Technische Universität Berlin

Open Distributed Systems
Fakultät IV
Einsteinufer 25
10587 Berlin
https://www.tu.berlin/ods

Bachelor's Thesis

Design and Implementation of Media Streaming using WebTransport and QUIC

Philip Nys

Matriculation Number: 397914
18.01.2024

Supervised by
Prof. Dr. Manfred Hauswirth
Prof. Dr. Volker Markl

Assistant Supervisor
Stefan Pham

Hereby I declare that I wrote this thesis myself with the help of no more than the mentioned literature and auxiliary means.

Berlin, 18.01.2024

. . . . . .

Philip Nys

Abstract

Media streaming is becoming more and more popular and is taking up a vast portion of the entire internet traffic. This increasing popularity makes it crucial to continue research to making media streaming more efficient. One avenue of making media streaming more efficient is improving the underlying transport protocols. This thesis analyzes a favorable contender to become the next transport protocol for media streaming. This protocol is called QUIC, developed by Google and it was designed with low-latency, security and scalability in mind. This thesis focuses on media streaming using QUIC, specifically WebTransport protocol, compared to traditional HTTP methods in a web browser environment. A WebTransport and HTTP server and a website client have been implemented, which communicate between each other to stream Video-on-Demand as well as live low-latency media using the aforementioned protocols.

Zusammenfassung

Das Schauen von Online-Media wird immer beliebter und stellt damit ein großes Problem dar, denn ein Großteil des gesamten Internetverkehrs wird davon in Anspruch genommen. Diese zunehmende Beliebtheit erweckt das Streben die Forschung im Bereich des Media Streamings in Richtung von verbesserter Effizienz zu lenken. Eine Möglichkeit dafür ist die Verbesserung der verwendeten Transportprotokolle. Diese Arbeit befasst sich mit dem vielversprechenden Protokoll QUIC, welches von Google entwickelt wurde. Dieses Protokoll wurde speziell für Anwendungen entwickelt, welche Anspruch auf geringe Latenzen haben und dennoch gesichert sein müssen und im großen Maßstab funktionieren müssen. Diese Arbeit beschäftigt sich mit dem Streaming von Media mit der Verwendung von QUIC, im speziellen mit dem WebTransport Protokoll, und im Vergleich zu herkömmlichen HTTP-Methoden auf einer Internetseite. Für diese Arbeit wurden ein WebTransport und HTTP Server aufgesetzt und eine entsprechende Internetseite erstellt, welche untereinander kommunizieren, um Video-on-Demand und Live Video mit den genannten Protokollen zu streamen.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Media streaming has become a favorite past time activity for many people all over the world. This steadily increasing interest in media streaming has lead to media streaming taking up about 58% of all internet traffic in 2018 [1] and showing a trend upwards. Additionally, the range of media streaming types has increased exponentially, from basic Video-on-Demand (VoD) and live broadcast to very large commercial streaming services and sport events, but also increased popularity of personal streaming on social media or in the field of video game streaming.

This personal streaming sector specifically has a high demand of having the lowest possible latency between the producer and their viewers. Furthermore, the recent Covid-19 pandemic has opened the flood gate of video conferencing and it requires ultra low-latency to allow natural interactions between participants.

For the past 10 years HTTP adaptive streaming (HAS) in the forms of DASH.js and HLS.js have been doing an adequate job of leveraging HTTP and the underlying Transmission Control Protocol (TCP) to provide consumers with content to consume. However, TCP is still suffering from glaring problems that hinder it from staying relevant for the current scope of media streaming demand and its increasing requirements.

The emerging new transport protocol QUIC offers solutions for the issues of TCP and in theory is even capable of surpassing TCP's strengths.

1.2 Objective

As mentioned before TCP is suffering from issues holding it back from providing the consumers with the best possible viewing experience. It uses an outdated opening handshake method which is slower than it needs to be. In addition TCP doesn't support any encryption out of the box, which not only adds security concerns but also adds another layer to the opening handshake when adding encryption, further slowing the handshake down. Finally, it suffers from Head-of-Line-Block (HoL-Blocking) which describes an occurrence during congestion control when a specific data packet is stalled on the way leading to all following packets being stalled as well.

The QUIC protocol addresses many of these issues by streamlining the opening handshake having security in mind from the start as well as having vastly improved congestion algorithms completely eliminating HoL-Blocking.

This thesis describes an approach of combining the QUIC protocol and existing media methodologies to implement a Client-Server-Model for streaming and playing back media

in a web browser. The aim of this work is to prove that QUIC is superior to TCP in the context of media streaming.

1.3 Scope

During my work as student research assistant at FOKUS Fraunhofer Institute I have implemented a working demo of a WebTransport server using webtransport-go [2] in the Go programming language and an accompanying HTTP/1.1 server using express.js [3] in the TypeScript language. These servers are each hosting local Video-on-Demand (VoD) MPEG-DASH video segments and are also capable of receiving live low-latency segments from `FFmpeg` [4] and distribute them to clients. Furthermore, I implemented a client application for web browsers leveraging an HTMLVideoElement and the MediaSourceExtensions API (MSE) [5], similar to how Dash.js does it, to play back video in the browser.

In this thesis I will implement the preexisting servers using the Rust programming language and adjust the client implementation accordingly if required and compare their performance.

1.4 Outline

Finally, I will present the structure of this thesis by outlining the coming chapters.

Chapter 2 will go over the current State of the Arts in regards to HTTP streaming using HTTPS (TCP + TLS), followed by an overview of the QUIC protocol. In addition I will give you a detailed look into the WebTransport API, how to use it and what it can do. Finally coming to a close of this chapter by giving a callback to RTP and RTMP, transport protocols that were used before HTTP that have many similarities to QUIC and also incorporating concurrent approaches for media streaming using QUIC.

Chapter 3 will outline the requirements I had to full fill before I was able to start with the implementation.

Chapter 4 is about the actual design of the Client-Server-Model I will be implementing in the scope of this thesis, going into detail of the respective servers, the client and the overall architecture of this model and how they are going to operate.

Chapter 5 describes the actual implementation part of the design, the problems I came across, how I overcame them and the changes I had to make in order to realize the design.

Chapter 6 includes the evaluation of my implementation, how I verified the efficiency of the QUIC protocol and the benchmark I designed and ran.

Chapter 7 concludes this thesis, looking back on what I have done, what problems

I encountered and giving an outlook on how to expand this work and future work that can be conducted based on this thesis.

2 State of the Art

2.1 HTTP Streaming

The current protocols HTTP/1.1 and HTTP/2 have been well established as the defacto media streaming protocols for almost the past 15 years with the release of Apple's HTTP Live Streaming (HLS) in 2009 [6, 7] as well as Dynamic Adaptive Streaming over HTTP (Dash) in 2012 [8, 9].

For the remainder of this thesis Dash will be used for reference and comparison in examples.

These two HTTP versions both use TCP as underlying transport protocol, which is a reliable protocol. In this context reliable refers to guaranteeing that all data that is sent using TCP will not only arrive at the destination without errors but also in the intended order. This is achieved through packet loss and error detection as well as correction by retransmitting affected packets [10].

Media streams are encoded into multiple tracks of varying qualities, video resolution, encoding bit rate, frames per second, audio bit rate, audio sampling rate and more. These tracks are again split up into segments, chunks that are usually a few seconds long and are able to be played back independently. These segments make it possible to switch between the tracks of varying qualities depending on the current situation, for example if the player detects a drop in the network throughput it is able to drop the quality accordingly to put less strain on the network and allow the video to playback without stalling. This is what is known as Adaptive Bitrate (ABR) streaming where an ABR algorithm is used to determine which segments get requested.

All this is encoded in an XML-style Manifest, the Media Presentation Description (MPD), which the player parses and uses to request the correct segments, using HTTP GET requests in a client controlled pull-based scenario [11].

2.1.1 TCP + TLS

To reiterate, HTTP streaming consists of the client periodically requesting each and every segment and in the meantime the client needs to analyze the current condition of the video stream and network conditions to request the right segments to prevent the playback from stalling by playing a lower quality that it is able to, all of which would lower the quality of experience for the viewer [11].

All this may create a few complications. On the one hand there is the potential that low-end devices may not be able to keep up with the computational load. On the other hand having to specifically request every single video segment creates a lot of overhead that puts unnecessary strain on the network. Furthermore, before any video can be

played on the client they have to not only perform a opening TCP handshake but also request the manifest file, the initial segment and the first actual segment [11]. This leads to HTTP offering a fairly high latency and making it less than ideal for low-latency applications.

Additionally, TCP on its own does not provide any form of encryption and thus is required to be coupled with an additional protocol step to provide security in form of HTTPS. The Transport Layer Security (TLS) protocol has been the standard procedure to enable HTTP to become HTTPS. In order for the connection to be encrypted an additional TLS handshake has to be performed [12], further increasing the latency. HTTPS has to make the trade-off of having security by increasing latency. Unfortunately we have to go with the assumption of always having to use HTTPS since as of the writing of this thesis roughly 85% of all websites use HTTPS [13].

A simplified example on how the process of requesting a video can be seen in fig. 2.1.

2.2 QUIC

Development of QUIC started started at Google in form of the protocol SPDY (speedy) [14] in 2012, which was a TCP based transport protocol that tried to combat TCP's weaknesses and it already showed some similarities to today's QUIC. It had TLS directly integrated for a fast connection establishment, additionally it improved the multiplexing of multiple transmissions, allowed to prioritize specific requests to prevent HoL-Blocking and it allowed the server to initiate transmissions without the client having to requests these. Ultimately the development of SPDY lead to its features to be integrated into HTTP/2 in 2015 [15].

