior of each streaming setup and configuration.

While running and executing the experiments, we discovered that live streaming over HTTP/3 tends to show a suboptimal behavior that was not observed with HTTP/1.1 and HTTP/2. In particular, the default prioritization of HTTP requests tends to give more priority to video segments over audio, without taking into consideration that audio segments are always much smaller than video and could benefit from being loaded earlier.

With this in mind, we tried to implement a modified version of `HLS.js` that tries to resemble the user experience of linear TV channels in traditional broadcasting mechanisms. In those scenarios, when the signal is not good enough, video or audio playback will get stuck briefly but will not cause an increase in latency. To do that, we took advantage of HTTP/3 and experimented with ways to tweak the **priorities** given to requests, so that audio is prioritized. We then implemented an approach we call **segment dropping**, that consists of canceling the load of a video segment if it does not arrive in time. The missing video segment is then replaced with a **filler segment**, which we generate on the fly in the frontend with the browser WebCodecs API, also taking advantage of WebAssembly to mux the H.264 bitstream in an `fMP4`-compliant fragment in an efficient way.

The combination of these approaches makes it possible to **greatly reduce the number of playback stalls** and provides a smoother live streaming user experience overall. In fact, when the bandwidth suddenly drops and the ABR rate adaptation algorithm is not able to react in time and switch to a lower-quality rendition, our strategy kicks in and avoids playback stalls caused by missing video. Since the (small) audio segments are prioritized, audio is expected to be always available and is thus always played, even when the network is congested and no video can be loaded. The final result is that **the live latency of the stream does not increase** in case of congestion, which is the result that we wanted to achieve.

6.1 Future work

Although the solution we implemented in Chapter 5 showed some interesting results, there are a few aspects of the implementation that could be further studied.

For example, we tested the solution with a specific configuration and set of software pieces. Clearly, they are not the only possible option. For example, the suboptimal prioritization behavior of HTTP/3 that we observed in Section 4.2.1 probably depends on the scheduling strategy of the specific implementation of the HTTP server. In this testbed, we made the choice to use the `h2o` web server, but as we mentioned there are many HTTP/3 implementations and they could show different behaviors. This is something worth investigating.

Similarly, the testbed was built with Chromium as a web browser, but, as we mentioned, the solution can work on other browsers as well. A possible improvement is therefore to add support for multiple browsers to the testbed. This would also require swapping the Puppeteer library, used to control the browser programmatically, with a more generic library that also supports browsers other than Chromium.

With respect to the actual quality of experience provided by the solution we implemented, we noticed that in the case of sudden and substantial drops in bandwidth there could be several seconds in which the video segment cannot be downloaded, even if the segment dropping should help in freeing bandwidth. For this reason, we believe that the segment dropping approach should be combined with a more aggressive ABR switching algorithm that takes into account that a video segment could not be completely loaded for several seconds.

Significant improvements in the rate adaptation logic of `HLS.js`, especially in scenarios with short segments and low latency, were being developed while this thesis was being written and are expected to be released in early 2023. The solution should therefore be tested again after pulling these improvements in the forked source code.[45]

Another important aspect that is worth investigating is trying to design and develop a more complex prioritization strategy. For example, the problem of video segments being loaded in time that we mentioned above could be alleviated by giving more priority to "future" segments, while loading the segments in the middle only if there is enough bandwidth left. However, this would probably require substantial changes to the `HLS.js` stream controller implementation, since it does not currently support loading multiple segments of the same track at the same time. Experimenting with different HLS configurations, such as different segment lengths, is also an idea.

Finally, to overcome the requirement of the WebCodecs API being available with the correct encoder configuration, future work could investigate the feasibility of implementing a simplified version of an H.264 encoder in the browser. This implementation could take advantage of the possibilities offered by WebAssembly and provide an optimized way to encode a single video I-frame on the fly, without the overhead of a complete video encoder.

[45]`https://github.com/video-dev/hls.js/pull/4825`

Bibliography

[1] Audio Priming - Handling Encoder Delay in AAC. `https://developer.apple.com/library/archive/documentation/QuickTime/QTFF/QTFFAppenG/QTFFAppenG.html`.

[2] H.264: Advanced video coding for generic audiovisual services. `https://www.itu.int/rec/T-REC-H.264`.

[3] HTTP Live Streaming (HLS) Authoring Specification for Apple Devices. `https://developer.apple.com/documentation/http_live_streaming/http_live_streaming_hls_authoring_specification_for_apple_devices`.

[4] Bitmovin Video Developer Report. `https://go.bitmovin.com/video-developer-report-2021`, 2021.

[5] MPEG-4 Part 12: ISO base media file format. `https://www.iso.org/standard/83102.html`, January 2021. 7th edition.

[6] MSU Video Codecs Comparison 2021. `https://compression.ru/video/codec_comparison/2021/main_report.html`, 2021.

[7] Dynamic adaptive streaming over HTTP (DASH) — Part 1: Media presentation description and segment formats. `https://www.iso.org/standard/83314.html`, August 2022. 5th edition.

[8] Media Source Extensions™. https://www.w3.org/TR/media-source/, September 2022. W3C Working Draft.

[9] Paul Adenot and Bernard Aboba. Webcodecs. W3C working draft, W3C, November 2022. https://www.w3.org/TR/2022/WD-webcodecs-20221122/.

[10] Mike Belshe, Roberto Peon, and Martin Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540, May 2015.

