SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Low-Latency Live Streaming Using Media over QUIC

Vicente Almeida

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Low-Latency Live Streaming Using Media over QUIC

Live-Streaming mit niedriger Latenz über Media over QUIC

Author:	Vicente Almeida
Supervisor:	Prof. Dr.-Ing. Jörg Ott
Advisor:	M.Sc. Mathis Engelbart
Submission Date:	15.09.2024

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15.09.2024 Vicente Almeida

Acknowledgments

I want to thank Mathis, my supervisor, for his invaluable guidance, and Zita, my grandmother, whose kind words never failed to encourage me.

Abstract

HTTP Adaptive Streaming (HAS) is the de facto standard for delivering media content at scale over the Internet. Nevertheless, HAS systems, which were originally developed for Video on Demand (VOD), are poorly suited for low-latency live streaming because they are slow at adapting and responding to congestion.

To address the limitations of HAS, Media over QUIC (MoQ) aims to develop a scalable protocol designed for media distribution that leverages QUIC to achieve lower latencies. In this thesis, we implement a prototype live streaming system using MoQ and examine various prioritization strategies for reducing latency. One strategy is to divide the stream into a base layer and an optional enhancement layer, which can be deprioritized when the network is congested. We explore a concrete implementation of this approach that consists in transmitting B-frames using lower priority QUIC streams. Another strategy we explore is to prioritize newer video segments over older ones. We implement a testbed to simulate various network profiles and measure the reductions in latency achieved by these strategies.

Our results show that deprioritizing B-frames achieves better performance than transmitting frames in their encoded order when the network bandwidth drops slightly below the stream's bitrate. However, deprioritizing B-frames does not lead to a significant latency reduction for network profiles that more accurately represent real-world network conditions since B-frames don't contribute much to the bitrate of the video stream. On the other hand, prioritizing newer video segments over older ones reduces latency significantly, especially when the network bandwidth is below the stream's bitrate, resulting in a maximum latency close to the segment duration.

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 [1]. 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 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. HTTP Adaptive Streaming (HAS) 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 bandwidth suddenly 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 (MoQ) 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 analyze 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 provides the application with a high degree of flexibility 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 these.

- We present 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 (Section 3.3). We also describe an approach that skips old media segments after a period of congestion (Section 3.4).
- We evaluate our approaches and show that they can achieve lower latencies (Chapter 4). 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 the video is captured and when the 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 crucial in a production 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 necessary 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 [2] and H.265/HEVC [3]. 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 bi-directional 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 [4], formally known as MPEG-4 Part 14, 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 or video payload, and a *moov* box, describing the audio samples or video frames in the mdat box. The moov box contains information relevant to the media content as a whole, such as which tracks it consists of, the codec used, and information for every frame, such as the size, timestamp, or duration of the frame.

The regular file structure of an 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 describes the payload of the associated mdat box. The moov box in fragmented MP4 is used to describe information common to all audio or 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. At 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 continuously requests the video segments 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 [5] and HLS [6]. Although similar at a high level, each defines its 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 infrastructures and caches of Content Delivery Networks (CDNs). 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. 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, namely LL-DASH and LL-HLS, were developed. These extensions leverage the Common Media Application Format (CMAF)

[7], 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 [8, 9]. 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 [10] is a connection-oriented transport protocol built on top of UDP, providing an alternative to TCP. QUIC provides independent streams over a single multiplexed connection, which, unlike streams multiplexed over a TCP 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 [11]. 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 [12].

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 [13]. In order to achieve lower latencies and improvements in QoE, custom application-layer protocols need to be developed that fully leverage QUIC's features [14].

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 Internet Engineering Task Force (IETF) working group was formed in 2022 [15].

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.

Furthermore, 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 [16]. 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 whether to prioritize latency or quality.

The Media over QUIC Transport (MOQT) protocol is based on a publish/subscribe workflow. Producers publish media that 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 implementation of our prototype live streaming system. We first give a high-level overview of the architecture, and then we proceed with discussing in more depth the inner workings of the client and the server. We will describe the concrete streaming protocols in later sections. Our prototype uses draft version 3 of MOQT, but the main points outlined in this section should be applicable to later versions as well.

The architecture of our prototype live streaming system can be seen in Figure 3.1. The origin server ingests a live video stream through stdin and broadcasts it to clients using MoQ. The video stream is transmitted directly from the origin server to clients without passing to relays. The live video stream is produced with FFmpeg[1], which reads a source video at the native frame rate to simulate live streaming. We reencode the source video with H.264/AVC and package it in MP4 fragments.

3.1 Origin server

The origin server serves a single broadcast. It uses two MoQ tracks for this purpose: the *init* track and the *video* track. The init track is used to transmit the ftyp and moov boxes to clients, which contain the information that the client needs to parse the MP4 fragments and decode the video frames. Similar to how clients in a HAS system fetch the manifest file, clients subscribe to the init track on startup. The video track is used to serve the actual video content.

The origin server is implemented in Rust and uses moq-transport[2], a Rust crate that implements the MOQT protocol. The server ingests and broadcasts the stream as follows. First, the server creates a `Track` for the init and video tracks. A `Track` is an abstraction provided by moq-transport to fan out MoQ objects to multiple subscriptions. Using the `Track` API, an application writes objects to the track using a writer handle, and moq-transport notifies any readers reading the track using the reader handles. After creating the tracks, the server starts and executes two processes concurrently.