Shortly after the release of HTTP/2 Google began the development of QUIC and submitted the first draft to standardization in 2016 with the final release being in 2021 [16]. Again this lead to the release of the next version of HTTP in form of HTTP/3 [17].

2.2.1 Overview

QUIC is transport protocol based on the User Datagram Protocol (UDP) but heavily expands upon it. UDP is by its nature an unreliable protocol, in contrast to TCP which is reliable, meaning TCP guarantees data to arrive without error and in order, while UDP does not establish a persistent connection between client and server and instead just sends data with good faith that it arrives [18].

Since there is no additional handshaking or encryption step required, this gives a fantastic latency prospect. However, not having a reliable connection can lead to data arriving incorrectly, in the wrong order, multiple times or not at all [18].

This is where QUIC has expanded UDP by first of all integrating a opening QUIC handshake which not only negotiates the transport parameters but also cryptographic parameters. This way QUIC has already cut down the initial round-trip-times (RRT) from 2 (TCP + TLS) down to a single RRT. There is even the option to allow the client to immediately send data in a 0-RTT fashion if this has been previously configured [18].

An abstracted model on how data in form of a video stream may be sent, can be seen in fig. 2.1.

A single QUIC connection is capable of communicating with a multitude of clients simultaneously on the same UDP port [18].

QUIC offers two methods to send data between client and server, also allowing the server to send data to the client without requiring specific requests for them [18].

Datagrams

The first method are datagrams which are essentially normal UDP payloads with the added benefit of being encrypted and congestion controlled.

Just like UDP these datagrams are still unreliable [18].

Streams

The other method are so called QUIC streams. Streams are abstract transport tunnels, allowing server and client to reliably send data between each other. They can be initiated by both client and server and come in two varieties

- Unidirectional: only the initiator is able to send data and the recipient can only receive data
- Bidirectional: both participants can send data as well as receive data on the same stream

Sending data on these streams are reliable, meaning the data will arrive correctly and in order. This is achieved by concurrently opening streams in the background which will monitor the correctness of the payloads and in case of errors or packet loss, they will retransmit the affected packets [18].

2.3 WebTransport

Since QUIC is a transport protocol an interface on the application layer is required to allow developers to make use of QUIC. Since the goal of this thesis is to implement a browser-based client WebTransport [19] is the ideal API for this use case.

WebTransport allows low-level manipulation of the QUIC transport protocol in a high-level application.

WebTransport is an extension of the basic QUIC protocol. While it offers the same functionality of the underlying QUIC protocol just like HTTP/3, it also provides a fall-back to HTTP/2 when UDP is not available [20]. If that is the case HTTP/2 will be used in place of QUIC to still provide the same functionality.

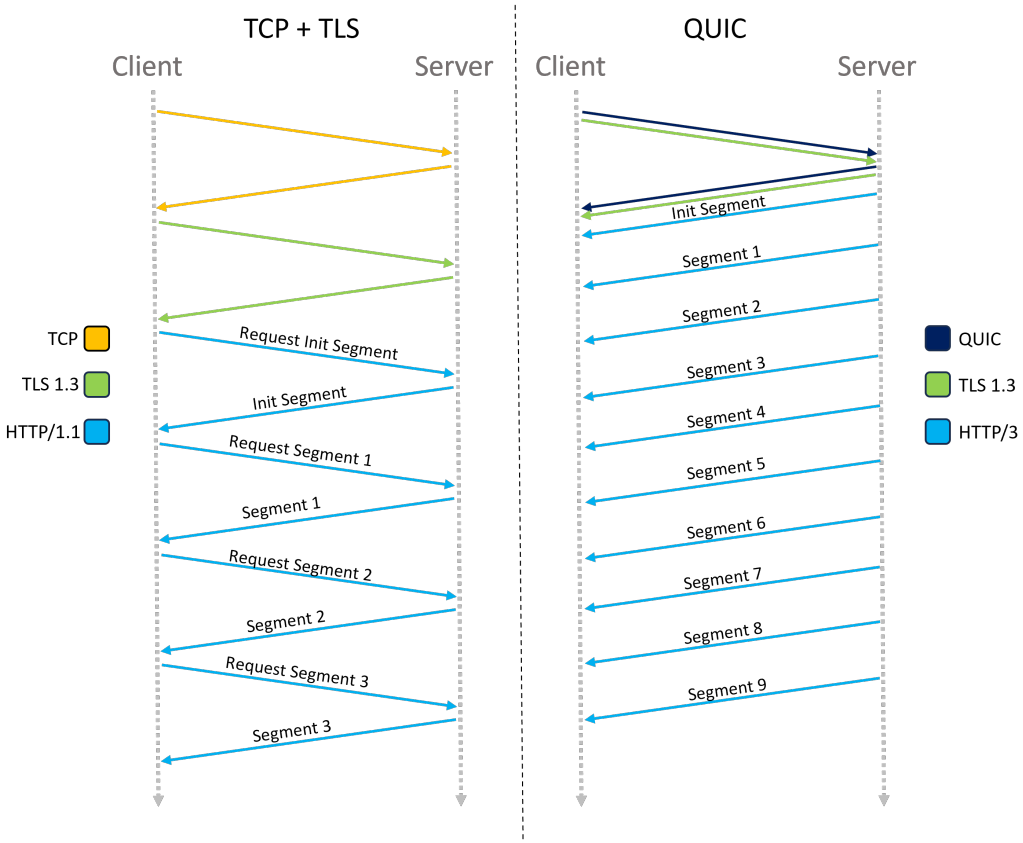

Figure 2.1: Sending Media Segments using TCP + TLS and QUIC

2.3.1 API Overview

As of the writing of this document all major browsers (Chromium-based and Firefox) support WebTransport, apart from Safari [21].

WebTransport is a highly asynchronous API and all of the following code snippets are supposed to be encapsulated in asynchronous environments. Additionally, please note to condense the examples as much as possible I will not repeatedly reinitialize repetitive elements.

Establishing a Connection

To establish a connection to a WebTransport-capable server all one needs to do is invoke the `WebTransport` Constructor (line 5 in listing 2.1) with the URL of the server (line 4 in listing 2.1). The URL must use the HTTPS protocol and include the hostname and port, it may point to a specific address endpoint and include query parameters, see listing 2.1.

```
1  // example URL pointing to localhost on port 5000
2  // including the "/media" endpoint and
3  // query key "id" and value "hello"
4  const url = "https://localhost:5000/media?id=hello";
5  const quic = new WebTransport(url);
6
7  // Optional: add handler on Connection closing
8  quic.closed
9      .then(() => console.log("WebTransport connection closed gracefully.")
           )
10     .catch((err) => console.error("WebTransport closed with Error: ", err
           ));
11
12 // wait for the connection to be ready
13 await quic.ready;
14 // Connection is ready to use from here on
```

Listing 2.1: Opening a WebTransport Connection

Additionally, it is possible to add callbacks for the events that the connection is closed either gracefully or with error (line 8-10 in listing 2.1).

Payloads

Important to note is that WebTransport streams are an extension of the existing Streams API [22]. The important part of that is that all the payloads one wants to send are required to be byte-encoded and similarly the data one will receive is also byte-encoded. In both cases the bytes are encoded using UTF-8 encoding.

To encode data one can use `TextEncoder` API as shown in listing 2.2 and analog the `TextDecoder` API can be used to decode data listing 2.3.

```
1  // Example data
2  const obj = {
3      method: "GET",
4      data: "Hello World",
```

```
};

// Create the TextEncoder
const encoder = new TextEncoder();

// Encode the Object
// Note: the encoder requires the data to be of type "String" => JSON.stringify()
const data = encoder.encode(JSON.stringify(obj));
// data is now ready to be sent
```

Listing 2.2: Encoding Data

```
const data = /* Data from stream */;

// Create the TextDecoder, specifying UTF-8 encoding
const decoder = new TextDecoder("utf-8");

// Decode data
const payload = decoder.decode(data);
// payload is now the data received as String, process accordingly
```

Listing 2.3: Decoding Data

Reading a Stream

There is a no easy to use catch all read function for reading from a stream. It is required to manually repeat the read operations in order to read data chunks from the stream. To receive all data from a stream one needs to keep reading from it until the stream is closed.

The read operation on a stream `stream.read()` is an asynchronous operation and returns the following object

```
{
    done: boolean,
    value: Uint8Array | Bidirectional Stream | ReadableStream | undefined
}
```

Listing 2.4: Object returned by Stream Read Operation

The `done` field of this object indicates wether or not the stream is still open, `false` equals as the stream still being open and that potentially there is still data supposed to arrive, logically `true` signifies that the stream is closed and no more data is coming. The data of the `value` field depends on the context of the `Reader`. If the `Reader` is a `ReadableStream` (e.g. an Unidirectional Stream initiated by the opposing peer or the readable half of a Bidirectional Stream) then the `value` will be raw bytes wrapped up in a `Uint8Array`, when the `Reader` is listening for incoming Bidirectional Streams then the `value` will be Bidirectional Stream and similarly when the `Reader` is listening for incoming Unidirectional Stream the `value` will be a `ReadableStream`.

Please note, when `done` is `true` then `value` is `undefined` and when `value` is not `undefined` then `done` is always `false`. In short, you cannot have `done` as `true` and any data in `value` at the same time.

The easiest way to read a stream in its entirety is to use an endless loop and breaking it once the `done` field is `true`, shown in listing 2.5.

```
1 const reader = /* ... */;
2 while ( true ) {
3     // read and destructure value and done for easier use
4     const { value, done } = await reader.read();
5     // stream is closed, value is undefined, stop loop
6     if ( done ) break;
7
8     // process value
9 }
```

Listing 2.5: Reading a Stream

Datagrams

As previously mentioned, QUIC is based on UDP and WebTransport allows to send data outside of the Stream API using basic datagrams. These datagrams are basically UDP payloads, but with the added benefit of being encrypted and congestion-controlled [23], while still being unreliable.

