Contents

1 Introduction

Video streaming is the major source of traffic on the Internet, accounting for more than 75% of the total traffic in 2022 [5]. Live Streaming itself accounted for 17% of internet video traffic in 2022, a 15-fold increase since 2017. At the same time, low latencies are becoming increasingly more important in live streaming applications.

In live streaming systems, latency increases when the network is congested, if the network bandwidth can't keep up with the stream's bitrate. The only way to prevent the latency from increasing is to send less data. HAS-based streaming protocols, such as Dynamic Adaptive Streaming over HTTP (DASH), and HTTP Live Streaming (HLS), are too slow at adapting the transmission rate in order to respond to network congestion. First, clients need to explicitly request lower quality segments, which adds at least one Round-Trip Time (RTT) before the client receives the lower quality segment. Second, applications using TCP as the transport layer protocol can't abort the sending of data, which has already been pushed to the TCP socket, unless the application terminates the connection. If the client requests a high quality video segment, and the network bandwdith suddendly drops, the full video segment has to be downloaded regardless.

QUIC enables new streaming methods that are better equipped to respond to network conditions. However new streaming protocols need to be designed to fully leverage QUIC's features. Media over QUIC was developed to bridge this gap. The protocol is still in its early days, but the potential is promising. However, there doesn't exist a consensus on how to best use MoQ to stream media and as far as we know there isn't much work on comparing and evaluating different streaming protocols that use MoQ.

In this thesis, we analyse and evaluate three streaming protocols using Media over QUIC. We make the following contributions:

- We describe the architecture and implementation of a MoQ-based live streaming system (Chapter 3). We explain the inner workings of our prototype, as well as subtle implementation details that might not be immediately obvious. In MoQ systems, the client has full control of how to render media, which gives a lot of flexibility and control to the application, but also increases the surface area of the implementation compared to the off-the-shelf players available for HAS-based streaming methods. We haven't found any work on the challenges of implementing a MoQ-based web player. An additional goal we have is to describe this.

- We describe two self-adaptive techniques to prioritize latency. We propose an approach that consists in deprioritizing B-frames to degrade the quality of the video stream when the network is congested (4). We also describe an approach that skips old media segments after a period of congestion (5).
- We evaluate our approaches and show that they can achieve lower latencies (6). For each approach, we simulate multiple network environments and measure relevant metrics. We also present the testbed that we've used for this purpose. Additionally, we present a qualitative evaluation of our approaches, where we discuss some disadvantages that are not represented in our measurements.

Media streaming falls broadly into two categories: live streaming and Video on Demand (VOD). In this thesis, we only concern ourselves with the former. VOD has other challenges and thus the streaming protocols are different. Achieving a low latency, which is defined as the delay between when video is captured and when video is played, is a non-goal in the context of VOD, since the video is pre-recorded and watched later.

Furthermore, we don't cover all components of a live streaming system, that are nonetheless necessary for a fully working system. We assume a simple architecture, consisting solely of a client and an origin server. We don't discuss the use of relays, which are crucial to scale the delivery of media to a large number of users. In this work, we also restrict ourselves to the streaming of video content. We don't consider audio, and how it interacts with video. In addition, our focus lies on the media distribution part of the pipeline. We don't discuss media contribution or ingestion. In our prototype contribution happens at the server, such that media doesn't need to be ingested over the network.

2 Background

2.1 Video coding standards and formats

Raw uncompressed video is too large to be transmitted over the network. Video compression algorithms reduce the size of video content by reducing the redundancy in video frames. Two popular codings include H.264/AVC [23] and HEVC [21]. Encoded video consists of a sequence of independently decodable units called Groups of Pictures (GoPs). A GoP contains intra-coded pictures or I-frames, predicted pictures or P-frames, and bidirectional predicted pictures or B-frames. An I-frame is a fully self-contained image, P-frames reference previous frames, and B-frames reference both previous and future frames. Since B-frames reference future frames, the encoder can only produce them, after the future frames that are referenced.

The compressed video content can be stored in several formats. MP4 is the most commonly used container format to store multimedia content, consisting of audio and video streams. The audio and video streams each correspond to a track in the MP4 format. MP4 uses a box model. A typical video file contains a ftyp box, identifying the compatible file type specifications, a mdat box, containing the actual audio/video payload, and a moov box, describing the audio/video frames or samples in the mdat box. The moov box contains information relevant to the media content as a whole such as the tracks, the codec used, and information for every frame such as the size, timestamp or duration of the frame.

The regular file structure of a MP4 file is not suitable for live streaming, because the moov box, which describes every frame in the media content, cannot be produced until all frames have become available. Fragmented MP4 packages media in a format suitable for live streaming, by dividing the media into chunks called fragments, which can be produced independently. These fragments are pairs of moof and mdat boxes. The moof box is similar to the moov box, except that it only the describes the payload of the associated mdat box. The moov box in fragmented MP4 is used to describe information common to all audio/video frames such as the codec used. Fragments can contain a varying number of frames depending on the packager configuration.

2.2 HTTP-based live streaming

HTTP-based media streaming is the most popular streaming method today by far. Using a pull-based approach, the client progressively downloads the media stream from an HTTP server. For this purpose, the media stream is split into segments, which are a few seconds long. In the beginning of a session, the client fetches a manifest file that describes the video segments in detail, containing information such as the codec, resolution, and duration of each segment. The manifest file contains all the information necessary for the client to request and play the video segments. The client, then, requests the video segments continuously from the server and plays them.

Furthermore, with HTTP Adaptive Streaming (HAS), the server reencodes the video segments into multiple resolutions and bitrates using a bitrate ladder in order to offer the media stream in multiple qualities. During playback, the client monitors the network throughput to decide which bit rate to choose for the next video segments. This allows the client to request lower quality segments, if the connection doesn't have enough bandwidth, or increase the quality of the stream if the device is capable and if there is enough bandwidth. This process is called Adaptive Bitrate (ABR) Streaming.

The most popular HAS-based methods today are DASH [2] and HLS [14]. Although similar at a high-level, each defines their own format for media segments and manifest files.

HTTP-based live streaming is the dominant streaming method due to the nature of the web infrastructure and HTTP itself. First, HTTP-based live streaming can scale massively, because it can leverage existing CDNs infrastructures and caches. Furthermore, scale is much easier, because HTTP is stateless and all the logic for controlling a session resides in the client, resulting in simple servers and relays. Second, the widespread deployment of HTTP makes an HTTP-based live streaming system easy to deploy. Finally, HTTP-based systems are much cheaper than custom push-based approaches due again to the widespread deployment of HTTP

However, the first versions of HAS streaming protocols such as DASH and HLS did not support low latency. The reason for this is that a video segment could not be delivered until it was fully generated, because it wasn't packaged until all the frames in it were produced. This results in latencies that are at least the segment duration, which is a couple of seconds long. Decreasing the segment duration does decrease the minimum latency, but it is not viable past a certain point, since it increases the number of requests the client needs to perform, impacting the performance of HTTP servers and caches. In addition, decreasing the segment duration also decreases the encoding efficiency.