The server reads the video stream from standard input (stdin), parses the MP4 boxes

[1] `https://www.ffmpeg.org/`
[2] `https://crates.io/crates/moq-transport`

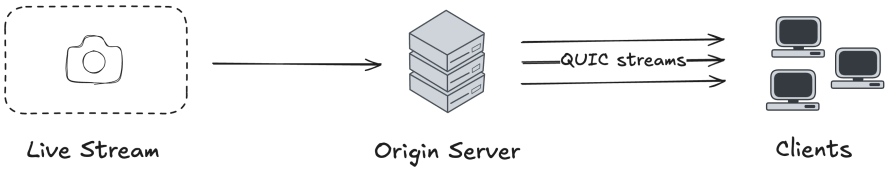


Figure 3.1: Architecture of our prototype system

and writes them to the respective `Track`. First, the ftyp and moov boxes are ingested. The server writes a MoQ object containing both boxes to the init track. Following the ftyp and moov boxes is a stream of moof and mdat boxes, with each pair corresponding to an MP4 fragment. The MP4 stream is fragmented at every frame with the FFmpeg option `frag_every_frame`[3] so that each MP4 fragment contains a single video frame. If the frame contained in the MP4 fragment is a keyframe, the server creates a new MoQ group. Keyframes always introduce new MoQ groups so that clients can start subscriptions at the latest possible point in time, which is the start of the most recent group. For each fragment that is ingested, using a handle to the current group, the server writes an object to the video `Track` with the fragment as the object's payload.

Simultaneously, the origin server listens to new subscription requests and serves existing subscriptions by broadcasting the local `Track`. For each subscription request that the server receives, it creates an asynchronous, non-blocking *task*, a form of a lightweight thread in Rust, to serve the init or video track to the client. In this task, the server reads the MoQ objects from the local `Track`, which are being written in the ingestion process, and transmits them to the client using the existing QUIC connection. To transmit the MoQ object, the server opens a new QUIC stream or uses an existing one, depending on the media to streams mapping configuration. Our baseline implementation transmits the frames reliably and in order, using a single QUIC stream, similar to a traditional live-streaming system that uses TCP. We explore other ways to map media to streams and prioritize these in sections 3.3 and 3.4.

[3] `https://ffmpeg.org/ffmpeg-formats.html#Options-6`

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.

To initialize playback, the client subscribes to the video track, indicating the *subscribe location*, which specifies where in the track the subscription should start. Since the player can't start playback in the middle of a GoP, two different subscribe locations are possible. The client can wait for the new GoP to be produced by using the mode `RelativeNext` and the value `0`, or it can subscribe to the latest GoP with the mode `RelativePrevious` and the value `0`. On the one hand, waiting for a new GoP to start ensures the lowest possible latency. On the other hand, starting playback at the latest GoP keeps startup delays low but it can lead to latencies up to the duration of a GoP. Whether to prioritize startup delay or latency should be decided 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 the frame metadata, such as the frame duration, decode timestamp, and presentation timestamp, from the MP4 fragment.

Then, it decodes the encoded frame with the frame metadata using 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"`. We used the latter setting because the VideoDecoder occasionally threw decoding errors on our test machine when hardware acceleration was enabled.

Finally, the client renders the decoded frame by drawing it to a canvas element. Rather than rendering the frames directly, we buffer and time the rendering of frames to handle packet jitter, or small variations in the delay of received packets. This ensures that frames are rendered at regular intervals, resulting in smoother playback. After decoding a frame, the player adds it to the buffer. Once the buffer reaches a target size, the player starts consuming frames by retrieving the frame with the lowest timestamp from the buffer and waiting for the correct time to render it. If the buffer runs out of frames, the player stops retrieving frames and waits until the buffer reaches the target size 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. 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
```

3.3 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 I- and P-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.

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

To implement this, 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. 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 don't exist any boxes in the MP4 container that contain this information. One needs to parse the encoded sample. For H.264/AVC encoded frames, for example, we first parse the Network Abstraction Layer (NAL) units with the type 0 or 5, which correspond to coded slices of a non-IDR picture and coded slices of an IDR picture respectively, 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.

3.3.1 Handling out-of-order frames

Because we are using multiple QUIC streams, which are independent and don't provide any ordering guarantees, frames may 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 arrives 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.

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
```

In practice, frames that are in order are not added directly to the render pipeline. Instead, frames are first added to a 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 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. 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 buffer. This buffer works like a min-heap, with the decode timestamps (DTS) 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 are no discontinuities between the frame's DTS and the DTS of the last enqueued frame, 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.

3.3.2 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. Dropping a B-frame that is used as a reference while trying to decode the dependent frames will cause video distortions. A decoder can't decode a frame without its dependencies without errors. The approach we described so far doesn't distinguish between reference and non-reference B-frames, and therefore, video distortions will occur when the network is congested.

The simplest solution is to disable the use of B-frames as references, at the cost of a slightly lower coding efficiency. For H.264/AVC, one can control this with the b-pyramid setting[4]. In live streaming, the decrease in coding efficiency resulting from disabling this option is limited since we use a relatively low number of B-frames.

However, one might not be able to configure the encoder settings. A production live streaming platform, for example, might ingest streams from a variety of sources, of

[4] `https://en.wikibooks.org/wiki/MeGUI/x264_Settings#b-pyramid`

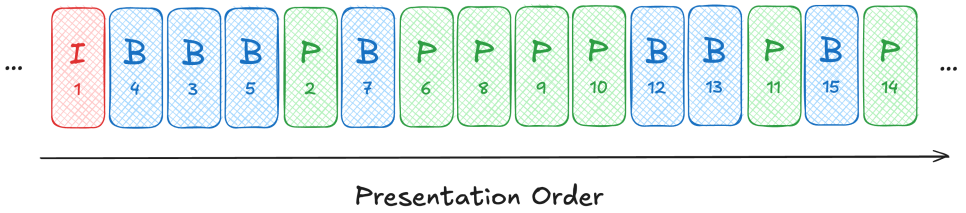


Figure 3.2: Example GoP with frames and their decode order

which the user is in control of the encoding configuration. In these cases, one must design a streaming protocol that handles reference B-frames properly.