To send and receive data using an open connection one can simply get the `Writer` and `Reader` directly from the WebTransport object, as shown in listing 2.6 on line 2 and 7 respectively.

```
1  // Send data using Datagrams
2  const writer = quic.datagrams.writable.getWriter();
3  const data = encoder.encode("Hello World");
4  writer.write(data);
5
6  // Receive data using Datagrams
7  const reader = quic.datagrams.readable.getReader();
8  while ( true ) {
9      const { value, done } = await reader.read();
10     if ( done ) break;
11
12     const payload = decoder.decode(value);
13     // process payload
14 }
```

Listing 2.6: Using Datagrams

Unidirectional Streams

Unidirectional streams allow the opener to send data to the connected recipient in a reliable fashion. A stream can be create using the `createUnidirectionalStream()` method of the WebTransport object and here it behaves very similarly to the previously shown

datagrams interface (as shown in listing 2.7). Unidirectional Stream can be initiated by both the client and the server. They are essentially `WritableStreams` from the Streams API [22].

```
// Create a unidirectional Stream
const stream = await quic.createUnidirectionalStream();

// get the Stream Writer
const writer = stream.writable.getWriter();

// encode payload
const data = encoder.encode("Hello World");

// send data
writer.write(data);
```

Listing 2.7: Sending Data on an Unidirectional Stream

Furthermore, it is possible to close a stream by calling `writer.close()`. This will continue to send any previously written data before closing the stream for good, which will also prevent any further data to be sent on the stream as well as signaling the server not to expect any more data on that stream. Finally, there is also `writer.abort()` which will immediately close the stream and discard any data written to the stream.

Naturally it is also possible to listen for incoming Unidirectional Streams. To do this one is able to get a Reader from the `incomingUnidirectionalStreams` field of the WebTransport object. Read operations from the Reader will return `ReadableStreams` from the Streams API [22].

```
// Listen for incoming Unidirectional Streams
const incoming_uni_reader = quic.incomingUnidirectionalStreams.getReader
    ();
// Read from Reader
```

Listing 2.8: Receiving an Unidirectional Stream

Bidirectional Streams

Bidirectional Streams are essentially two Unidirectional Streams, one initiated by the client and the other by the server. They allow both client and server to send and receive data on the same stream simultaneously. Again Bidirectional Streams can be initiated by both the client and the server. A Bidirectional Stream can be created by invoking the `createBidirectionalStream()` method of the WebTransport object. This stream behaves just like the datagrams field of the WebTransport as seen in listing 2.6, where it has a `readable` and `writable` fields which contain the Reader and Writer, respectively.

```
// Create a Bidirectional Stream
const stream = await quic.createBidirectionalStream();

// "receiver" is equivalent to an incoming Unidirectional Stream
// used to receive data payloads on the Bidirectional Stream
const receiver = stream.readable;
```

```
7
8  // "sender" is equivalent to an Unidirectional Stream
9  // used to send data payloads on the Bidirectional Stream
10 const sender = stream.writable;
```

Listing 2.9: Using Bidirectional Streams

Analog to Unidirectional Stream the `incomingBidirectionalStreams` field can be used to receive from the server initiated Bidirectional Streams.

```
1 // Listen for incoming Unidirectional Streams
2 const incoming_bidi_reader = quic.incomingBidirectionalStreams.getReader
      ();
3 // Read from Reader
```

Listing 2.10: Receiving an Bidirectional Stream

2.4 RTP / RTMP

Before adaptive streaming using HTTP (Dash, HLS) have become the prevalent media streaming protocols, the Real-Time Transport Protocol (RTP) and Adobe's proprietary Real-Time Messaging Protocol (RTMP) were used instead. RTMP specifically was used for the widely popular Adobe Flash Player to streaming audio, video and any other kind of data to the users.

Noteworthy of these protocols is their usage of UDP as transport protocol as well as sharing other similarities to QUIC.

For example both support encryption out of the box. However, encryption is not mandatory and is only optional feature which may be enabled if desired [24]. Furthermore, RTP also offered multiplexing of multiple streams using a single session. However, it was strongly discouraged to send different media types on the same session using multiple streams and rather open separate sessions for every type of media instead to prevent a myriad of possible problems [24].

With the "Death of Flash" [25] on December 31, 2020 RTMP is no longer in use for streaming media. Nowadays end-user devices no longer come with support for RTMP. However, RTMP is still widely used on the backend for data ingestion as of 2021 according to a report by Wowza [26].

Today RTP still finds use in web browsers as part of the WebRTC API [27, 28].

In a nutshell the rise of QUIC in the media streaming field could be seen as coming full circle back to UDP based transport protocols, starting out using RTMP and RTP (UDP) in the mid 90s to HLS and Dash (TCP) around 2010 back to QUIC (UDP) [29].

2.5 Concurrent Approaches

QUIC has been eyed as the next big thing in media streaming for a few years now and it has grabbed enough attention to warrant the creation of an IETF working group, Media over QUIC (moq), in October 2022 [30].

The main objective of the moq group is to develop a low-latency media delivery protocol based on QUIC for distribution and ingest of media. The scope of application for this protocol will include live streaming, media conferencing and multiplayer video gaming [30].

In parallel the group members are implementing various approaches using QUIC for a variety of use cases. Projects include moq-rs by Luke Curley and Mike English [31] as general purpose live streaming protocols for streaming as well as ingesting live streams, MetaMoQ by Meta as generic relay and protocol for live streaming, video conferencing and text chat, Quiche by Google as chat client and more [32].

One noteworthy aspect of these approaches so far is that all of them are using the WebCodecs API [33] and an HTMLCanvasElement to play their videos, rather than MSE and an HTMLVideoElement as I will be doing, inspired by Dash.js.

Their most recent achievement is the second version of their standardized protocol Media over QUIC Transport (MOQT) IETF draft outlining the core behavior of MOQT. MOQT will be able to publish media by producers and allow a multitude of viewers to consume said media by subscribing to it. Additionally, MOQT will allow distribution of content over highly scalable networks with low latency [34].

3 Requirements

This section will go over the requirements needed to implement the WebTransport and HTTP/1.1 servers in Rust. To largely simplify the complexity of this work, I needed to find a fitting Rust crates (packages in Node.js, python, modules in Go or libraries in C) that implement the WebTransport protocol and provide utilities to help with the implementation of an HTTP/1.1 server to adapt the original webtransport-go and express.js servers into Rust. In addition to this I first had to learn Rust itself, since up until this thesis I haven't had any previous experience with Rust.

3.1 Learning Rust

When researching Rust I quickly comes across "The Rust Book" `The Rust Programming Language` [35] as well as the webpage "Rust by Example" [36]. Both offer an extensive insight into the Rust programming language.

The Rust Book goes over the intricacies of Rust in technical detail in very clear and easy to follow instructions with concise code snippets. It starts with a "Getting Started" guide, detailing how to install the Rust toolchain on the most common operating systems Linux, maxOS and Windows, as well as showing how to create and run a rudimentary "Hello, World!" program and also how to create a project using Rust's package manager Cargo.

Following the setup introduction the Rust Book continues to present the basics of the Rust language which will most likely be used in every single Rust program. For example, it goes over the inherent immutability of variables, the rich type system (which is Turing-Complete by the way [37]), nuances of declaring functions, controlling the flow a program using expressions like `if`, `if ... let`, `for`, `while` and `loop` loops, the variable ownership model, structures, enums, pattern matching and more.

In addition the Rust Book outlines more advanced topics like the hierarchy of a Rust project, how projects are categorized into crates which anyone may publish on `crates.io` [38], error handling, automated testing, functional programming, smart pointers and more.

To learn Rust practically and with more hands-on examples, the webpage "Rust by Example" is fantastic to go through concurrently to the Rust Book. Overall it presents the same topics as the Rust Book but includes more detailed example code snippets. In the Rust Book the code snippets can be executed right inside the browser to see what they would actually do when ran. However, Rust by Example goes one steps further and allows the reader to edit the snippets to play around with the code.

3.2 Finding the right Rust Crates

The aforementioned `crates.io` offers the functionality to search for crates by area of application, which I used to find suitable crates for the HTTP/1.1 server. Searching for the keyword "http" resulted in over 1,000 crates connected to HTTP servers and protocol.

After looking at a few of the results in more detail at random and finding out that as expected they mostly offered the same functionalities with varying syntax, I decided to simply stick to the most popular crate according to download numbers. This happened to be the crate Hyper [39].

Hyper promises to be a leader in performance with support for HTTP/1 and HTTP/2, asynchronous design and its implementation to be tested and being correct.

Finding a suitable WebTransport crate proved to be more difficult. The biggest issue was that I was able to find several promising crates, e.g. Quinn by quinn-rs [40], quiche by cloudflare [41] and neqo by Mozilla [42], however, all of those crates I found where all just pure QUIC implementations and would have required additional work to make them support WebTransport. At further inspection I noticed that neqo does in fact support WebTransport, but I decided against neqo since it doesn't provide a stable release on `crates.io` and it would require to either work with an potentially ever changing code base or incomplete code.

Around this time I was made aware of the project `moq-rs` by Luke Curely and Mike English [31] which also uses Rust and they are using Quinn. Fortunately, to support WebTransport Luke Curely had created an extending crating to enable WebTransport for Quinn, webtransport-quinn [43].

Additionally, the Go module webtransport-go is using the congestion control algorithm CUBIC and Quinn uses CUBIC by default as well, but it is possible to enable an experimental implementation of BBR. It is the assumption that BBR is offering better performance than CUBIC [44, 45].