To address the low-latency requirements that became increasingly more common, low-latency extensions of DASH and HLS, LL-DASH and LL-HLS respectively, were

developed. These extensions leverage Common Media Application Format (CMAF) [1], developed in 2016 to standardize the format of media segments, which breaks segments down into chunks. In essence, the principle behind LL-DASH and LL-HLS is to encode, package and deliver segments in smaller chunks [4, 8]. The encoder makes frames available to the packager immediately after encoding them, the packager in turn packages them into CMAF chunks, containing a couple of frames at most. Finally the chunks are delivered to the client as soon as they become available. In summary, with chunked encoding, packaging, and delivery the latency is no longer determined by the segment duration.

2.3 QUIC and Media over QUIC

QUIC is a connection-oriented transport protocol built on top of UDP, providing for the first time an alternative to TCP. QUIC provides independent streams over a single multiplexed connection, which unlike streams multiplexed over a TPC connection, don't suffer from Head-of-line (HOL) blocking. Furthermore, streams can be prioritized and terminated, giving applications much more control over the transmission of their data. Other features include a faster handshake process, improved performance during network-switching events by using a unique connection identifier, and support for unreliable datagrams.

Systems using TCP may notice some improvements by simply switching to QUIC. QUIC provides lower startup, and seeking delays by starting media streams more quickly, and handles network switching events more gracefully, providing a better Quality of Experience (QoE) for users that are mobile [3]. In addition, QUIC shows a reduced number of stalls and lower stall durations in lossy networks than TCP, providing a better QoE in environments where packet loss is frequent [20].

Nevertheless, transitioning from TCP to QUIC does not result in major improvements out of the box. In network environments with low packet loss, HAS-based applications using QUIC do not perform better than their TCP counterparts [22]. In order to achieve lower latencies and improvements in QoE, custom application-layer protocols need to be developed that fully leverage QUIC's features [18].

Media over QUIC (MoQ) is a relatively recent application-layer protocol designed for low-latency streaming of media. It is designed for various applications such as live streaming, gaming, and media conferencing. An IETF working group, was formed in 2022 [17].

MoQ was designed with a couple of goals in mind to address the challenges with traditional streaming protocols. MoQ aims to define a single transport protocol for media ingestion and distribution, eliminating the need to repackage media at multiple

stages of the streaming pipeline. In addition, MoQ is meant to be highly scalable, by designing the protocol with first-class support for relays.

In addition, MoQ has the potential to achieve lower latencies than the traditional streaming protocols. First, applications are able to map media to multiple QUIC streams, which don't suffer from HOL blocking. Second, MoQ enables new ways to respond to congestion by leveraging QUIC. When the network conditions are not ideal, the latency of a protocol is determined by how fast it can detect and respond to congestion [7]. MoQ enables new ways to respond to congestion, by leveraging QUIC. Using stream prioritization, applications can prioritize the delivery of the most crucial media. Furthermore, applications can drop media by terminating streams to save bandwidth for their most important media. The protocol is flexible, allowing applications to choose wether to prioritize latency or quality.

The Media over QUIC Transport (MOQT) protocol is based on a publish/subscribe workflow. Producers publish media, which clients can subscribe to. Relays simply forward media, providing the link between publishers and subscribers. MoQ represents media using an object model. An object is the smallest unit of data in MOQT, which in the video use case corresponds to the video frames. At the application level, objects might depend on each other, meaning the application can't process object X without having object Y. Groups in MOQT contain objects that depend on each other, which themselves are independent. Groups provide a join point for new subscriptions. A typical configuration, maps GoPs to groups. Finally, groups belong to tracks. A track is simply a sequence of groups that clients can subscribe to. MoQ is intended to be flexible, and therefore leaves the specifics of how the media content is mapped to these primitives up to the application.

3 Implementation

We now turn to the base implementation of our prototype live streaming system. Our prototype uses draft version 3 of MOQT, but the main points outlined in this section should be applicable to versions 4 and 5, the latest version at the time of writing.

We first describe our base implementation. Similar to a traditional live-streaming system that uses TCP, this version transmits the frames reliably and in order, using a single QUIC stream. We also use this section to describe the high-level architecture of the system and the inner workings of the client and server components, which all our approaches share. We start with the server implementation and then we proceed with the client.

3.1 Origin server

The server ingests a live video stream and broadcasts it to clients. Our prototype ingests a live video stream from `stdin`, which is encoded with H.264/AVC and packaged in MP4 fragments. The stream is fragmented at every frame with the ffmpeg option `frag_every_frame`.

The server ingests and parses the livestream, serving subscribed clients, and concurrently listens for new subscription requests. To serve the clients, the server provides two tracks: an "init" track and a "video" track. The init track is used to serve the ftyp and moov boxes, which the server parses and stores in the beginning of the ingestion process. Clients subscribe to the init track on stream startup to configure the decoder.

The video track is used to serve the actual video content. The server parses the MP4 fragments, which correspond to the pairs of moof and mdat boxes, that follow the ftyp and moov boxes, and fans them out to subscribers. Each fragment, which contains exactly one frame, is sent as a MoQ object on the "video" track. The server groups together MoQ objects that belong to the same GoP into the same MoQ group, by incrementing the group number for each keyframe, to provide a join point for new subscriptions. The baseline version uses a single stream for all objects in the video track.

3.2 Live streaming client

We now turn our attention to the client.

Subscribing and playing a live stream works as follows. First, the client downloads the ftyp and moov boxes, by subscribing to the init track. These boxes contain metadata about the tracks in the MP4 stream that the client will later need to parse the mp4 fragments and decode the video frames.

When the user clicks "play", the client subscribes to the video track. Since the player can't start playback in the middle of a GoP, the client can choose between two different locations for which the subscription should start. The client can either wait for the new GoP to be produced or subscribe to the latest GoP. Assuming a GoP to MoQ group mapping, like we do in our server, in MoQ draft 3 we can achieve the former by using the Subscribe location mode RelativeNext and value 0 and to achieve the latter, we use mode RelativePrevious and value 0. Waiting for the next GoP to start ensures the lowest possible latency, while starting playback at the latest GoP keeps startup delays low but it can lead to latencies up to the duration of GoPs. Whether to prioritize startup delay or latency is up to decide based on the use case.

As the client receives the MP4 fragments from the video track, it adds them to a pipeline that processes the raw fragments, extracting the video frames and eventually rendering them. The pipeline consists of three stages.

First, the player extracts the frame payload and frame metadata such as the frame duration, decode timestamp and presentation timestamp from the mp4 fragment.

Then we decode the encoded frame using the frame metadata with the VideoDecoder from the WebCodecs API. We configure the decoder using the ftyp and moov boxes, which we downloaded from the init track. To configure the VideoDecoder, two options are worth mentioning. We enable `optimizeForLatency` and set `hardwareAcceleration` to `"prefer-software"`. Regarding the latter, with the default setting, the VideoDecoder was occasionally throwing errors with the message "Decoding error". With `"prefer-software"` these errors did not occur, although we can't fully explain the reason behind it.

Finally, we render the decoded frame, by drawing it to a canvas element. Rendering frames as they are received results in jerky playback, and consequently poor QoE. To provide smooother playback, we use a jitter buffer and time the rendering of frames, rather than rendering the frames immediately after decoding them. After a frame is received and decoded, it is added to the jitter buffer. Once the buffer size reaches the target size, the player start consuming frames from the buffer. While the buffer has frames, the player continually retrieves the frame with the lowest timestamp from the buffer, and waits for the correct time to render it. If the buffer runs out of frames, the player starts the process of filling the buffer again.