One approach consists of distinguishing reference from non-reference B-frames and transmitting reference B-frames together with the I- and P-frames. In the client, we wait for any reference B-frames if a non-reference B-frame arrives early, just like we do for I- and P-frames. This ensures that reference B-frames are not dropped, and therefore, we don't cause video distortions. To identify the 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. One can also try identifying reference B-frames by parsing the NAL units of the H.264/AVC encoded stream.

We propose a different approach that doesn't require knowing precisely which B-frames are used as references. This approach is based on the observation that B-frames with a lower decode order are more likely to be referenced by other B-frames. Recall that whenever a frame depends on a future frame, which is the case with B-frames or bi-directional predicted frames, the future frame is encoded first and, therefore, has a lower decode timestamp. Frames only depend on frames with a lower decode order.

We found this heuristic to work well for GoP structures with a maximum of three B-frames between each I- or P-frame. We haven't tested GoP structures that use more than three B-frames since using more than three B-frames is not common in low-latency live streaming. From our experiments, we observed that when there were consecutive B-frames, the first two frames were always out of order, as we can see in the first group of B-frames in Figure 3.2. This indicates that the second B-frame is being used as a reference by the first B-frame, otherwise the two frames wouldn't be out of order. Furthermore, dropping only the second B-frame caused very noticeable video

(a) Original frames

(b) Frames decoded without the second B-frame

Figure 3.3: Video artifacts caused by dropping the second B-frame of a group of three consecutive B-frames

distortions, both for the first and third frames in the group, as shown in Figure 3.3. We conclude that in addition to the first B-frame, the third B-frame also depends on the second B-frame. On the other hand, dropping the second and third B-frames didn't cause any video distortions, and therefore, we assume that these frames are not used as references. With less than three consecutive B-frames, none was used as a reference in our experiments.

Since B-frames with a lower decode order are more likely to be used as references, we prioritize older B-frames over newer ones within a group of consecutive B-frames so that frames that are depended on are more likely to be delivered first. We will use the term B-frame group to denote a group of consecutive B-frames. Across B-frame groups, we use the same new over old policy by assigning a higher priority to newer groups. Note, however, that H.264/AVC allows P-frames to reference B-frames. As a result, video distortions will still occur whenever a B-frame that is referenced by a P-frame is dropped.

3.4 Skipping old GoPs

Degrading the stream quality by deprioritizing B-frames is one way to decrease latency when the network is congested, but it is not enough to handle large changes in the network bandwidth. 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 while on the move and temporarily passes through an area with poor 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 at the server. When the network recovers, rather than playing all the video that is queued, 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.

3.4.1 Snapping video forward

Our goal is to render new media immediately, such that playback 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 result in 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.

3.5 Combining both strategies

Ideally, we want the live streaming system to handle both small prolonged drops in the network bandwidth and short spikes effectively. While we have not yet confirmed our hypothesis, we believe that skipping old GoPs, while also deprioritizing B-frames,

would result in the most optimal QoE. In this section, we propose an approach that combines the two strategies.

We divide the broadcast into two MoQ tracks, a track for the base layer that consists of the I- and P-frames and a track for the B-frames, which form the enhancement layer, like we did in Section 3.3. However, rather than using a stream per track, we use a stream per GoP for the base layer like in Section 3.4. Each B-frame is transmitted in its own stream.

To prioritize the QUIC streams, the idea is to prioritize new over old media and frames from the base layer over frames from the enhancement layer. The latest GoP in the base layer is given the highest priority, followed by the B-frames from the latest GoP, such that they are transmitted next if there is sufficient network bandwidth. Old GoPs have lower priorities.

An important caveat is that this priority scheme requires many priority values, one for each GoP and each B-frame, and it is possible that there aren't enough to cover the entire live stream. quinn-rs[5], for example, which is the QUIC implementation used by moq-rs, uses 32 bits to represent the priority value, which might not be sufficient if the GoPs are short, and the frame rate of the video stream is high. One can address this issue, by wrapping the priority values around when the maximum value is exceeded. However, if any streams are queued when this occurs, then these streams will have a higher priority than the new streams. The client can close the old streams immediately upon receiving them, but the server will always transmit at least some bytes of the old streams and will keep transmitting them until it receives the `STOP_SENDING` QUIC frame. Furthermore, the new streams won't start to be transmitted until this process occurs for all old streams. So far, we haven't been able to identify a solution, and further work is needed to resolve this issue.

[5] `https://github.com/quinn-rs/quinn`

4 Evaluation

We now turn to the evaluation of the streaming protocols we presented, referred to in this chapter as deprioritize B-frames and skip old GoPs. We begin by describing the testbed and metrics used to measure the performance of our streaming protocols. Then, we present the results collected from multiple test runs of our approaches across different network profiles. We conclude with a qualitative evaluation, in which we discuss additional factors that should be accounted for when considering either approach.