This made me settle with said crates Quinn and webtransport-quinn for this thesis.

4 Design

This chapter introduces the architectural design of streaming media using WebTransport as well as using HTTP request in place of WebTransport in the browser. This consists of several components including a WebTransport-capable server, an HTTP server, an NGINX server proxy and a website.

4.1 Server

The original project, which I will be replacing, consists of two separate servers, a TypeScript Express [3] HTTPS server and a Go webtransport-go [2] WebTransport server.

The Express Server acts as host for the website, hosting all static files, as well as processing all backend API endpoint calls, such as utility procedures for collecting benchmark data, processing said data and hosting the video segments.

The WebTransport Server provides the functionality to use WebTransport to stream media, as well as hosting the same media files as the Express Server.

Since the goal of this thesis is to replace both servers with two new servers, both of which will be written using the Rust programming language, I decided to combine both of them into a single program running both servers concurrently to simplify the setup and streamline the operation.

The combined server can be started using a variety of command line arguments to customize some aspects of it, as shown in table 4.1.

4.1.1 WebTransport Server

The first half of this server is the WebTransport server and as previously mentioned in section 3.2 Quinn [40] and webtransport-quinn [43] will be used to replace the webtransport-go server. This will also shift the programming language from Go to Rust.

Please note, Quinn and subsequently webtransport-quinn likewise have been implemented in an asynchronous way. This allowed me to pair Quinn with the excellent asynchronous runtime crate `Tokio` [46], which results in many function calls needing to be ended by `.await`. Rust maintains a strict Zero Cost Abstraction policy and in this case this results in asynchronous function to not execute until they are being "awaited".

A detailed overview of the operation of the WebTransport server half can be seen in fig. 4.1. A complete example of an Echo Server using the following Quinn and webtransport-quinn crates can be seen in listing 1 with accompanying Client in listing 2.

The WebTransport server starts by going through all media directory (specified by table 4.1) and parsing all available MPEG-DASH manifest files, extracting relevant key data

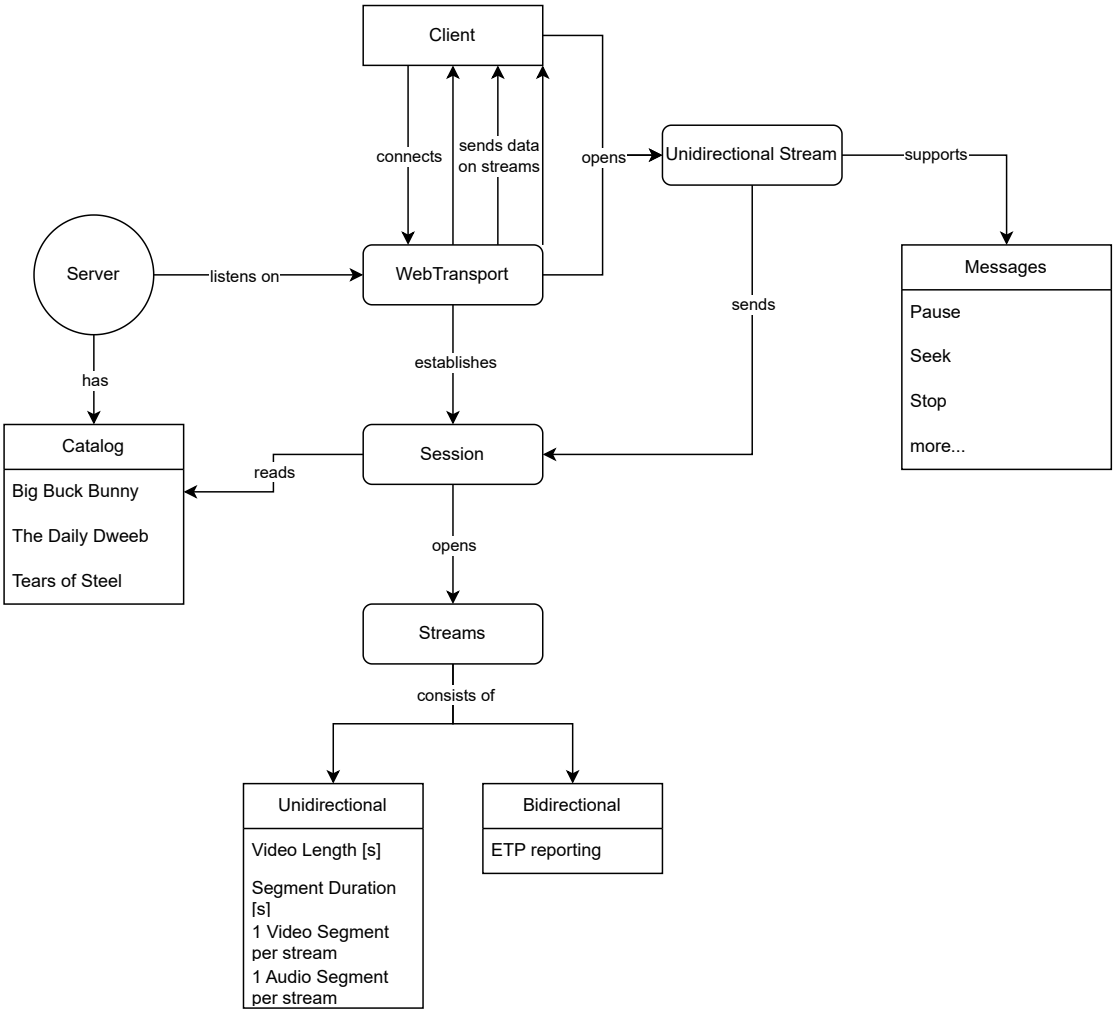

Figure 4.1: Functionality of the WebTransport Server

Key	Data Type	Description
"–cert", "-c"	String	Path to the TLS Certificate file, defaulting to the locally provided Certificates from section 4.3
"–key", "-k"	String	Path to the TLS Private Key file, defaulting to the locally provided Certificates from section 4.3
"–media", "-m"	Vec<String>	Space-delimited list of paths to local media directories containing Manifest files
"–wt_addr"	SocketAddr	Address for the WebTransport server to listen on, defaulting to "[::]:5002" ("[::]" equals IPv6 localhost)
"–http_addr"	SocketAddr	Address for the HTTP server to listen on, defaulting to "[::]:8081" ("[::]" equals IPv6 localhost)
"–docker", "-d"	bool	true or false indicating whether the server has been started inside a Docker container

Table 4.1: Server Command Line Arguments

- starting URL: root path to the segment directory
- video duration: length of the video
- segment duration: length of individual segments
- video title: name of the video
- audio data:
  - path to init: path to the init segment, starting from root (starting URL)
  - bit rate: bit rate in bits per second (bps)
  - additional path data:
    - starting index: index of the first media segment
    - final index: index of the last media segment
    - path: path to media segments, starting from root (starting URL)
    - placeholder string: format string to be replaced by index (e.g. "$Number%05d$" with index 13 results in 00013)
- video data:
  - the same as audio data, but multiple times, once for every available quality

and compiling all collected videos into a `Catalog`. This `Catalog` maps the videos in reference to an acronym of the video's title (e.g. "Big Buck Bunny" is "bbb"). Once the `Catalog` has been created the actual server is instantiated by creating a WebTransport listener according to the Quinn crate.

Since QUIC mandates TLS encryption a WebTransport server requires a TLS certificate and private key, they are acquired according to section 4.3 and parsed to their

respective `Certificate` and `PrivateKey` structures from the `rustls` crate [47]. In addition to the TLS certificates, the WebTransport server requires a config which encapsulates the TLS certificates and a large selection of parameters to configure the behavior of the server. These parameters are initialized with sensible default values but for this thesis I made two notable adjustments to this config.

The first adjustment is changing the Application-Layer Protocol Negotiation (ALPN) to use the value required by WebTransport to enable this server to support WebTransport. Please note this just one step of making this server WebTransport-capable, more are following.

ALPN is an extension of the TLS protocol which allows the participants of the TLS handshake to specify which protocols are going to be used on the connection [48]. In this specific case, WebTransport uses the ALPN identifier `"h3"`.

The second adjustment is changing the transport config to use the BBR congestion control algorithm instead of CUBIC.

Finally, using the created config and the address provided by table 4.1 the server is created as an `quinn::Endpoint`. This Endpoint will be the server that is listening for incoming QUIC connections on the specified address. The listener is started by repeatedly called the `accept().await` method of the Endpoint in an endless loop.

The `accept` method returns a variable of type `Connecting`, I am calling it `conn`. This is a connection with a partially completed QUIC handshake and this handshake can be completed by awaiting it using `conn.await`, which will convert this to a variable of type `Connection`, I am calling it `connection`. Once this handshake is completed Quinn has done its job and we moving over webtransport-quinn to convert this QUIC connection to a WebTransport connection and continue handling the connection.

The first step of this conversion is performing the WebTransport handshake by calling `webtransport_quinn::accept(connection).await`. This returns a variable of type `Request`, I am calling it `request`.

Finally, to complete the connection setup one has to accept the request by calling `request.ok().await`. This returns the completed WebTransport `Session`, which is use to create streams, receive streams, send data on them, etc.

Since a WebTransport connection is established using a URL, it also enables the client to append query parameters to this URL. I am exploiting this feature to send initial settings for the video stream which are set by the client according to table 4.2. This allows me to start opening streams to send data right away without needing to wait for the client to send the same information on an individual stream once the connection has been established, speeding up the progress of delivering the video to the client.

Now that the WebTransport session has been established and initial information required to deliver the media has been processed, the session can start sending the media.