Algorithm 1 shows how the player calculates the time until a frame is to be rendered. The time the player should wait depends on the current media time, which is the elapsed time since playback started in the current session. If the stream can be played without interruptions, the time until a frame is to be rendered is the difference between the frame timestamp and the media time. However, if the player rebuffers, using the absolute media time would cause all frames that are lagging behind to be rendered immediately after the player resumes playback at the same. If the network conditions that led to the rebuffering event are not short-lived, then the player would continuously flush all frames, emptying the buffer, and start buffering again. We handle this issue, by timing the rendering of frames relative to the point in time, at which playback resumed after a rebuffering event, rather than the absolute playback start.

Algorithm 1 Calculate time to render frame

```
function CALCULATETIMEUNTILFRAME(frameTimestamp)
    now ← NOW()

    if resumedPlayingAt = undefined then
        resumedPlayingAt ← {}
        resumedPlayingAt.localTime ← now
        resumedPlayingAt.mediaTime ← frameTimestamp
    end if

    relativeMediaTime ← now − resumedPlayingAt.localTime
    relativeFrameTimestamp ← (frameTimestamp − resumedPlayingAt.mediaTime)/1000
    return max(0, relativeFrameTimestamp − relativeMediaTime)
end function
```

4 Deprioritizing B-frames

In order to prevent the latency from increasing during network congestion, the server needs to send less data to the client. One way to reduce the stream's bitrate is by dropping B-frames. The reason this works is twofold.

First, B-frames are not usually depended by other frames, and therefore they can be dropped without affecting the decodability of other frames. We will assume for the remainder of this section that B-frames are not used as references. We note however that state of the art encoders allow the use of B-frames as references through B-pyramid schemes. We will discuss the use of reference B-frames, including ways to adapt our implementation to support them, towards the end of the section.

Second, the P-, and I-frames alone form themselves a stream that is playable, albeit with video artifacts, such that we can drop all B-frames if necessary. This is only true, if the number of consecutive B-frames is limited, which is the case in low-latency live streaming.

We effectively divide the stream into two layers: a base layer, consisting of the I-, and P-frames, and an enhancement layer containing all B-frames. During congestion, the server drops the enhancement layer, degrading the stream quality to reduce the stream's bitrate.

4.1 Implementation

Our goal is to deprioritize the B-frames, such that they are sent on a best-effort basis. To accomplish this, we prioritize the layers as follows. The base layer is assigned the highest priority so that the server always transmits I-frames and P-frames first, if any are available. On the other hand, B-frames get a lower priority, such that they are transmitted only if there is enough network bandwidth.

We divide the stream into two MoQ tracks, one for each layer. In both tracks each GoP forms a group. The group boundaries must be aligned in this way to provide a join point for new subscriptions that is synchronized across both tracks. The server implements this by incrementing the groupId for each keyframe. At startup, the client subscribes concurrently to both tracks referencing the same MoQ group.

The track that serves the base layer uses a single QUIC stream to deliver the I- and P-frames in order. We assign the second highest priority to this stream. (The highest

priority value is assigned to the init track).

To stream the enhancement layer, one could think of using a single stream with a lower priority. However, this simple prioritization strategy wouldn't have the desired effect. We demonstrate the issue with an example. Suppose that the network was congested and we didn't transmit any B-frames because there wasn't enough bandwidth. When the network recovers, we want to transmit the new B-frames. However the old B-frames are transmitted because they were queued first. If the network has just enough bandwidth to send the B-frames of one GoP, we would never get a chance to render the B-frames. They would always be late. The same reasoning applies to a stream per GoP for the B-frames. If the server gets enough bandwidth to transmit some B-frames towards the end of a GoP, we don't want to send the first frames of the GoP since they are useless.

Each B-frame should have a higher priority than the previous B-frame that was produced. In general, frames in enhancement layers should be prioritized according to the new over old policy, since they are always transmitted on a best-effort basis.

Since, in QUIC, the unit of prioritization is the stream, each B-frame needs to be sent in a separate stream. The decode timestamps of B-frames can be used as the priority for each stream, or if one wants to make optimal use of the number of bits, the current count of B-frames.

To prioritize I-, and P-frames over B-frames, the server must be able to distinguish frames of different types in the first place. Identifying the type of a frame is not as trivial as one might think. Parsing the mp4 fragments is not enough, because there doesn't exist any boxes in the mp4 container that contain this information. One needs to parse the encoded sample. For AVC encoded frames, for example, we first parse the Network Abstraction Layer (NAL) units with the type 0, Coded slice of a non-IDR picture, and type 5, Coded slice of an IDR picture, from the mp4 sample. We then parse the slice types from the slice header of these NAL units.

Additionally, the server includes the frame type in the header of the payload of the MoQ object, such that clients, and potentially relays as well, can easily extract the type of a frame from the MoQ object without having to parse the encoded samples.

4.2 Handling out-of-order frames

Because we are using multiple QUIC streams, which are independent and don't provide any ordering guarantees, frames will arrive out-of-order. Within the base layer, I- and P-frames will arrive in decode order, however B-frames can arrive out-of-order relative to the frames in the base layer. Furthermore, B-frames can arrive out-of-order relative to each other. First, if the network becomes temporarily congested such that some

B-frames are not transmitted, then old B-frames will be sent to the client when the network recovers. Second, a B-frame can arrive ahead of another B-frame that was sent first, because the underlying packets take different network paths or due to packet loss. In this subsection, we propose a solution to this issue. First, we explain the intuition behind our approach, and then we describe it more precisely, by explaining relevant implementation details.

We add frames from the base layer, which arrive in order relative to each other, directly to the render pipeline. Although there might be discontinuities between these frames, we don't wait for the missing B-frames, because they might never arrive.

B-frames can be late or early. We define a B-frame to be late, if at the time of its arrival a frame with a higher timestamp has already arrived. Similarly we say a B-frame is early, if it arrived ahead of frames with a lower timestamp. If a B-frame is late, we drop it, because we don't want to decode and render frames out-of-order. If a B-frame is early and no later I-, or P-frames have arrived, we wait for the earlier frames to arrive. This is because the B-frame might have arrived ahead of a frame from the base layer, which we can't drop. Note that the B-frame might simply be ahead of other B-frames, but we have no way of telling, so we need to assume the worst case. If a B-frame is neither late nor early by our definitions, we add it directly to the render pipeline.

In practice, frames that are in order are not added directly to the render pipeline. Instead frames are first added to a jitter buffer to handle small variations in the arrival of frames. Otherwise, if we add frames directly to the render pipeline, a frame that is late by only a few milliseconds will be dropped. From our experiments, we found a 100 to 200 milliseconds jitter buffer to be optimal.