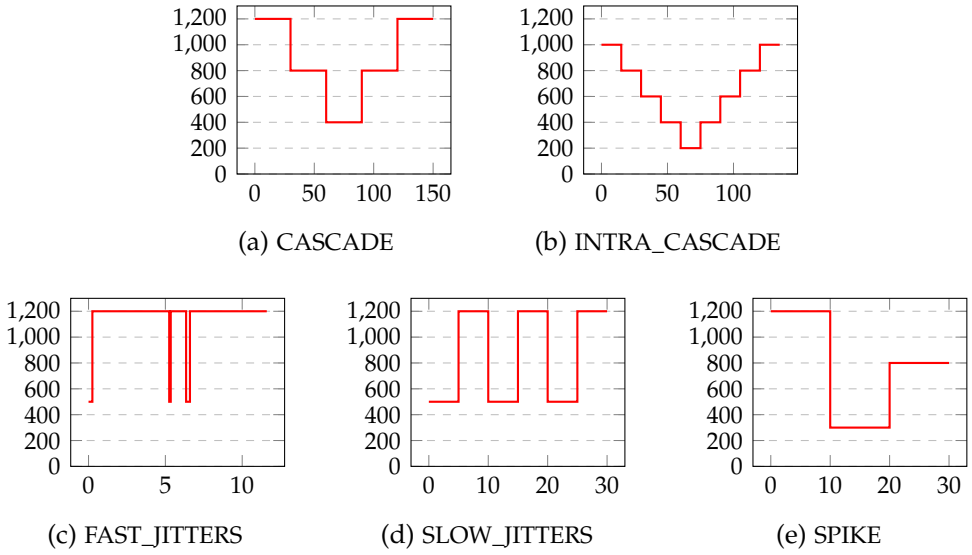
4.1 Testbed

Our testbed emulates network bandwidth limits during a live streaming session between the client and the origin server. We perform multiple test runs for each streaming protocol, in which we first launch the client, establish a connection to the server, and subscribe to the video stream. Once playback starts, we simulate various types of network patterns that are representative of real-world network conditions and collect relevant metrics. The network profiles used, which are taken from Twitch's 2020 Grand Challenge on low-latency live streaming [17], are shown in Figure 4.1.

We use dummynet[1] to emulate a link with limited bandwidth between the origin server and the client. Dummynet is a traffic shaping tool that simulates queue size and bandwidth limitations, delays, and packet loss by intercepting traffic between the transport protocol layers [18]. In order to simulate the bandwidth limits, we use a dummynet *pipe* and feed traffic from the client to the server and vice versa into the pipe using PF[2] (Packet Filter).

The client and origin server run on the same machine and communicate through the loopback interface. The origin server listens for incoming QUIC connections on port 443. We use PF rules that filter traffic on the localhost address from port 443 to any port, which corresponds to traffic between the server and the client, and forward it to the dummynet pipe. The dummynet pipe is configured with a bandwidth limit that varies according to the network profile throughout the test run. We don't configure any other

[1] `https://man.freebsd.org/cgi/man.cgi?dummynet`
[2] `https://www.openbsd.org/faq/pf/`



(a) CASCADE (b) INTRA_CASCADE

(c) FAST_JITTERS (d) SLOW_JITTERS (e) SPIKE

Figure 4.1: Bandwidth profiles - Bandwidth (Kbit/s) vs. Time (s)

options of the dummynet pipe explicitly so that their default values are used. Some options worth mentioning include the queue size, which is set to 50 slots or packets by default, and the delay, which is zero if not configured explicitly.

For the source video stream, we use the Big Buck Bunny[3] video and read the input file at the native frame rate using the FFmpeg option `-re` to simulate live streaming. The video frame rate is 24 frames per second. We also reencode the video stream with H.264/AVC using a custom bitrate and GoP structure and repackage it in MP4 fragments. We use the following FFmpeg options to achieve this:

- `-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. Note that the encoder can use a lower number of B-frames between two reference frames.

[3] `https://peach.blender.org/`

- `-f mp4 -movflags cmaf+frag_every_frame`: use CMAF compatible fragmented MP4 as the output format and package each frame in a separate fragment.

In both the deprioritize B-frames, and skip old GoPs approaches, we use a buffer size of 100 ms for the jitter buffer. Furthermore, in the deprioritize B-frames approach, the reorder buffer has a size of 100 ms, leading to a total buffer size of 200 ms. It is also worth mentioning that the server uses the BBR congestion control algorithm.

4.2 Metrics

We measure the latency of the live stream to evaluate the performance of our streaming protocols in poor network conditions. In a real-world live streaming system, many components in the media streaming pipeline contribute to the end-to-end latency [8]. In our measurements, we only take into account the latency added by the delivery and consumption phases since our work focuses on the media distribution phase of the pipeline. We measure latency as the delay between the frame reaching the server and being rendered in the client.

We measure the latency for each frame as follows. 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 when the frame is rendered as the difference between the availability time and the time at which the frame was rendered. Note that both the server and client use the same clock, since they run on the same machine, and therefore clock drift is not an issue.

We would like to note however, that latency is not a complete metric by itself. Other factors, such as rebuffering events and stream quality, are also important in determining the viewer's QoE. An interesting area for future work is to use a weighted combination of these factors, such as the QoE model proposed by Yin et al. in [19] to evaluate our approaches.

4.3 Measurements

We now present our results. First, we analyze the deprioritize B-frames approach, using measurements from two sample runs for the profiles 500 Kbit/s and INTRA-CASCADE. Next, we examine the skip old GoPs approach, reviewing measurements from a sample run. Finally, we show the average latency of each approach across multiple test runs for all profiles.

Deprioritizing B-frames results in lower latencies when the network bandwidth is slightly below the bitrate of the video stream. Figure 4.2 (a) shows the latency of each

frame over time for sample runs of baseline and deprioritize B-frames. The bandwidth is limited to 500 Kbit/s, which is 100 Kbit/s below the average bitrate of the encoded video stream. At the start of the test run, the latency for both versions is similar. However, around 0:40, they begin to diverge, with deprioritize B-frames showing a slower rise in latency compared to baseline. The gap between the two versions increases steadily, reaching around 5 seconds at the 1:30 mark.