The session handler consists of three parts which are all executed asynchronously using `Tokio`'s `try_join!()` macro. Macros in Rust are special types of functions which are able to take an arbitrary number of parameters, normal functions in comparison are required to specify a fixed number of parameters, to automatically write more code on compile time. In particular `try_join!()` is a macros taking a number of asynchronous

functions and executing them concurrently and only returning from the call either once all function have completed successfully or once one of the functions fails.

The first of the three functions is sending the initial information of the video length and segment duration on unidirectional streams to allow the client to display the video length in the HTMLVideoElement and more efficiently clear the buffer.

The next function is a handler function which listens for incoming unidirectional streams from the client and handles the messages arriving on these streams. All possible message can be seen in table 4.3.

The third and final function is sending the actual media. Beginning with the differentiation whether the requested video is a local VoD or a live low-latency stream by looking at the video title, I am adding the identifier `(LL)` to the end of low-latency videos.

How a low-latency video is stream is detailed in the following section 4.5.

To send a VoD I am again making use of `try_join!` to create two concurrent procedures to send the audio and video tracks in parallel, starting by sending the init segments. These init segments are required by the MSE player to play back media.

Since sending data on a WebTransport stream is always raw bytes, a way to let the client identify the data it has received is required. Since wrapping the segments inside JSON objects can get very inefficient very quickly, I am simply sending the respective track names "audio" and "video" before every segment. Luckily those are both exactly five letter long, making it very easy for the client to analyze the first five bytes of every stream to quickly know what is arriving on that stream.

Once the init segment has been sent I am extracting the final index of the media stream and start an endless loop to send the actual media segments. Beginning by extracting the index of the current media segment that is lined up to be sent next, initially starting with the first index and then incrementing this index by one for every segment that has been sent. The endless loop is broken once the current index has exceeded the final index, indicating that the entire track has been sent.

When the client has sent the message (table 4.3) that the video has been paused, this loop will do nothing until the client has resumed the video.

Using the current segment index it is possible to read the next segment to memory from the local file system and send it on a new stream.

After a video segment has been sent the server-side throughput will be estimated (ETP) (section 5.2.1). Next, the configured ABR algorithm is executed to switch the quality of the following video segments if deemed necessary, this also returns a time value used to wait before sending the next segment. This waiting will prevent overloading the client's buffer and unnecessarily send data too quickly.

Using the wait time from the ABR algorithm and the segment duration of the segments being sent, the server is able to vaguely estimate the expected buffer level the client is supposed to be at, which is used by the ABR algorithm.

One iteration of the sending loop is complete and will now jump back to the start extracting the next segment index and doing this all over again until the loop is broken, having sent the entire video.

After the video has been sent completely the server rolls back up until the starting

function from the three session functions, where it is verified whether the client wants the video in loop. If that is the case this function calls itself recursively starting everything all over again, otherwise the server signals the client the end of this session and closes the connection.

4.1.2 Rust HTTP Server

The second half of the server is the HTTPS server, previously implemented using express.js in TypeScript. In section 3.2 I settled on Hyper [39] and while inspecting the API in closer detail, I noticed that Hyper has two deficits for this thesis.

The first one being that Hyper doesn't offer an out-of-the-box way to convert the standard HTTP server to an TLS encrypted HTTPS server. To overcome this issue I added an NGINX HTTPS reverse proxy as outside entry point. This way I was able to create the NGINX server using the HTTPS protocol and redirecting any incoming requests I want to the Hyper server. I accomplished this by redirecting every request where the URI starts with `/rust/` and prepending said `/rust/` to every request I am sending from the client.

Later I found a couple of complementing Hyper crates (`hyper-tls` and `hyper-openssl`) which would have allowed me to directly implement a HTTPS server using just Hyper and Rust. However, the other deficit I alluded to is also easily remedied by the NGINX server. This deficit was serving static files, like the HTML, JavaScript and image files. It is possible to serve these static files using Hyper, however, that would have added a lot of tedious coding and cluttered the code with unnecessary functionality which is easily done using a handful of lines in the NGINX configuration. In addition to this, NGINX is a highly efficient server and serves these files faster than I would have been able to replicate using Hyper and this also more efficiently distributed the computational load.

Hyper is a perfect fit to accompany the WebTransport server using Quinn, since Hyper uses a very similar asynchronous implementation like Quinn does, allowing the use the same asynchronous runtime crate `Tokio` [46]. This is leading to the same reminder why most of the following function calls require to be ended by `.await`.

A detailed overview of the operation of the HTTP Server can be seen in fig. 4.2. A complete example of an Echo Server using the following Hyper crate can be seen in listing 3 and accompanying Client in listing 4.

The primary server of this setup is the NGINX proxy listening for incoming HTTPS requests. Any requests with their URI starting with `/rust/` will be redirected to the Hyper server and every other requests will be handled directly by NGINX, in this case these requests are only the ones automatically sent by the browser to serve static files, since this NGINX server is the host for the client website.

The Rust server will receive the redirected requests from NGINX (with the prepending `/rust` removed) and process these. The responses will be sent back to the client routed through the NGINX proxy. The primary function of the Hyper server is to host the same media segments as the WebTransport server, hence why it has access to the same Catalog as the WebTransport server only this time the Hyper server doesn't do any processing of them. All it does is sending the requested files to the client. Additionally, there are some

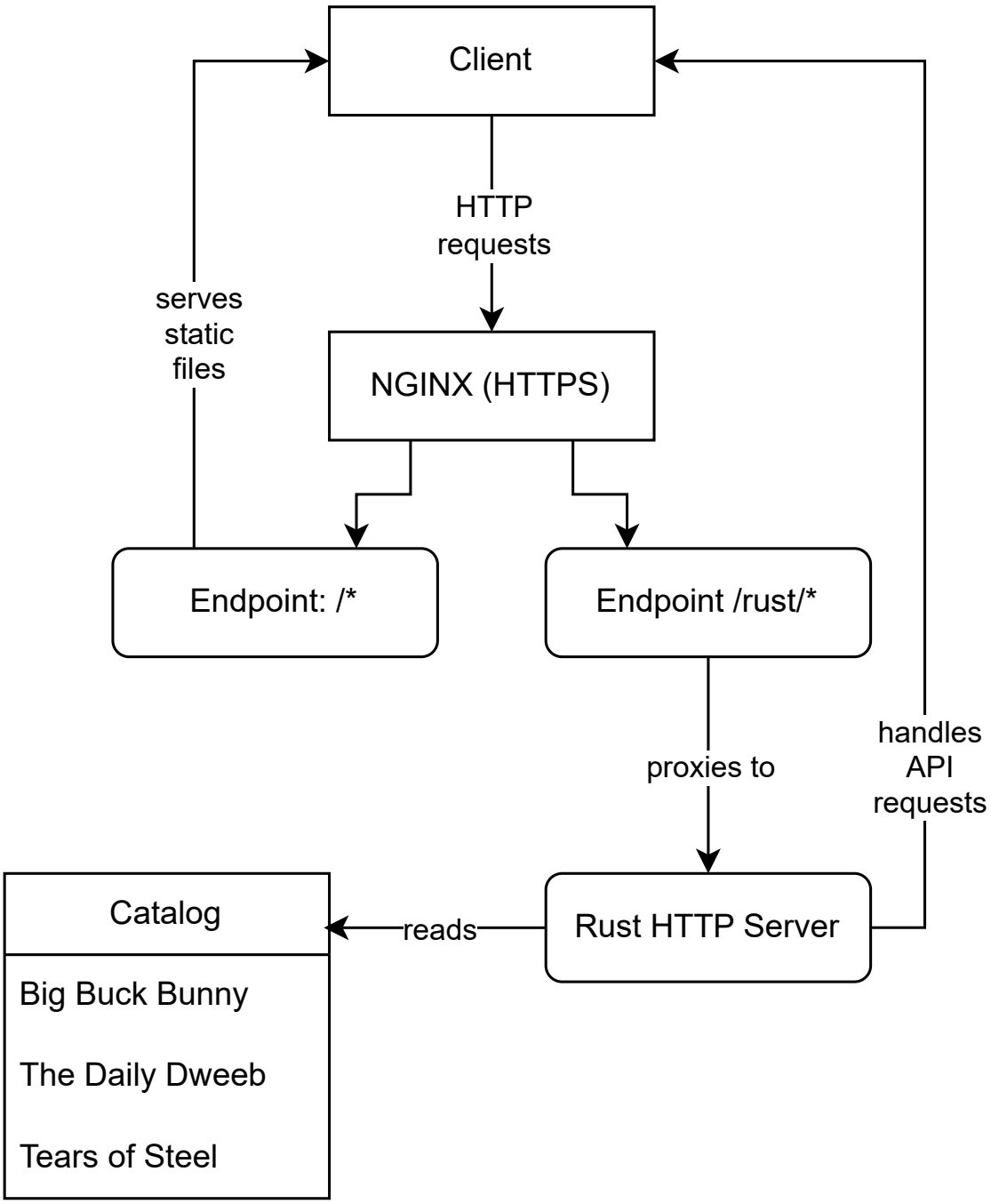

Figure 4.2: Functionality of the HTTP Server

further helper endpoints on the server to make running benchmarks easier and other utility.

To create a server using Hyper one starts by binding the address provided by table 4.1 to an asynchronous `TcpListener` from Tokio using `TcpListener::bind(address).await`. This listener is our the HTTP server's endpoint listening for incoming requests, to keep it in line with the previous WebTransport example I am again calling it `endpoint`.

Very similar to Quinn and WebTransport the listener is started by repeatedly calling the `accept().await` method of the Endpoint in an endless loop to accept incoming HTTP requests. This will return a tuple containing the `TcpStream` and the address of the remote peer from which the request is coming from. A tuple is a list of arbitrary variables. To make further coding easier I am destructuring the tuple returned by the accept function to `(stream, _)`, with `stream` being the `TcpStream`. The remote address serves no purpose for this application and I am ignoring it (denoted by the underscore in the destructuring).