Algorithm 2 shows pseudocode for the process of handling an incoming frame and retrieving the next frame to be enqueued to the render pipeline. In summary: first, we check if the frame is a B-frame and if the frame is late, by checking if it has a lower decode timestamp than the decode timestamp of the last enqueued frame. If the B-frame is late, we drop it. Otherwise, we add the frame to the jitter buffer. This buffer works like a min-heap with the decode timestamps serving as the key. Then, if the buffer size is greater than the target size, we *try* to retrieve the next frames from the buffer and enqueue them to the render pipeline. To do this, we check the frame with the lowest timestamp in the buffer. If the frame is an I-, or P-frame or if there is no discontinuities between the frame's dts and the last enqueued frames dts, we enqueue the frame to the buffer and try to enqueue the next frame. If the frame is a B-frame and it is early, then we scan the buffer for an I-, or P-frame. If we find an I-, or P-frame, then it must be the case that the frame has a higher timestamp, and we enqueue the frame. Note that the B-frame cannot be ahead of any I-, or P-frames since frames from the base layer are transmitted in order. If, on the other hand, the buffer only has B-frames, then we wait for the missing frames to arrive by returning, since it might be ahead of

I-, or P-frames.

4.3 Reference B-frames

We assumed until now that B-frames are not depended on by other frames. However, H.264 and newer video codings allow B-frames to be used as references for the encoding of other frames. Our current approach, which treats all B-frames as independently droppable, is incompatible with this setting. Recall that if we drop a B-frame, we also need to drop any frames that depend on it. Otherwise, the player would try to decode a frame for which it does not have its dependencies, which would break the player.

Linking B-frames that depend on each other so that if we drop a B-frame, we also drop any B-frames that depend on it, would add too much complexity. It would involve extracting the dependencies between B-frames, which requires parsing the encoded video samples, and implementing a media to streams mapping or prioritization scheme that groups frames and their dependencies together. For this reason, this solution is explicitly a non-goal for us.

One can simply disable the use of B-frames as references. For H.264/AVC, one can control this with the b-pyramid setting. This would slightly decrease the compression rate. We believe this is acceptable in a low-latency live-streaming scenario, because we don't use that many B-frames anyways.

However the above solution might not be possible, if we are not in control of the encoder. Then, an alternative solution, is to only use B-frames that are not used as references in the enhancement layer. To distinguish reference B-frames from non-reference B-frames, one can try using the mp4 sample flags `sample_depends_on` and `sample_is_depended_on`. We note, however, that in our experiments, both flags were set to the value 0, indicating that the dependencies were unknown, for the majority of frames. We don't know if this is a limitation of the library that we used to package the stream into mp4 or if it's something we didn't configure correctly.

Algorithm 2 Reorder Algorithm

```
function REORDER(frame)
    if frame.type = B ∧ lastEnqueued ≠ null ∧ frame.dts < lastEnqueued.dts then
        return                                                  ▷ Drop late B-frames
    end if

    add frame to buffer

    while BUFFERSIZE() > targetSize do
        next ← buffer[0]
        if next.type ≠ B ∨ lastEnqueued = null ∨ next.dts = lastEnqueued.dts +
lastEnqueued.duration then
            enqueue next
            pop buffer
            lastEnqueued ← next
            continue
        end if

        i ← 1                                                   ▷ B-frame is early
        while i < buffer.length ∧ buffer[i].frameType = B do
            i ← i + 1
        end while
        if i < buffer.length then          ▷ B-frame is not ahead of any I-, or P-frames
            enqueue next
            pop buffer
            lastEnqueued ← next
        else                ▷ Wait, because B-frame might be ahead of I-, or P-frames
            break
        end if
    end while
end function

function BUFFERSIZE()
    if buffer.length = 0 then
        return 0
    end if

    oldest ← buffer[0]
    newest ← buffer[buffer.length − 1]
    return newest.dts − oldest.dts
end function
```

5 Skipping old media

Adapting a live stream's quality is an effective technique for prioritizing latency, but it can't handle large changes in the network bandwidth effectively alone. Consider the case where a viewer is watching a live stream on their phone and the available network bandwidth drops drastically and then recovers. This can happen because the viewer is watching the live stream on his phone on the go and temporarily goes past an area of bad coverage for example. While the network connectivity is poor, streaming a lower quality stream surely helps. However video will still lag behind and latency will increase, due to the low network throughput. Video will queue up at the server. When the network recovers, rather than playing all the video that queue up, we want to resume playback at the live edge. We want to skip large portions of media if we don't get a chance to transmit it due to network conditions.

To achieve this, we prioritize new over old media.

Specifically, we map and prioritize the video track as follows. We map each GoP of the video stream to a MoQ group, and send each group in its own QUIC stream. We prioritize new GoPs over old ones, by using the timestamp of the first frame in the group as the priority of the QUIC stream.

In the client, we drop any frames belonging to old GoPs. Handling late GoPs is similar to how we handled out-of-order frames in Section 4.2. One can keep track of the last frame that we enqueued to the processing pipeline and compare the timestamps of new frames with the timestamp of this frame, dropping the new frames if they have a lower timestamp.

5.1 Snapping video forward

Our goal is to render new media immediately, such that plackback jumps from the old media forward to the new media. Recall, however, that the player, as we first described in Section 3.2, times the rendering of frames using the relative frame timestamp. Using Algorithm 1 to time the rendering of new media, would lead to the player waiting for the frame's timestamp before rendering the new GoPs, and thereby not decreasing the latency. On the one hand, we want to render new media immediately. On the other hand, we want to buffer and time consecutive frames according to their presentation time for smoother playback. To achieve this, we compare the presentation timestamp of

the previously rendered frame and the frame that we are about to render. If there is a discontinuity between the presentation timestamps, we render the frame immediately, otherwise we time the rendering of the frame.

To implement this, the client keeps track of the timestamp of the last frame that was rendered and we adapt Algorithm 1, so that it first checks if there is a discontinuity between $frameTimestamp$ and the timestamp of the last rendered frame. If there is a discontinuity, $calculateTimeUntilFrame$ returns 0, so that the frame is rendered immediately. Note that if there is a discontinuity, we also need to update $resumedPlayingAt$ so that we use the correct media time as the reference when timing the next frames.

5.2 Combining both strategies

Ideally, we want the live streaming system to handle both small drops in the network bandwidth and short spikes effectively. Although, we haven't had the time to test our hypothesis, we believe that combining both approaches lead to the most optimal performance. In this section, we describe how we would combine deprioritize B-frames and skip old GoPs. We begin by presenting a new priority scheme that integrates the new over old policy with the priority scheme of deprioritize B-frames. We then propose a way to time frames in the client.

Just like in **??** we create a track for the I-, and P-frames and a track for the B-frames. Now instead of using a stream per track for the base layer, we use a stream per GoP, and prioritize new GoPs over old GoPs. Again, we must assign the highest priorities to the streams in the base layer. For this purpose, we divide the value range of the priority values into two ranges. The values in the highest range are for the streams in the base layer, while the lowest range is for the B-frames. To make optimal use of the available bits, one can allocate more bits for the latter, since the number of B-frames is in general higher than the number of GoPs.

We would like to note that we are aware that this approach might not be viable with certain QUIC implementations, because the prioritization implementation does not use enough bits. quinn-rs for example uses 32 bits for the priority value. These are not enough bits for long duration streams.