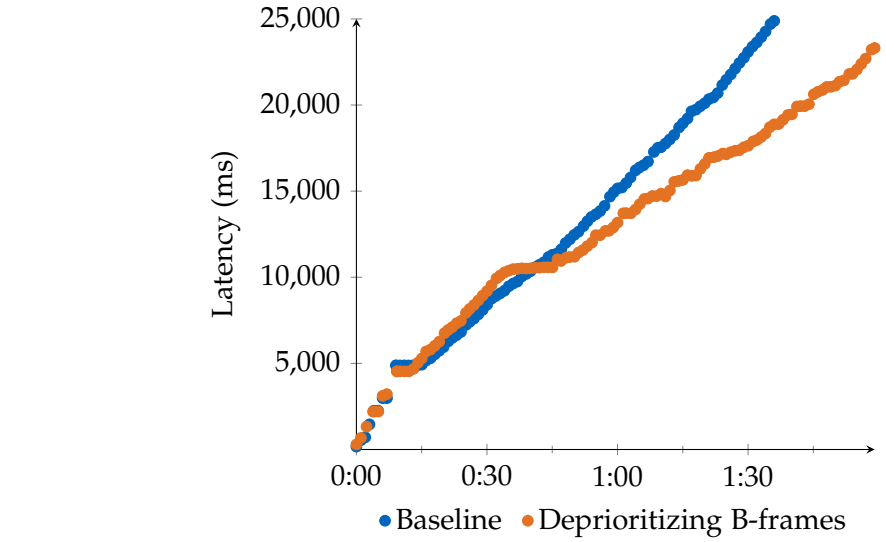
Taking a closer look at the frames delivered to the client throughout the test run provides an explanation. Figure 4.2 (b) shows the bitrate of frames received by the client, categorized by frame type. About 40 seconds into the test run, the client in deprioritize B-frames stops receiving B-frames, which indicates that the server has stopped sending them. Because the server is no longer transmitting any B-frames, more bandwidth is left for the base layer. For instance, between 1:25 and 1:35, baseline receives approximately 200 Kbit/s of B-frames and 200 Kbit/s of P-frames, while deprioritize B-frames receives 0 Kbit/s of B-frames and 300 Kbit/s of P-frames, 100 Kbit/s more than baseline. By deprioritizing B-frames, frames from the base layer are delivered to the client sooner, causing latency to increase at a slower rate.

However, deprioritize B-frames shows little improvement over baseline when the network bandwidth drops significantly below the stream's bitrate. This is because B-frames account for only a small fraction of the stream's bitrate compared to I-, and P-frames. Figure 4.3 (a) shows the latency over time for sample runs of baseline and deprioritize B-frames for the profile INTRA-CASCADE. Latencies for both versions are similar between 0:00 to 1:30. At around 1:30, when the bandwidth increases from 400 Kbit/s to 600 Kbit/s, the latency for deprioritize B-frames stops rising, while for baseline, it continues growing until 1:45.

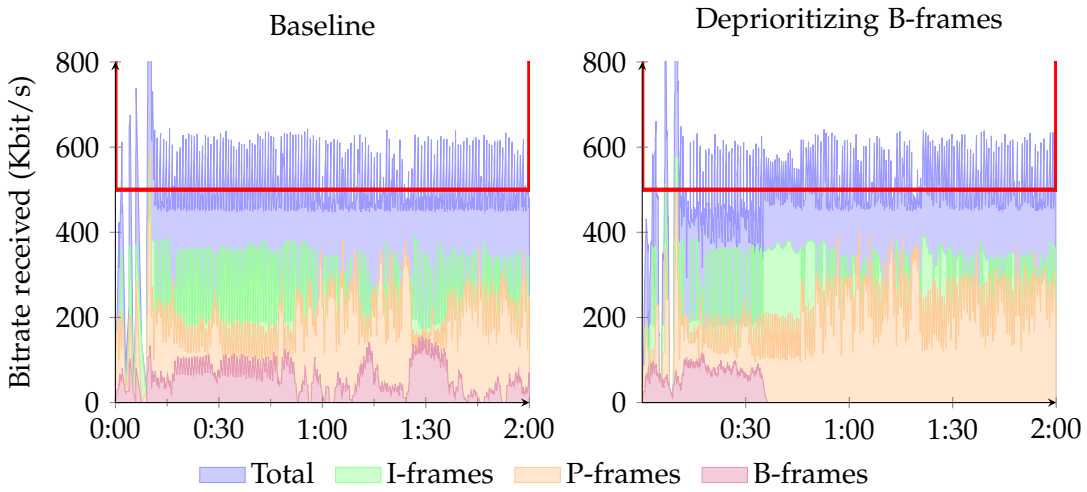
This is once again explained by the bitrate of frames received by the client, as shown in Figure 4.3 (b). This time, the client in deprioritize B-frames stops receiving B-frames around the 1:07 mark. From this point forward, the server only transmits frames from the base layer to the client. Latency stops increasing for deprioritize B-frames at 1:30, when the network bandwidth increases from 400 Kbit/s to 600 Kbit/s, as 600 Kbit/s is above the bitrate of the base layer. However, 600 Kbit/s remains below the total bitrate of the stream. Therefore, latency continues to increase for baseline until the bandwidth jumps to 800 Kbit/s at 1:45.