Now that a connecting stream has been established the server can start processing the request using Hyper. To do this I needed to create a service function and to force these requests to use HTTP/1.1 specifically I wrapped this in an http1 connection server provided by Hyper (listing 4.1).

```
1 let io = hyper_util::rt::TokioIo::new(stream);
2
3 tokio::spawn(async move {
4     hyper::server::conn::http1::Builder::new()
5         .serve_connection(
6             io,
7             hyper::service::service_fn(move |req| process_request(req))
8         ).await.unwrap();
9 });
```

Listing 4.1: Hyper Service Function

The connection server, `serve_connection().await`, requires two parameters, an Input/Output handler for the `TcpStream`, `io`, and a service function, `service_fn`, which will process the actual request.

The complementary `hyper_util` crate provides a easy I/O handler for Tokio, which can be easily created in line 1 and passed into the function in line 6.

The service function is created by using Hyper's provided `service_fn` in line 7, which takes a function that processes the request `req` and returns the response body, here called `process_request`.

To improve performance and allow the server to process multiple request concurrently. I wrapped the connection serving inside an `async` block (line 3-9) and letting it be executed in the background concurrently using `tokio::spawn` in line 3.

The two `move` key words in line 3 and 7 denote a passing of ownership from the executing path to the inner executing body following the move.

The variable "req" between the two pipe symbols in line 7 following by the function call `process_request(req)` is called a Closure, which are essentially the same as Callbacks from JavaScript.

Finally, to process the requests I implemented the function `process_request`, which takes the request struct and returns the response struct, a simplified example is shown in listing 4.2.

```
1  use hyper::{Request, Response};
2  use hyper::body::Incoming;
3  use bytes::Bytes;
4
5  async fn process_request(req: Request<Incoming>) -> Result<Response<Full<
       Bytes>>, Error> {
6      println!("{} {}", req.method(), req.uri().path());
7
8      let res = b"Hello, World!";
9
10     Ok(Response::new(Full::new(Bytes::from(res.to_vec()))))
11 }
```

Listing 4.2: Hyper Process an HTTP Request

To keep this example simple, all it does is printing out the request HTTP method and request URI to the standard output (line 7) and returning a default response (line 10) using the created byte string "Hello, World!" as body (line 8).

The `use` statements from line 1-3 are used to bring external function, structures, etc. in the namespace of the current file to make the code more readable.

The function declaration (line 5) reads this function is asynchronous, denoted by the `async` keyword at the beginning, is named `process_request`, takes one parameter of name "req" and type `Request` with its body of type `Incoming` and returns an `Result` of types `Response` in case of success and `Error` in case of failure.

Returning the response from this service function is akin to finishing the request and sending the response back to the client, Hyper handles this automatically.

4.2 Browser Client

The user interface is a web-based browser client using plain HTML, CSS and JavaScript. In terms of functionality I didn't need to do too many changes. The biggest change was the switch from TypeScript back to JavaScript to simplify the development experience and restructuring the file hierarchy by moving the browser code away from Express and into its own directory to streamline the access to it from NGINX.

An overview on how the Client operates to play back video can be seen in fig. 4.3.

The important element of the Client is the HTMLVideoElement used to play the media and the Media Source Extensions API [5] (MSE Player) to control said HTMLVideoElement.

The Client consists of the aforementioned HTMLVideoElement, several control boxes to configure the stream and run benchmarks and various outputs to show previous benchmark results.

The mentioned control boxes allow the user to select which video to watch, in what quality, segment duration (if available), wether to send audio or not and more. More

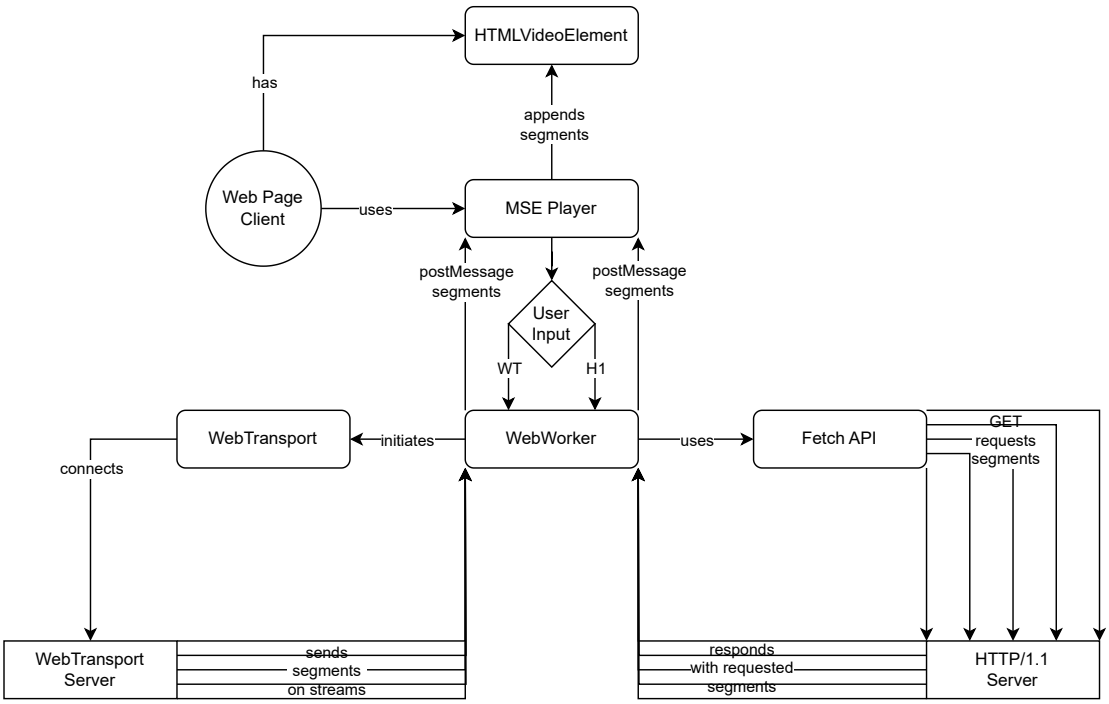

Figure 4.3: Functionality of the Web Client

importantly the user is able to select the method of how the video is streamed by pressing the respective button. This will play the video using the current settings on the page. The available options are

- WebTransport, which will use my own MSE Player and fetch the segments using WebTransport from my Quinn server
- HTTP/1.1, which will use my own MSE Player and fetch the segments using the Fetch API from my Hyper server
- Dash, which will use the official Dash.js (v4.7.1) build and fetching the segments from my Hyper server

Pressing the WebTransport or HTTP/1.1 button will start my MSE Player by initializing The MSE API and invoking a WebWorker which will be responsible for handling the WebTransport connection and fetch requests respectively. A WebWorker is a browser API which allows a web page to run a specific script file in a separate thread.

Whenever the WebWorker receives a chunk of data from their respective server they will send this data back to the MSE Player on the main thread, using `postMessage(data)`, to allow the MSE Player to append this chunk to the HTMLVideoElement's buffer.

4.2.1 WebTransport

In case of the WebTransport WebWorker, the Worker will create a WebTransport connection using the constructor `new WebTransport(url, options)`. The URL will be the address of the Quinn server, the options can usually be omitted, however, there is a caveat when using this in Google Chrome as detailed in section 4.3.

Since one is using an URL to connect to the WebTransport server, it is possible to append query parameters to the URL, which allows the Client to send the requested video settings alongside the initial handshake to reduce the time it takes for the media to start arriving. All queries I have implemented can be seen in table 4.2.

Key	Data Type	Description	Default
video	String	Video Identifier (e.g. bbb for Big Buck Bunny)	
id	UUID [49]	unique Client ID	
segdur	u8	Segment Duration ID of the video (only bbb)	1
abr	String	Name of the desired ABR algorithm	Basic
minBuffer	f32	Target minimum Client Buffer length [s]	5.0
maxBuffer	f32	Target maximum Client Buffer length [s]	30.0
noAudio	bool	Don't send Audio	false
reportETP	bool	Report ETP values on Bidirectional Streams	false
quality	u64	Desired maximum Video Quality as resolution height [px]	720 (≈ 2mbps)
static_bitrate	bool	Disable Quality switching, maintain requested Quality	false

Table 4.2: WebTransport URL Query Parameters

The keys `video` and `id` don't have a default value in table 4.2 because they are mandatory and the connection is refused when they are not provided or when the `id` already exists.

As soon as the connection has been established the server will immediately start sending the media segments on unidirectional streams, which the WebWorker is receiving and forwarding back to the MSE Player on the main thread.

The WebTransport connection will remain open until the entire video has been sent, an error has occurred or either side has manually closed the connection. While the connection is open, the client is able to send specific message to support the server to maintain a stable stream and allow the client to make adjustments to the video on the fly, for example pause the video, which will suspend the transmission of additional segments until resuming the video or to seek on video timeline to either re-watch an earlier scene or skip to a later point in the video.

The messages are JSON encoding (seen in listing 4.3) and contain an identifying type `"name"` and an accompanying payload `"value"`, both of which are of data type `String`. All message that I have implemented on the server are listed in table 4.3.

```
1 {
```

```
    "name": string,
    "value": string
}
```

Listing 4.3: Client Message JSON

Message (JSON)		Description
Type	**Value**	
stop	-	Ends listening loop
loop	bool	Send Video in loop
seek	f64	Target seek time [s] from where to continue sending
pause	bool	Client paused Video, suspend sending Segments
minBuffer	f32	Update minimum Buffer Target Length
maxBuffer	f32	Update maximum Buffer Target Length
static	bool	Stop switching Quality, maintain current Quality
buffer	f32	Client is reporting a too low Buffer Length [s], overriding estimate

Table 4.3: Client Messages

The data type `bool` is a Boolean in Rust and can have the value `true` or `false`. The data types `u8` and `u64` are unsigned integer (natural numbers) with the number indicating the number of bits this variable uses. Analog to unsigned integer, `f32` indicates a floating point number (rational numbers) using 32 bits.