Regarding the client, we use the approach described in 5.1 with a small modification. Rather than rendering frames instantly if there is any discontinuity, we time frames if the discontinuity is bigger than a predefined threshold. This is because frames from the base layer might have discontinuities, if we drop B-frames, and we don't want to render them immediately.

6 Evaluation

6.1 Testbed

We now describe the testbed that we've used for our experiments. The client and server run on the same machine and communicate through a congested link, simulated using PF and dummynet.

We measure the latency of each frame, as the delay between the frame reaching the server and being rendered in the client. For this purpose, the server includes the availability time, the time at which the frame became available in the server, in the header of the MoQ object's payload. In the client, we calculate the latency for a given frame by taking the difference of the availability time and the time at which the frame was rendered. Note that clock drift is not an issue, as both the server and client run on the same machine and use the same clock.

We've used the Big Buck Bunny video with a frame rate of 24 frames per second. We use ffmpeg to reencode the video and produce the output stream that the server ingests. We use the `-re` to simulate live streaming. The most important output options include:

- `-an`: discard the audio track.
- `-c:v libx264 -b:v 600k -bufsize 200K`: reencode the video using a target bitrate of 600 Kbit/s and a buffer size of 200 Kbit/s. The libx264 encoder is used.
- `-g:v 15 -keyint_min:v 15 -sc_threshold:v 0`: set the minimum and maximum GoP size to 15 frames, and disables scene change detection so that each GoP has exactly the same size.
- `-bf 3`: set the maximum number of consecutive B-frames to 3.
- `-f mp4 -movflags cmaf+frag_every_frame`: use CMAF compatible fragmented MP4 as the output format and package each frame in a separate fragment.

The jitter buffer and also the reorder buffer, in the deprioritize B-frames approach, have a size of 100 ms.

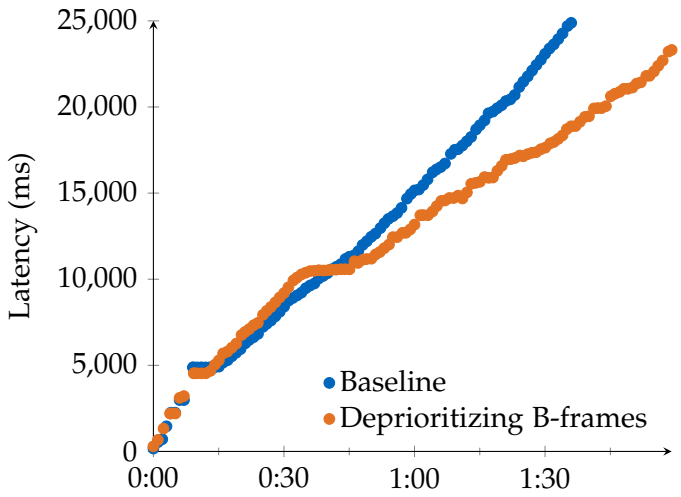

Figure 6.1: Sample run of deprioritizing B-frames for profile 500 Kbit/s - Latency

6.2 Measurements

We now present our measurements. We first analyse the deprioritize B-frames approach and we present data for two sample runs one for the profile 500 Kbit/s and one for the profile INTRA-CASCADE. Then we present data for a sample run of the skip old GoPs approach. Finally, we show the average latency for all profiles.

Deprioritizing B-frames results in lower latencies when the network bandwidth is a little below the video stream's bitrate. Figure 6.1 shows the latency over time for sample runs of baseline and deprioritize B-frames. In these test runs, the bandwidth is limited to 500 Kbit/s, which is 100 Kbit/s below the target bitrate that we specifed in ffmpeg. In the beginning of the test, the latency for both versions is similar, until around 0:40, when they diverge with the latency for deprioritize B-frames increasing at a slower rate than the latency for baseline. The gap between the two versions increases steadily. At the 1:30 mark, the latency for deprioritize B-frames is about 5 seconds lower than the latency for baseline.

Taking a look at the frames that the clients in both approaches receive over time explains why. Figure 6.2 shows the bitrate of frames that reached the client by frame type. Around the 0:40 mark, the client in deprioritize B-frames stops receiving B-frames, which indicates that at this point the server stopped transmitting them. Because the server is not transmitting any B-frames, more bandwidth is left for the base layer. Between 1:25 and 1:35 for example, baseline receives around 200 Kbit/s of B-frames 200 Kbit/s of P-frames, while deprioritize B-frames receives 0 Kbit/s of B-frames and 300 Kbit/s of P-frames, which is 100 Kbit/s more than baseline. Consequently, with

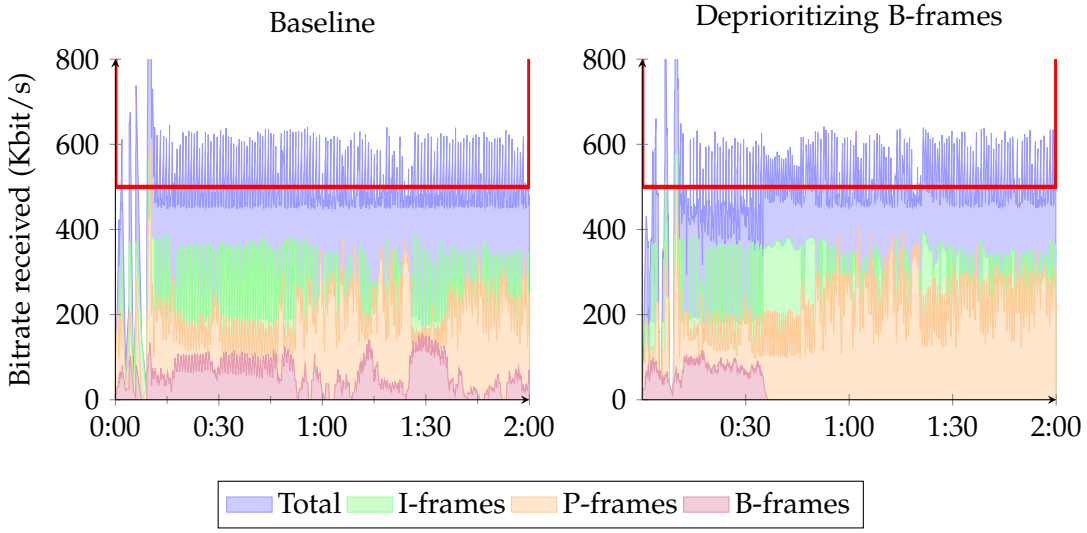

Figure 6.2: Sample run of deprioritizing B-frames for profile 500 Kbit/s - Bitrate received

deprioritize B-frames new frames are sent to the client sooner than with baseline, such that the latency increases slower.

However, deprioritizing B-frames alone does not perform much better than baseline when the network bandwidth drops significantly below the stream's bitrate. The reason for this is that B-frames make up only a small portion of the stream's total bitrate compared to I-, and P-frames. Figure 6.3 shows the latency over time for sample runs of baseline and deprioritize B-frames for the profile INTRA-CASCADE. Both versions show similar latencies from 0:00 to 1:30. At around 1:30, after the bandwidth increased from 400 Kbit/s to 600 Kbit/s, the latency stops growing for deprioritize B-frames, while for baseline it keeps increasing until 1:45.