Most importantly, once the latency has increased, it does not decrease for either approach, even when the network bandwidth exceeds the base layer's bitrate for deprioritize B-frames and the total bitrate for baseline at 1:30 and 1:45, respectively. This is because, on the client side, both baseline and deprioritize B-frames play older video segments before new ones and do not skip any video segments.

Thus, the bottom line is that degrading the stream quality to lower the bitrate of the live stream is not enough to ensure low latency. It is necessary to skip old media when

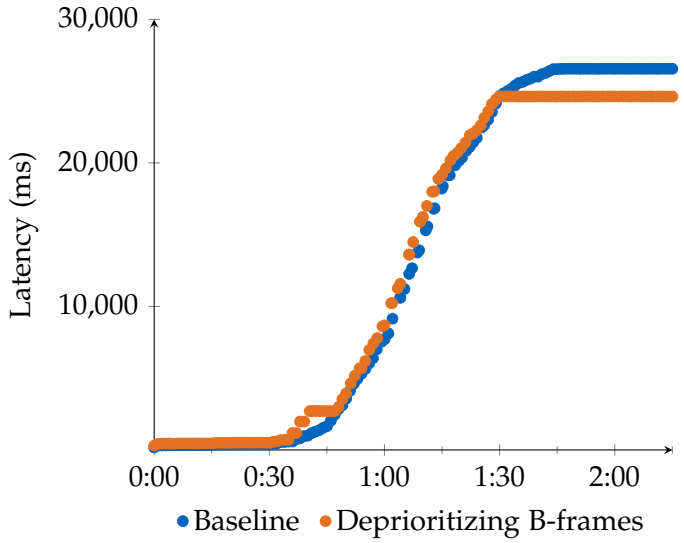


(a) Latency

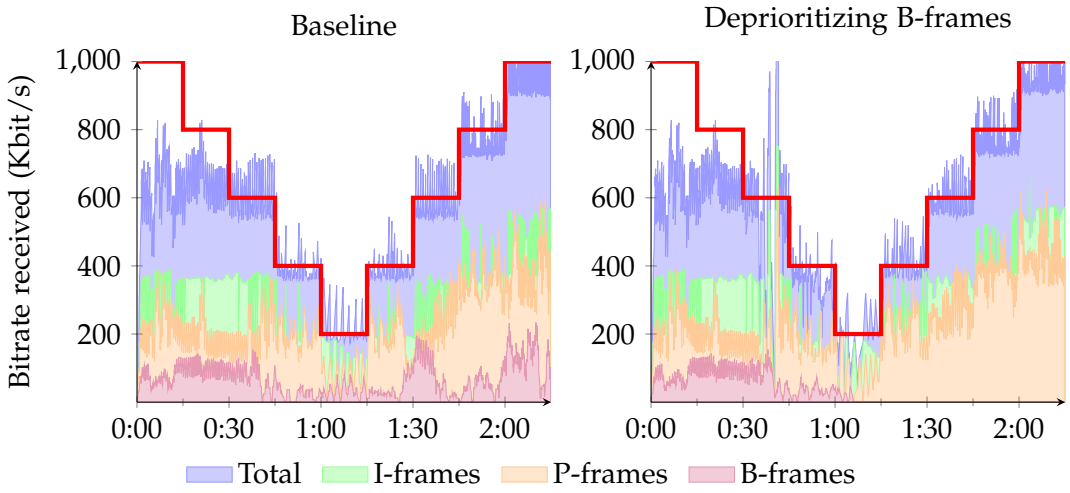


(b) Bitrate received

Figure 4.2: Sample run of deprioritize B-frames for profile 500 Kbit/s



(a) Latency



(b) Bitrate received

Figure 4.3: Sample run of deprioritize B-frames for profile INTRA-CASCADE

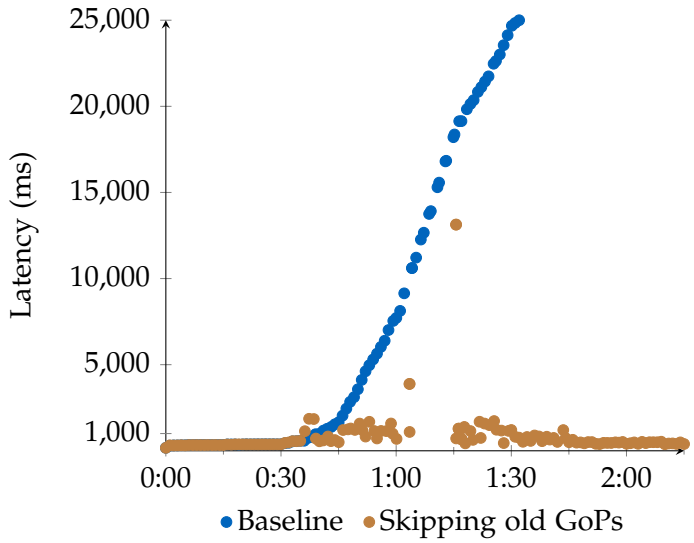
the network bandwidth drops substantially.

Skipping old GoPs significantly reduces latency across different network conditions. Figure 4.4 (a) shows the latency over time for sample runs of baseline and skip old GoPs for the profile INTRA-CASCADE. Unlike baseline, where latency increases consistently after 0:30, skip old GoPs achieves an average latency of around 1.5 seconds between 0:30 and 1:30. Moreover, the latency returns to its minimum at 1:45.