4.2.2 HTTP/1.1

The order of operation for the HTTP/1.1 WebWorker and its requesting of media segments is only rudimentary and only serves as comparison for benchmarks. All it does is request the media segments in their correct order using the fetch API and forwarding the responded segments back to the MSE Player on the main thread, just like the WebTransport WebWorker.

4.3 Certificates

Both servers, WebTransport and NGINX, require TLS certificates to function. Normally, one would acquire them official sources, which would be valid for usages on a public network.

However, since I am only doing everything locally, self-signed certificates are adequate. Self-signed certificates are only valid for a local network, but they are free and easy to get.

An easy to use tool for this is the program `mkcert` which can create self-signed certificates and also automatically add them to a browsers trusted store to make them valid in the eye of the browser.

With WebTransport there is, however, a minor caveat to keep in mind. A TLS certificate must not have a validity period of more than 14 days. Out-of-the-box `mkcert` does not support setting a custom expiry date of the certificate and always creates them with a validity time of two years, which is why I am using a forked version of `mkcert` which allows to specify an expiration period. I found this version and script in an GitHub repository by Luke Curely which has since been overridden.

```
1 #!/bin/bash
2
3 go run filippo.io/mkcert -install
4
5 go run filippo.io/mkcert -ecdsa -days 10 -cert-file "localhost.pem" -key-
    file "localhost-key.pem" localhost 127.0.0.1 ::1
6
7 openssl x509 -in localhost.pem -outform der | openssl dgst -sha256 -
    binary | xxd -p -c 256 > localhost.hex
```

Listing 4.4: Generating Self-Signed TLS Certificates

The code snippet of listing 4.4 is a shell script to create the TLS certificate and private key. Line 1 defines in the which shell this script will be executed, here in `bash`. The certificates created by `mkcert` are added to the trust store of browsers using the install command found in line 3. The actual certificate and private key are created in line 5 using the Elliptic Curse Digital Signature Algorithm (ECDSA) with a validity time of 10*days* (this is what the forked version added). The arguments `cert-file` and `key-file` denote the output path for the certificate and private key, respectively, and the final three arguments *localhost*, 127.0.0.1 and :: 1 all give the names for which the certificates are valid. All three describe localhost as domain, IPv4 address and IPv6 address, respectively.

Finally, on line 7 the fingerprint of the certificate is created using `openssl`. This fingerprint is currently required when using WebTransport in Google Chrome to allow the use self-signed certificates.

To make WebTransport work in Google Chrome, one needs to parse the certificate fingerprint to hexadecimal and use an HTTP request to acquire the hexadecimal fingerprint.

```
1  // array collecting fingerprint elements
2  let mut fingerprint = Vec::new();
3
4  // read fingerprint file in bytes
5  let data = std::fs::read("path/to/fingerprint.hex").unwrap();
6
7  // split bytes in pairs of two
8  let chunked = data.chunks(2);
9
10 for chunk in chunked {
11     // parse chunk to UTF-8 String
12     let hex_string = String::from_utf8(chunk.to_vec()).unwrap();
13
14     // convert String to hexadecimal byte
15     let hex_chunk = u8::from_str_radix(&hex_string, 16).unwrap();
16
```

```
17      // append fingerprint element
18      fingerprint.push(hex_chunk);
19  }
```

Listing 4.5: Parsing the Certificate Fingerprint

To parse the fingerprint one starts by reading the fingerprint file, line 3 in listing 4.5. This returns a bytes array, which is to be splitting into parts of size two on line 5. Then one will loop over each of those parts using a `for ... in` loop on line 6. In this loop each of those chunks will be converted to a `String` on line 8 and then parsed to a hexadecimal byte on line 10. Finally, we will append this byte to a `Vector` which was created on line 1 by pushing it to the `Vector` on line 12. The fingerprint is now ready to be sent to the client upon request.

On the client-side one can use the fetch API to request the fingerprint as plain text, then parse the response to the fingerprint using `JSON.parse(response)`. Finally, one can create the WebTransport options parameter (see listing 4.6), which must be used in Google Chrome, as seen in section 4.2.1.

```
1  {
2      "serverCertificateHashes": [
3          {
4              "algorithm": "sha-256",
5              "value": new Uint8Array(fingerprint)
6          }
7      ]
8  }
```

Listing 4.6: The WebTransport Options Object

4.4 Limiting the Bandwidth

The problem with local development for network related applications is the unrealistic and essentially unlimited network bandwidth. To overcome this, it was required to synthetically limit the bandwidth.

The Ubuntu operating system provides a perfect tool for this use case. The program Traffic Control (tc) [50] has the ability to set so called `qdics` which can be applied to individual network interfaces to throttle the throughput and introduce latency.

Additionally, to simplify the setup on new machines and to execute the bandwidth limiting in a safe space to prevent accidentally messing up something on my computer, I deployed the whole project to a Docker Container. A Docker Container is similar to a virtual machine, but more lightweight and specialized to only run one specific program, my servers in this case.

This way I am able to safely limit the bandwidth of the server inside the Docker Container without the limiting affecting my machine. To enable the limiter I have added an HTTP endpoint on the Hyper server for limiting the bandwidth and resetting it again.

The endpoint for setting the bandwidth limit expects a JSON object (seen in listing 4.7) to perform the limiting.

```
{
    "speed": number,
    "profile": String,
    "trajectory": [
        {
            "speed": number,
            "duration": number,
            "latency": number
        }
    ]
}
```

Listing 4.7: The Bandwidth Limit Object

There are two possible ways to limit the bandwidth. The first option is a static bandwidth with no time limit by using the the desired limit as value for the field "speed" in kilobits per second (kbps) and setting the "profile" to "NONE", in this scenario "trajectory" is ignored. The other option is to set a trajectory, which is a variable bandwidth limit. This is accomplished by setting "profile" to anything other than "NONE" and filling the array of "trajectory" with as many elements as desired. "speed" is the same as before, "duration" is the time in milliseconds (ms) for how long the speed is supposed to last for before switching to the next and "latency" is the simulated latency in milliseconds (ms), here is the field "speed" being ignored.

Examples for a static bandwidth limit and a trajectory can be seen in listing 5 and listing 6, respectively, with a visualization of the trajectory in fig. .1.

4.5 Pseudo Live Streaming

To simulate a low-latency live streaming I used the FFmpeg command line builder by Akamai [51] to create a script that takes a file as input and uses this file to transcode an endless low-latency live stream.

Rather than writing the output to local files, I instead use an URL to an HTTP endpoint on my Hyper server. This will send the transcoded segments directly to the server. The segments are transmitted using HTTP/1.1 chunked transfer encoded, which allows the receiver to read the request body while it is still arriving. This is greatly cutting down on delay to great an as low as possible latency.

On arrival, the server will cache the most recent segments for both tracks (audio and video) in memory. When requesting a low-latency stream with the client with any of the possible solutions (WebTransport, HTTP, Dash) this cache will be used to respond to the client with the most up-to-date segments.

4.6 ABR Algorithm

In preparation for future expansions to this project, I have added a few rudimentary ABR algorithms to send the segments with WebTransport in a slightly more orderly fashion. I have added three such algorithms, as detailed in table 4.4.

Name	**Description**
Basic	• Maintains the Quality set by Client (no Switching) • Sends Segments as fast as possible, as long as the Client Buffer Level Estimation is below the requested Minimum • Otherwise, waits for the Segment Duration before sending the next Segment
Simple	• Switches to highest possible Quality according to ETP, while not going above the quality set by the client • Attempts to stay as close as possible to the Maximum Client Buffer Level requested • If the Buffer Level would be exceeded by a Segment, wait for the Segment Duration before sending the next
StaticSimple	Same as Simple, but doesn't switch Quality. Maintains the Quality set by the Client

Table 4.4: Available ABR Algorithms

5 Implementation

This chapter describes the implementation of the media streaming. The servers are based on Quinn and webtransport-quinn for the WebTransport server and Hyper for the HTTP server using the Rust programming language. The client is a HTML browser web page.

5.1 Environment

The following software, operating systems and hardware were used for the implementation:

	Work Laptop	**Private Laptop**
Vendor	Lenovo ThinkPad L14 Gen 1	Framework Laptop 13
Operating System	Ubuntu 22.04.3 LTS	Ubuntu 23.10
CPU	AMD Ryzen 7 pro 4750u	Intel i5 1340p
RAM	16 GB	32 GB
Browser	Google Chrome 119	
Code Editor	Visual Studio Code	
Rust	Version 1.72.0	
NGINX	Version 1.24.0	
Docker	Version 24.0.7	

Table 5.1: Hardware and Software Environment

5.2 Important Implementation Aspects

Over the course of implementing the conversion from using the Go programming language and the module webtransport-go to the Rust programming language and the crates Quinn and webtransport-quinn, I came across a plethora of issues which are linked to the stricter coding limitations when using Rust in comparison to Go.

This section will go over these issues and what changes I had to make to overcome them.

5.2.1 Estimated Throughput

The first and gravest problem is not directly linked to the Rust programming language itself, but rather the crate Quinn I am using to implement the WebTransport server.

The webtransport-go module from Go was providing the option to enable QLOG [52] logging on a per session basis. QLOG is standardized logging format which allows close monitoring of the performance of a server. Among these QLOGs are entries which keep track of every individual packet which are sent to the clients, including the metrics – bytes in flight and smoothed round-trip-time (RTT) with the corresponding time stamp – which allowed me to fairly accurately estimate the server-side through.