[11] Mike Bishop. HTTP/3. RFC 9114, June 2022.

[12] Karlheinz Brandenburg. MP3 and AAC explained. March 2001.

[13] VNI Cisco. Cisco visual networking index: Forecast and trends, 2017–2022 white paper. *Cisco Internet Rep*, 17:13, 2019.

[14] Luke Curley, Kirill Pugin, and Suhas Nandakumar. Warp - Segmented Live Media Transport. Internet-Draft draft-lcurley-warp-02, Internet Engineering Task Force, October 2022. Work in Progress.

[15] Kerem Durak, Mehmet N. Akcay, Yigit K. Erinc, Boran Pekel, and Ali C. Begen. Evaluating the Performance of Apple's Low-Latency HLS. In *2020 IEEE 22nd International Workshop on Multimedia Signal Processing (MMSP)*, pages 1–6, 2020.

[16] Roy T. Fielding, Henrik Nielsen, Jeffrey Mogul, Jim Gettys, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2068, January 1997.

[17] Ozer Jan. *Video Encoding by the Numbers. Eliminate the Guesswork from your Streaming Video.* Doceo Publishing, December 2016.

[18] Ryan Lei, Jun Sik Song, Adrian Grange, Jingning Han, John Simmons, and Andrey Norkin. AV1 benchmarking test for 3GPP. In Andrew G. Tescher and Touradj Ebrahimi, editors, *Applications of Digital Image Processing XLV*. SPIE, October 2022.

[19] Thierry Lelegard. An introduction to MPEG-TS. `https://tsduck.io/download/docs/mpegts-introduction.pdf`.

[20] Robin Marx, Joris Herbots, Wim Lamotte, and Peter Quax. Same Standards, Different Decisions: A Study of QUIC and HTTP/3 Implementation Diversity. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, EPIQ '20, page 14–20, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Kazuho Oku and Lucas Pardue. Extensible Prioritization Scheme for HTTP. RFC 9218, June 2022.

[22] Jan Ozer. The Future of HEVC Licensing Is Bleak, Declares MPEG Chairman. `https://www.streamingmedia.com/Articles/Editorial/Featured-Articles/The-Future-of-HEVC-Licensing-Is-Bleak-Declares-MPEG-Chairman-122983.aspx`, January 2021.

[23] Roger Pantos and William May. HTTP Live Streaming. RFC 8216, August 2017.

[24] Kirill Pugin, Alan Frindell, Jordi Cenzano, and Jake Weissman. RUSH - Reliable (unreliable) streaming protocol. Internet-Draft draft-kpugin-rush-01, Internet Engineering Task Force, March 2022. Work in Progress.

[25] Darijo Raca, Jason J. Quinlan, Ahmed H. Zahran, and Cormac J. Sreenan. Beyond Throughput: A 4G LTE Dataset with Channel and Context Metrics. In *Proceedings of the 9th ACM Multimedia Systems Conference*, MMSys '18, page 460–465, New York, NY, USA, 2018. Association for Computing Machinery.

[26] Iain E. Richardson. *The H.264 Advanced Video Compression Standard.* Wiley Publishing, 2nd edition, 2010.

[27] Traci Ruether. MPEG-DASH: Dynamic Adaptive Streaming Over HTTP Explained. `https://www.wowza.com/blog/mpeg-dash-dynamic-adaptive-streaming-over-http`, April 2022.

[28] Kevin Spiteri, Ramesh Sitaraman, and Daniel Sparacio. From Theory to Practice: Improving Bitrate Adaptation in the DASH Reference Player. In *Proceedings of the 9th ACM Multimedia Systems Conference*, MMSys '18, page 123–137, New York, NY, USA, 2018. Association for Computing Machinery.

[29] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K. Sitaraman. BOLA: Near-Optimal Bitrate Adaptation for Online Videos. *IEEE/ACM Transactions on Networking*, 28(4):1698–1711, 2020.

[30] Sydney Whalen. What is CMAF? `https://www.wowza.com/blog/what-is-cmaf`, August 2022.

[31] Maarten Wijnants, Robin Marx, Peter Quax, and Wim Lamotte. HTTP/2 Prioritization and Its Impact on Web Performance. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, page 1755–1764, Republic and Canton of Geneva, CHE, 2018. International World Wide Web Conferences Steering Committee.

[32] Zuo Xiang, Sreekrishna Pandi, Juan Cabrera, Fabrizio Granelli, Patrick Seeling, and Frank H. P. Fitzek. An Open Source Testbed for Virtualized Communication Networks. *IEEE Communications Magazine*, 59(2):77–83, 2021.

[33] Mariem Ben Yahia, Yannick Le Louedec, Gwendal Simon, Loutfi Nuaymi, and Xavier Corbillon. HTTP/2-Based Frame Discarding for Low-Latency Adaptive Video Streaming. *ACM Trans. Multimedia Comput. Commun. Appl.*, 15(1), feb 2019.

Ringraziamenti

Ringrazio il prof. Granelli per avermi dato l'opportunità di svolgere questo lavoro, nonché tutte le persone che mi hanno accompagnato in questo lungo percorso universitario. Un ringraziamento speciale va a Giovanni, Alberto, Verdiana, Francesco, Emilio, Luca, Alessandro per l'a volte inconsapevole supporto datomi nell'ultimo anno. Infine, ringrazio i miei genitori per il continuo sostegno in questi anni.