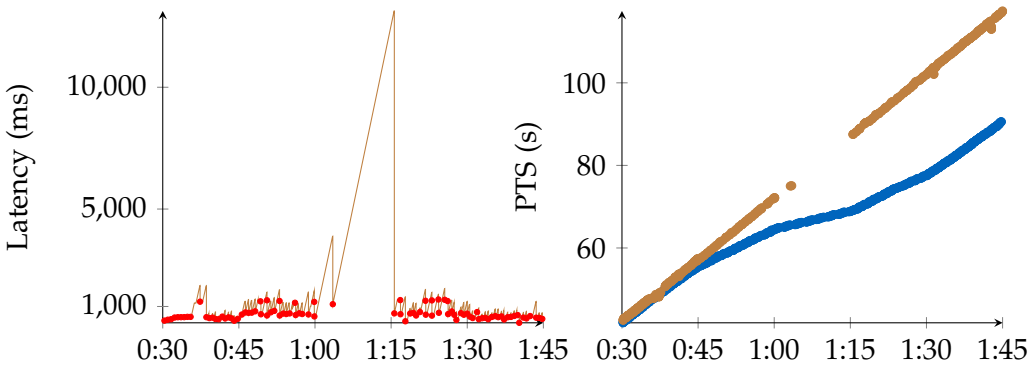
In general, the latency increases throughout each GoP and decreases when a new GoP begins. When a new GoP starts, which happens with each I-frame, the latency is effectively reset. This is depicted in Figure 4.4 (b), which shows that I-frames correspond to the lowest points in the latency graph. As a result, the maximum latency is bound by the GoP size. Smaller GoPs result in lower latencies.

Skip old GoPs achieves this by prioritizing new GoPs over older ones, ensuring they are transmitted as soon as the encoder produces them. Figure 4.4 (c) shows the presentation timestamp (PTS), the time a frame should be rendered relative to the stream's start, plotted against the test time. For skip old GoPs, the PTS of received frames increases, in general, at the same rate as the test time, meaning that the average latency remains more or less constant. Note that the latency grows between I-frames as previously mentioned, but these small increases are not visible on the graph. In contrast, the rate at which the PTS increases for baseline begins to slow down 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 too low for the server to transmit the I-frame of the latest GoP before the I-frame of the next GoP is available. The average size of I-frames throughout the sample run was approximately 173.76 Kbit. At 200 Kbit/s, the server would need about $0,87$ seconds to transmit a single I-frame. However, given that the frame rate of the stream is 24 frames per second and the GoP size is 15 frames, the encoder produces a new I-frame every 0.625 seconds. As a result, the server begins transmitting the new I-frame, which has a higher priority, before finishing transmitting the current one. Hence, no frames are fully transmitted during this period. This is a flaw of the prioritization scheme used by skip old GoPs, and further work is necessary to address this issue.



(a) Latency



(b) Latency of I-frames (red marks)

(c) Presentation timestamps of frames

Figure 4.4: Sample run of skip old GoPs for profile INTRA-CASCADE

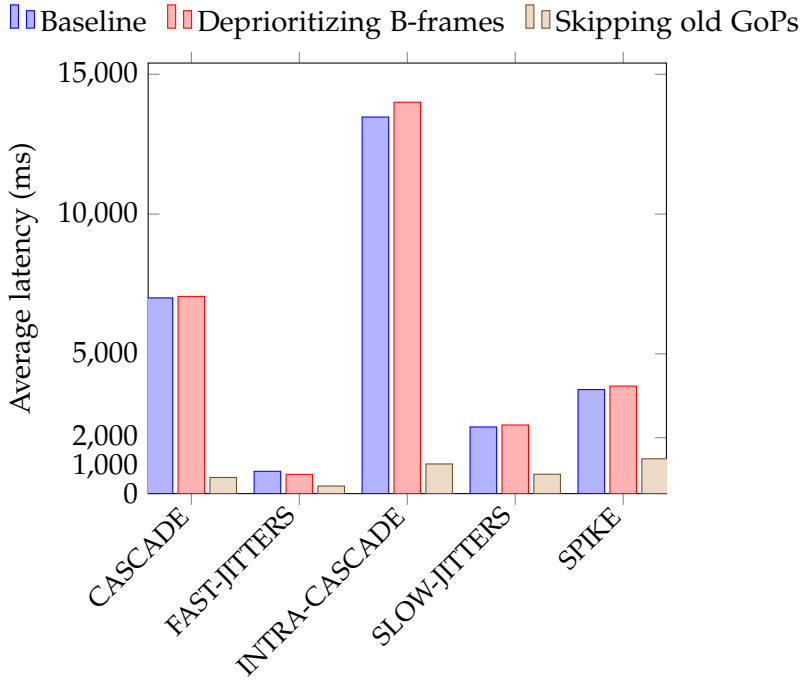


Figure 4.5: Average latency for all profiles

We now present the results from multiple test runs of our approaches for the network profiles previously shown in Figure 4.1. Figure 4.5 shows the average latency for each of our streaming protocols. Skip old GoPs is the clear winner, consistently achieving lower latencies across all profiles. The difference is especially noticeable in the CASCADE and INTRA-CASCADE profiles. In contrast, deprioritize B-frames performs similarly to baseline for all profiles.

4.4 Qualitative evaluation

Having presented our measurements, we now discuss additional factors that influence the viability of each approach.

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 unfavorable network conditions

is proportional to the size of the enhancement layer and therefore, the number of B-frames. However, B-frames add latency because they reference future frames, so the encoder cannot produce them until those frames are available.

- The server must parse the encoded stream to extract the frame 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 noticeable when the network bandwidth is extremely low, and these jumps are frequent.