Once again, the bitrate received by the client explains this. Figure 6.4 shows the bitrate received by the client. At the 1:07 mark, the client in deprioritize B-frames client stops receiving B-frames, indicating that the server has stopped transmitting them. From this point onwards, the server only transmits frames from the base layer to the client. When the network bandwidth increases from 400 Kbit/s to 600 Kbit/s, the latency stops increasing for deprioritize B-frames, because 600 Kbit/s is above the base layer's bitrate. 600 Kbit/s is not big enough to be greater than the base layers plus the B-frames, which the server in baseline is transmitting, and therefore the latency keeps increasing until the bandwidth increases to 800 Kbit/s at 1:45.

Most importantly, the latency does not ever decrease for either approach, even when the network bandwidth increases above the stream's bitrate at 1:30 and 1:45 respectively.

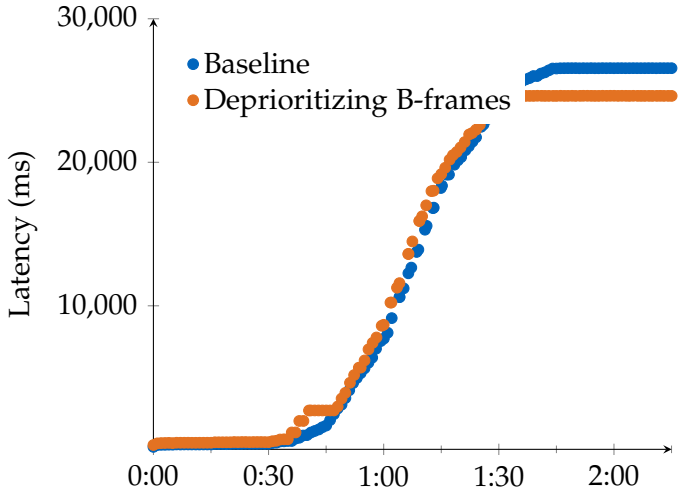

Figure 6.3: Sample run of deprioritizing B-frames for profile INTRA-CASCADE - Latency

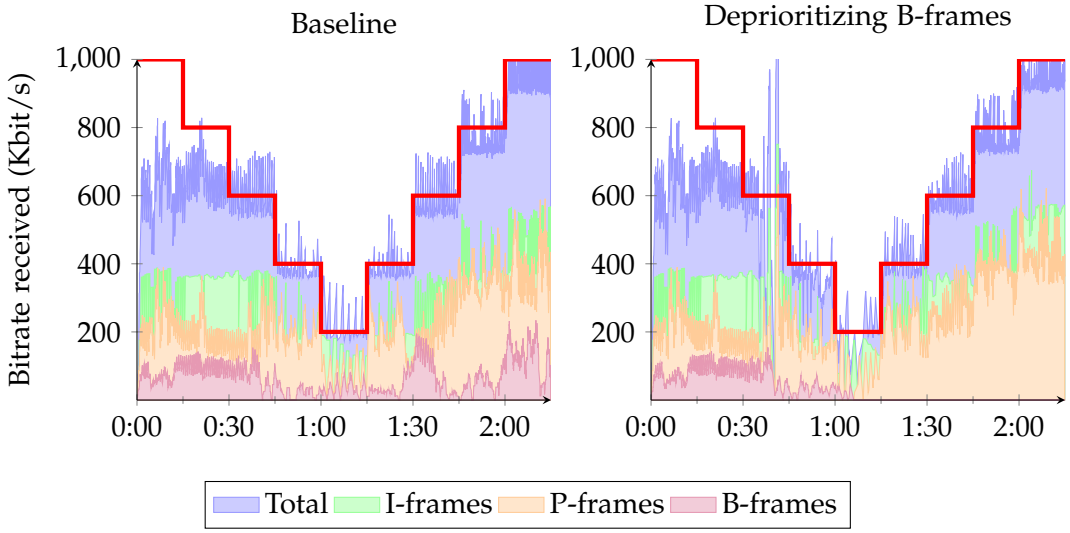

Figure 6.4: Sample run of deprioritizing B-frames for profile INTRA-CASCADE - Bitrate received

This is because, in the client, both baseline and deprioritize B-frames play old video before playing new video and never skip any video segments.

Thus, the bottom line is that to ensure low latencies adapting the stream quality is not sufficient by itself. We need to skip old media to handle the cases in which the network bandwidth decreases significantly.

Skipping old GoPs decreases latency considerably in all types of network conditions. Figure 6.5 shows the latency over time for a sample run of skip old GoPs for the profile INTRA-CASCADE and the same sample run of baseline. In contrast to baseline, the latency doesn't increase indefinitely from 0:30 onwards. Instead the latency per frame fluctuates, with the average latency being around 1.5 seconds throughout the middle of the test. Additionally, the latency decreases back to its minimum at 1:45.

Figure 6.6 marks the I-frames in the latency graph. In general, the latency for each frame increases monotonically until the next I-frame, and then decreases. The maximum latency is therefore bound by the size of the GoP. Smaller GoPs result in lower latencies.

This is because I-frames are always transmitted as soon as the encoder produces them. Figure 6.7 plots the pts of each frame that the client received against the local test time. For skip old GoPs the frames pts grows, in general, at exactly the same rate as the local time. Note that, as previously explained, the latency grows between every I-frame, however these small increases in latency are not visible in the graph. In contrast, the rate at which the frames pts for baseline increases slowly decreases beginning at 0:45.

Note as well that in skip old GoPs the client does not receive any frames between 0:63 and 0:75. The reason for this is that the network bandwidth during this period is so low that there is not enough bandwidth to transmit the I-frame of the current GoP before the I-frame of the next GoP becomes available. The average size of I-frames throughout this test was around 173.76 Kbit. At 200 Kbit/s the server would need $173.76 Kbit/s / 200 Kbit/s \approx 0,87 s$ to transmit one I-frame. Since the frame rate is 24 frames per second, and the GoP size is 15 frames, the encoder produces a new I-frame every $1/24 * 15 = 0.625$ seconds though. As a result the server starts transmitting the next I-frame, which has a higher priority, before it has finished transmitting the current one. Hence, no frames are fully transmitted during this period. This is the most significant disadvantage of our priority scheme. During this period baseline actually performs better than skip old GoPs.

We now present the results of each approach for the twitch profiles. Figure 6.8 shows the average latency for each of our versions. Skip old GoPs is a clear winner, consistently showing lower latencies for all profiles. The difference is specially noticable for the CASCADE and INTRA-CASCADE profiles. On the other hand, deprioritize B-frames performs almost the same as baseline for all profiles.

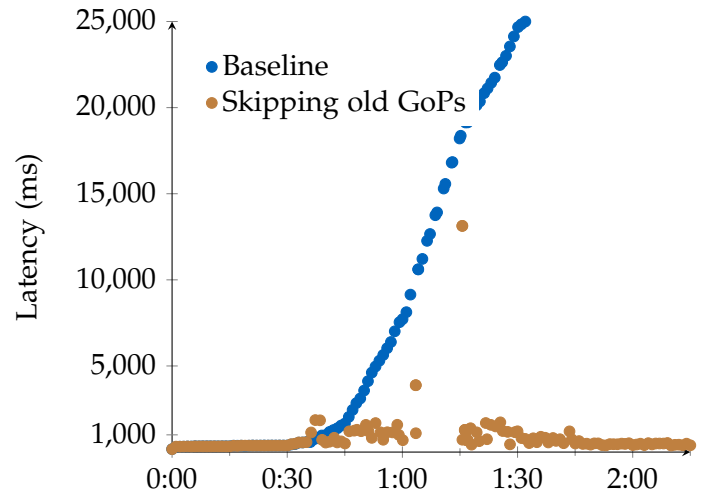

Figure 6.5: Sample run of skipping old GoPs for profile INTRA-CASCADE - Latency

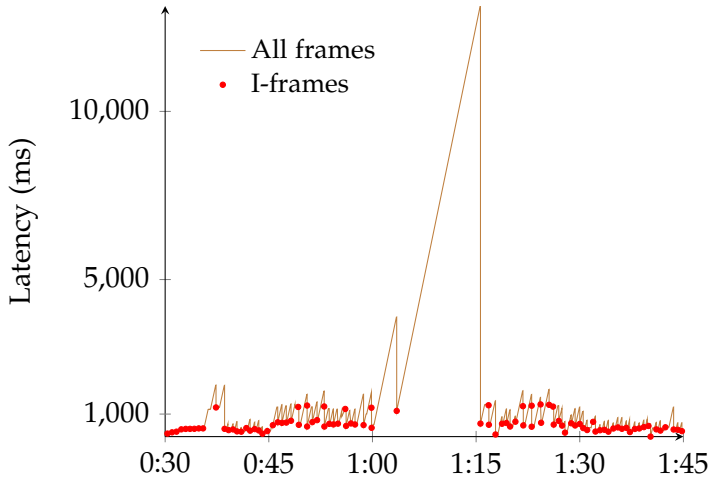

Figure 6.6: Sample run of skipping old GoPs for profile INTRA-CASCADE - Latency with I-frames

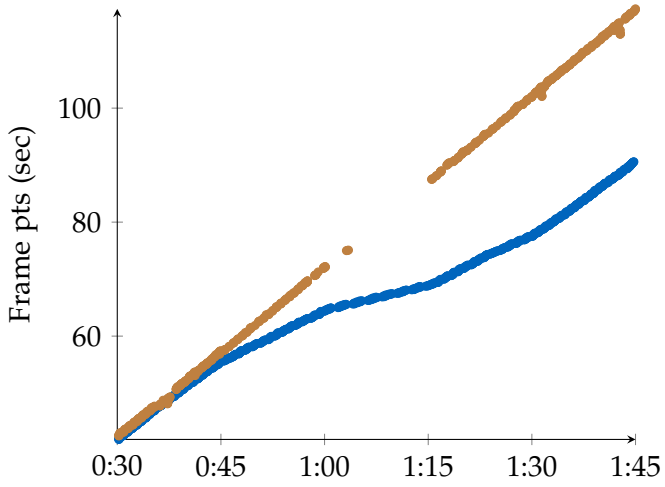

Figure 6.7: Frames received pts

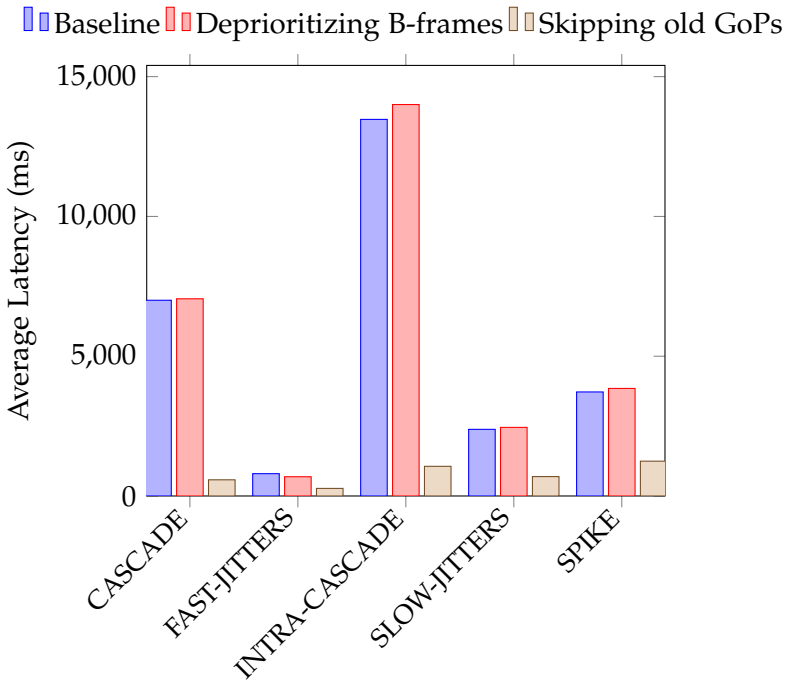

Figure 6.8: Average latency (all profiles)

6.3 Qualitative evaluation

Having presented the measurements, we now discuss additional factors that influence the viability of each approach, but are hard to measure.

Against deprioritizing B-frames:

- Introduces temporal artifacts.
- Requires an encoding configuration that slightly increases the minimum latency. Deprioritize B-frames requires the encoder to produce some B-frames and the degree to which the system is able to cope with unfavourable network conditions is proportional to the size of the enhancement layer and therefore the number of B-frames. However, B-frames reference future frames, and therefore the encoder cannot produce them until after the future frames, meaning that B-frames add latency.
- The server has to understand and parse the codec with which the samples are encoded with to extract the type and dependencies of each frame. This adds complexity and creates a dependency on the video codec.

Against skipping old GoPs:

- Excessive "warping" – suddenly snapping forward to catch up to the live edge, when the video pauses or lags behind – leads to a poor QoE. It is particularly noticable when the network bandwidth is extremely low and these jumps are frequent.

7 Related work

Some QUIC-based media streaming protocols habe been developed prior to MoQ. Warp [6], developed by Twitch, was the first to propose mapping each video segment to a separate QUIC stream, and prioritizing newer streams over older ones for live streaming. RUSH [19], developed by Facebook, another media streaming protocol using QUIC was designed primarily for media ingestion. In additional to the standard configuration, in which a single stream is used to stream the media content, RUSH proposed a *Multi Stream Mode* that maps each audio/video frame to a separate stream. The client then reassembles the frames in the right order. The RUSH draft does not make any mention of stream prioritization.

In [13], Gurel et al. tested a Warp-based MoQ prototype and compared server- to client-side approaches to rate adaptation and bandwidth measurements. Later on, Gurel et al. showed that stream prioritization could lead to increased performance [11, 10]. In this work, a separate stream is used for each frame type. The stream that is used for I-frames has the highest priority, while the stream used to transmit the B-frames has the lowest priority. They showed that their prioritization scheme consistently achieves higher On-Time-Display Ratio (OTDR)s than sending frames based on the encoding order. [12] proposes a testbed to compare the performance of LL-DASH and a MoQ implementation. The MoQ configuration used consists in a stream per GoP, and prioritizes newer GoPs over older GoPs.