5 Related work

Some QUIC-based media streaming protocols have been developed prior to MoQ. Warp [20], 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 [21], developed by Facebook, another media streaming protocol using QUIC was designed primarily for media ingestion. In addition 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 [22], 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 [24, 23]. 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. [25] 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 [26], a Rust implementation of MOQT, origin server, relay, and other components, and moq-js [27] 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 can be found in [28].

6 Conclusion

In this thesis, we studied the design and implementation of a live streaming system using MoQ. We first described an implementation of a prototype live streaming system consisting of an origin server and a client. We then presented two streaming protocols that use stream prioritization to decrease latency. In our first approach, the idea is to use the B-frames as an enhancement layer that we can drop to save bandwidth when the network is congested. In order to achieve this, we transmit B-frames in their own streams and assign them a low priority while transmitting the I- and P-frames in a single stream with a high priority. We then describe a second approach, in which we skip old video by prioritizing new GoPs over old ones. Similar to Warp, we map each GoP to a separate stream and assign newer GoPs a higher priority. We evaluated the performance of these two streaming protocols by measuring the latency for a variety of network profiles and presented our results.

Deprioritizing B-frames results in slightly lower latencies than transmitting all frames in the same stream with the same priority, but the effect is not significant. In cases where the network bandwidth slightly drops below the stream's bitrate for an extended period of time, the extra bandwidth allocated to the base layer can help slow down the rate at which latency increases and reduce the number of buffering events. However, this is an exception, as in the majority of most network bandwidth patterns, deprioritizing B-frames results in a similar performance to transmitting B-frames with the same priority as I- and P-frames. This is because B-frames make up only a small amount of the total bitrate. Finding other ways to divide the video stream into base and enhancement layers, such that the enhancement layer constitutes a larger portion of the total bitrate, can enable new streaming protocols that can trade off stream quality for latency more aggressively.

We also conclude that an approach that doesn't prioritize across the temporal dimension cannot be optimal. To prevent the latency from increasing, the stream's bitrate must be below the network bandwidth, and one can only decrease the stream's bitrate by degrading the stream quality until a certain point. Even if one could drastically reduce the stream's bitrate while maintaining a watchable quality, clearly, if the network throughput drops to zero due to a network fault, the stream will inevitably lag behind, causing the old video to queue. When this occurs, the server must skip old video segments that are queued so that the client resumes playback at the live edge in order

to ensure the minimum possible latency. To do this, we can skip old GoPs, which we've shown consistently achieves lower latencies.

We can design and implement live streaming systems using Media over QUIC that are able to respond to congestion in a variety of new and improved ways by leveraging QUIC. Using QUIC's features, such as concurrent independent streams and stream prioritization, it's possible to achieve low latency even under unfavorable network conditions by trading off stream quality and skipping parts of the live stream.

In future work, we plan to explore alternative ways of dividing the stream into base and enhancement layers. One promising approach is to use Scalable Video Coding (SVC), which encodes the video stream into a hierarchy of layers that can be dropped to reduce quality, making it a natural fit. Another option is to use Reference Frame Invalidation (RFI) to force P-frames to reference only I-frames, allowing P-frames to be dropped independently without affecting the decodability of other P-frames. This would allow for strategies like sending every other P-frame in the enhancement layer to achieve temporal scalability. Additionally, an interesting direction for future work is to send B-frames unreliably using QUIC's datagram feature.

Abbreviations

ABR Adaptive Bitrate

CDN Content Delivery Network

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

IETF Internet Engineering Task Force

MoQ Media over QUIC

MOQT Media over QUIC Transport

NAL Network Abstraction Layer

OTDR On-Time-Display Ratio

QoE Quality of Experience

RFI Reference Frame Invalidation

RTT Round-Trip Time

SVC Scalable Video Coding

UDP User Datagram Protocol

VOD Video on Demand

List of Figures

Bibliography

[1] Cisco. *Cisco Visual Networking Index: Forecast and Trends, 2017–2022*. 2018.

[2] 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.

[3] 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.

[4] MPEG-4 Part 14. *MPEG-4: MP4 File Format*.

[5] ISO/IEC. *ISO/IEC 23009-1:2022 Information Technology — Dynamic Adaptive Streaming over HTTP (DASH) — Part 1: Media Presentation Description and Segment Formats*.

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

[7] ISO/IEC. *ISO/IEC 23000-19:2024 Information Technology — Multimedia Application Format (MPEG-A) — Part 19: Common Media Application Format (CMAF) for Segmented Media*.

[8] 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.

[9] 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.

[10] J. Iyengar and M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Request for Comments RFC 9000. Internet Engineering Task Force, May 2021. 151 pp. DOI: 10.17487/RFC9000.

[11] S. Arisu and A. C. Begen. "Quickly Starting Media Streams Using QUIC." In: *Proceedings of the 23rd Packet Video Workshop*. PV '18. New York, NY, USA: Association for Computing Machinery, June 12, 2018, pp. 1–6. ISBN: 978-1-4503-5773-9. DOI: 10.1145/3210424.3210426.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] Twitch. *Grand Challenge on Adaptation Algorithms for Near-Second Latency*. URL: https://2020.acmmmsys.org/lll_challenge.php (visited on 09/12/2024).

[18] L. Rizzo. "Dummynet: A Simple Approach to the Evaluation of Network Protocols." In: *SIGCOMM Comput. Commun. Rev.* 27.1 (Jan. 1, 1997), pp. 31–41. ISSN: 0146-4833. DOI: 10.1145/251007.251012.

[19] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli. "A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP." In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. New York, NY, USA: Association for Computing Machinery, Aug. 17, 2015, pp. 325–338. ISBN: 978-1-4503-3542-3. DOI: 10.1145/2785956.2787486.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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." In: *IBC*.

[25] 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.

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

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

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