Other MoQ implementations include moq-rs [16], a Rust implementation of MOQT, origin server, relay, and other components, and moq-js [15] the corresponding client-side implementation, written in Javascript. This implementation is based on the Warp streaming protocol, with moq-pub mapping each GoP to a stream and prioritizing new over old GoPs. Another implementation developed by Meta to experiment with MoQ be found in [9].

Abbreviations

ABR Adaptive Bitrate

CMAF Common Media Application Format

DASH Dynamic Adaptive Streaming over HTTP

GoP Group of Pictures

HAS HTTP Adaptive Streaming

HLS HTTP Live Streaming

HOL Head-of-line

MoQ Media over QUIC

MOQT Media over QUIC Transport

NAL Network Abstraction Layer

OTDR On-Time-Display Ratio

QoE Quality of Experience

RTT Round-Trip Time

VOD Video on Demand

List of Figures

List of Tables

Bibliography

[1] 14:00-17:00. *ISO/IEC 23000-19:2024*. ISO. URL: https://www.iso.org/standard/85623.html (visited on 08/31/2024).

[2] 14:00-17:00. *ISO/IEC 23009-1:2022*. ISO. URL: https://www.iso.org/standard/83314.html (visited on 08/31/2024).

[3] Ş. Arısu and A. Begen. "Quickly Starting Media Streams Using QUIC." In: June 12, 2018, pp. 1–6. DOI: 10.1145/3210424.3210426.

[4] A. Bentaleb, M. Lim, M. N. Akcay, A. C. Begen, S. Hammoudi, and R. Zimmermann. *Toward One-Second Latency: Evolution of Live Media Streaming*. Oct. 4, 2023. DOI: 10.48550/arXiv.2310.03256. arXiv: 2310.03256 [cs]. URL: http://arxiv.org/abs/2310.03256 (visited on 08/31/2024). Pre-published.

[5] "Cisco Visual Networking Index: Forecast and Trends, 2017–2022." In: (2018).

[6] L. Curley. *Warp - Segmented Live Video Transport*. Internet Draft draft-lcurley-warp-00. Internet Engineering Task Force, Feb. 10, 2022. 9 pp.

[7] L. Curley, K. Pugin, S. Nandakumar, V. Vasiliev, and I. Swett. *Media over QUIC Transport*. Internet Draft draft-ietf-moq-transport-05. Internet Engineering Task Force, July 8, 2024. 42 pp.

[8] K. Durak, M. N. Akcay, Y. K. Erinc, B. Pekel, and A. C. Begen. "Evaluating the Performance of Apple's Low-Latency HLS." In: *2020 IEEE 22nd International Workshop on Multimedia Signal Processing (MMSP)*. 2020 IEEE 22nd International Workshop on Multimedia Signal Processing (MMSP). Sept. 2020, pp. 1–6. DOI: 10.1109/MMSP48831.2020.9287117.

[9] *Facebookexperimental/Moq-Encoder-Player*. Meta Experimental, Aug. 28, 2024.

[10] Z. Gurel, T. E. Civelek, and A. C. Begen. "This Is The Way: Prioritization in Media-over-QUIC Transport." In: *Proceedings of the 3rd Mile-High Video Conference on Zzz*. MHV '24: Mile-High Video Conference. Denver CO USA: ACM, Feb. 11, 2024, pp. 113–114. ISBN: 9798400704932. DOI: 10.1145/3638036.3640280.

[11] Z. Gurel, T. E. Civelek, A. C. Begen, and A. G. 1. September 2023. *IBC2023 Tech Papers: A Fresh Look at Live Sports Streaming with Prioritized Media-Over-QUIC Transport*. IBC. URL: https://www.ibc.org/technical-papers/ibc2023-tech-papers-a-fresh-look-at-live-sports-streaming-with-prioritized-media-over-quic-transport/10264.article (visited on 09/01/2024).

[12] Z. Gurel, T. E. Civelek, D. Ugur, Y. K. Erinc, and A. C. Begen. "Media-over-QUIC Transport vs. Low-Latency DASH: A Deathmatch Testbed." In: *Proceedings of the 15th ACM Multimedia Systems Conference*. MMSys '24. New York, NY, USA: Association for Computing Machinery, Apr. 17, 2024, pp. 448–452. ISBN: 9798400704123. DOI: 10.1145/3625468.3652191.

[13] Z. Gurel, T. Erkilic Civelek, A. Bodur, S. Bilgin, D. Yeniceri, and A. C. Begen. "Media over QUIC: Initial Testing, Findings and Results." In: *Proceedings of the 14th ACM Multimedia Systems Conference*. MMSys '23. New York, NY, USA: Association for Computing Machinery, June 8, 2023, pp. 301–306. ISBN: 9798400701481. DOI: 10.1145/3587819.3593937.

[14] A. Inc. *HTTP Live Streaming (HLS)*. Apple Developer. URL: https://developer.apple.com/streaming/ (visited on 08/31/2024).

[15] kixelated. *Kixelated/Moq-Js*. Aug. 22, 2024.

[16] kixelated. *Kixelated/Moq-Rs*. Aug. 29, 2024.

[17] *Media Over QUIC (Moq)*. URL: https://datatracker.ietf.org/wg/moq/about/ (visited on 08/29/2024).

[18] M. Nguyen, C. Timmerer, S. Pham, D. Silhavy, and A. C. Begen. "Take the Red Pill for H3 and See How Deep the Rabbit Hole Goes." In: *Proceedings of the 1st Mile-High Video Conference*. MHV '22. New York, NY, USA: Association for Computing Machinery, Mar. 17, 2022, pp. 7–12. ISBN: 978-1-4503-9222-8. DOI: 10.1145/3510450.3517302.

[19] K. Pugin, A. Frindell, J. Cenzano, and J. Weissman. *RUSH - Reliable (Unreliable) Streaming Protocol*. Internet Draft draft-kpugin-rush-00. Internet Engineering Task Force, July 12, 2021. 16 pp.

[20] T. Shreedhar, R. Panda, S. Podanev, and V. Bajpai. "Evaluating QUIC Performance Over Web, Cloud Storage, and Video Workloads." In: *IEEE Transactions on Network and Service Management* 19.2 (June 2022), pp. 1366–1381. ISSN: 1932-4537. DOI: 10.1109/TNSM.2021.3134562.

[21] G. J. Sullivan, J.-R. Ohm, W.-J. Han, and T. Wiegand. "Overview of the High Efficiency Video Coding (HEVC) Standard." In: *IEEE Transactions on Circuits and Systems for Video Technology* 22.12 (Dec. 2012), pp. 1649–1668. ISSN: 1558-2205. DOI: 10.1109/TCSVT.2012.2221191.

[22] C. Timmerer and A. Bertoni. *Advanced Transport Options for the Dynamic Adaptive Streaming over HTTP*. June 1, 2016. DOI: 10.48550/arXiv.1606.00264. arXiv: 1606.00264 [cs]. URL: http://arxiv.org/abs/1606.00264 (visited on 08/29/2024). Pre-published.

[23] T. Wiegand, G. Sullivan, G. Bjontegaard, and A. Luthra. "Overview of the H.264/AVC Video Coding Standard." In: *IEEE Transactions on Circuits and Systems for Video Technology* 13.7 (July 2003), pp. 560–576. ISSN: 1558-2205. DOI: 10.1109/TCSVT.2003.815165.