QLOG logs every entry with a time code when the specific event took place which allowed me to roughly assign which entries belong to which code operation. I took the timestamps of just before and after sending data on a stream. Using those timestamps I was able to extract all relevant log entries from the QLOG digest and compile them to an ETP value.

Every entry contained the aforementioned field `bytes_in_flight`, measured in bytes, and `smoothed_rtt`, measured in milliseconds. For every entry I calculated an individual ETP value using the following formula

$$ETP_i = \frac{bytes\_in\_flight * 8}{smoothed\_rtt} \tag{5.1}$$

and collecting them in a list. Once I had collected all ETP values from every extracted entry, I reduced this list of ETP values to one value by taking the arithmetic mean

$$ETP = \frac{\sum_{i=1}^{N} ETP_i}{N} \tag{5.2}$$

One goal of this thesis was to replicate this ETP calculation using Rust with the expectation that the underlying BBR congestion control algorithm would further improve this estimation.

However, Quinn does not currently implement QLOG and implementing it has been on the table with an open GitHub Issue since 22nd July 2019 [53] and only recently Luke Curely has stated to be open to implement this, eventually.

I was made aware that internally Quinn has already implemented a bandwidth estimation inside of the BBR implementation which they use to pace the congestion control. Important to here is that in the documentation for the primary BBR struct warn the user from using this BBR implementation because it is currently still in an experimental state.

My next attempt to estimate the throughput was to use the internal bandwidth estimation from the BBR pacing. This required making changes to the internal code of Quinn. I have forked the official Quinn GitHub repository and added it as a submodule to my project and patched my project to use this local version of Quinn rather than the official version from crates.io. This allowed me to make changes to Quinn and directly use these in my server.

Since the estimations from Quinn are getting calculated periodically, they can no longer be roughly assigned to specific operations in time, my first instinct was to check how many of these estimations get generated per second. Based on this number I would choose a good amount and then combine this many estimations to one ETP value which I would pass to the ABR algorithm where it is required.

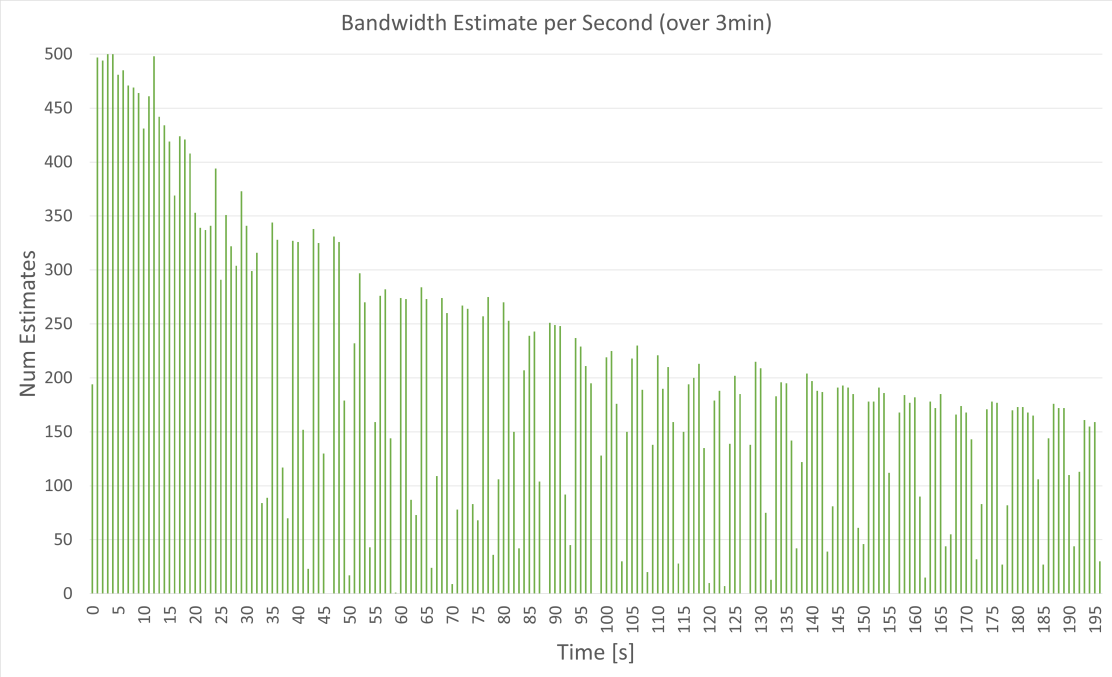

Figure 5.1: Number of Bandwidth Estimates per Second

The result of the estimates per second are shown in fig. 5.1 and on average there were approximately 190 estimates per second over a time period of roughly 200 seconds, which made me choose the amount of 100 values to use to condense them to one estimate for further tests.

The next experiment I conducted was seeing how those 100 values compare to a static bandwidth limit. For this I collected the estimates in batches of 100 in a file each, sorted them in ascending order and plotted them along side the expected bandwidth limit. In total I collected 250 batches creating 250 plots, however, they all showed the same results, represented by the first 25 plots shown in fig. 5.2.

Overall it can be observed that the majority of estimates are well below the expectation and only a handful are close or above the expectation. Overall the estimate was approximately $4,300kbps$ compared to the expected limited of $5,000kbps$.

Based on those results I continued experimenting with a static limit by cutting off a percentage of the lowest values to improve the estimation. I tested cutting of the lower 20% (fig. 5.3), 30% (fig. 5.4), 40% (fig. 5.5) and 50% (fig. 5.6). Each step slightly improve the estimate going from $\approx 4,550kbps$ using a cutoff of 20% to $\approx 4,750kbps$ with a cutoff of 50%.

At this point I realized that it is not that valuable to keeping doing these experiments using a static limit and moved on to testing a trajectory.

To make experimenting using a trajectory more efficient, I also decided to test different strategies simultaneously to compile multiple values to a single estimate.

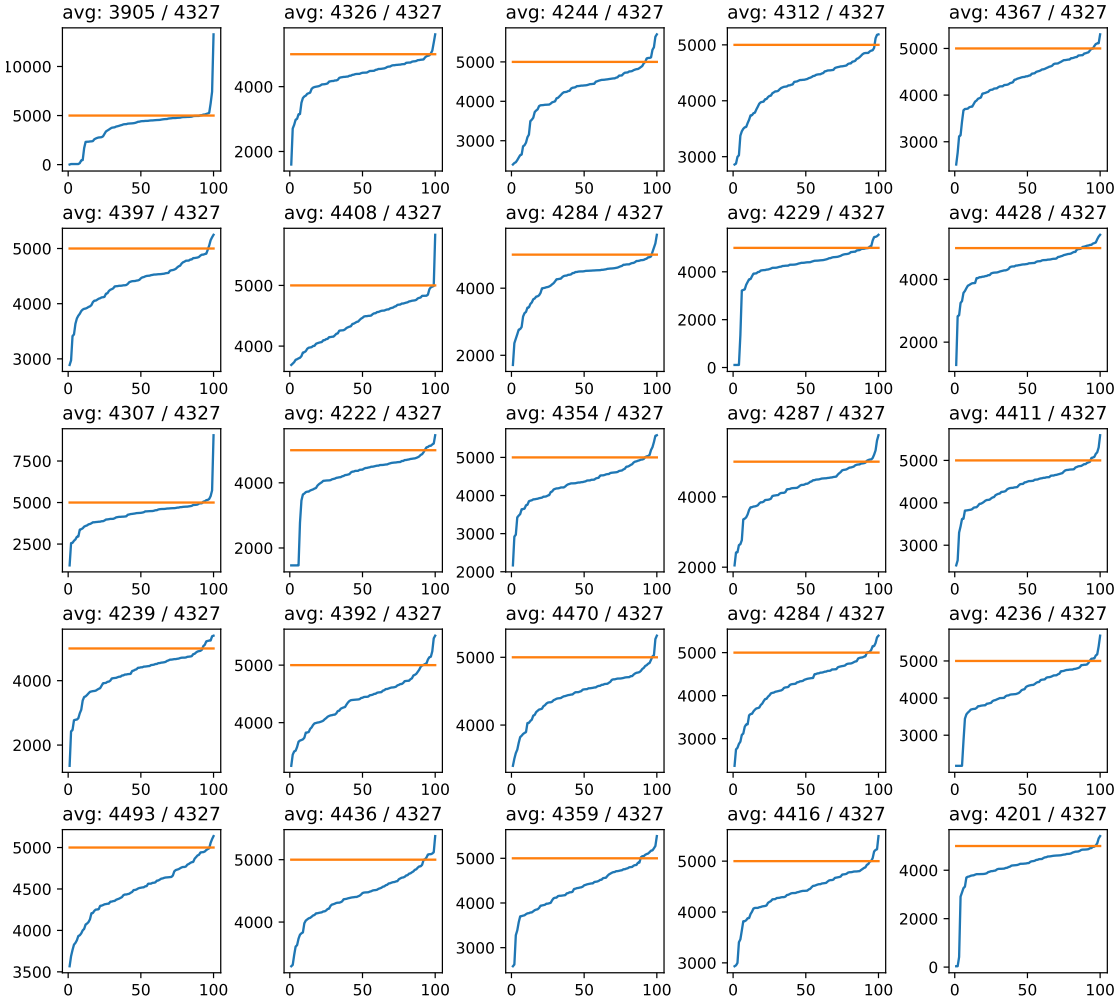

Figure 5.2: Batches of 100 Estimates compared to Static Bandwidth Limit
x-Axes are the indices of values
y-Axes is the Bandwidth in kbps

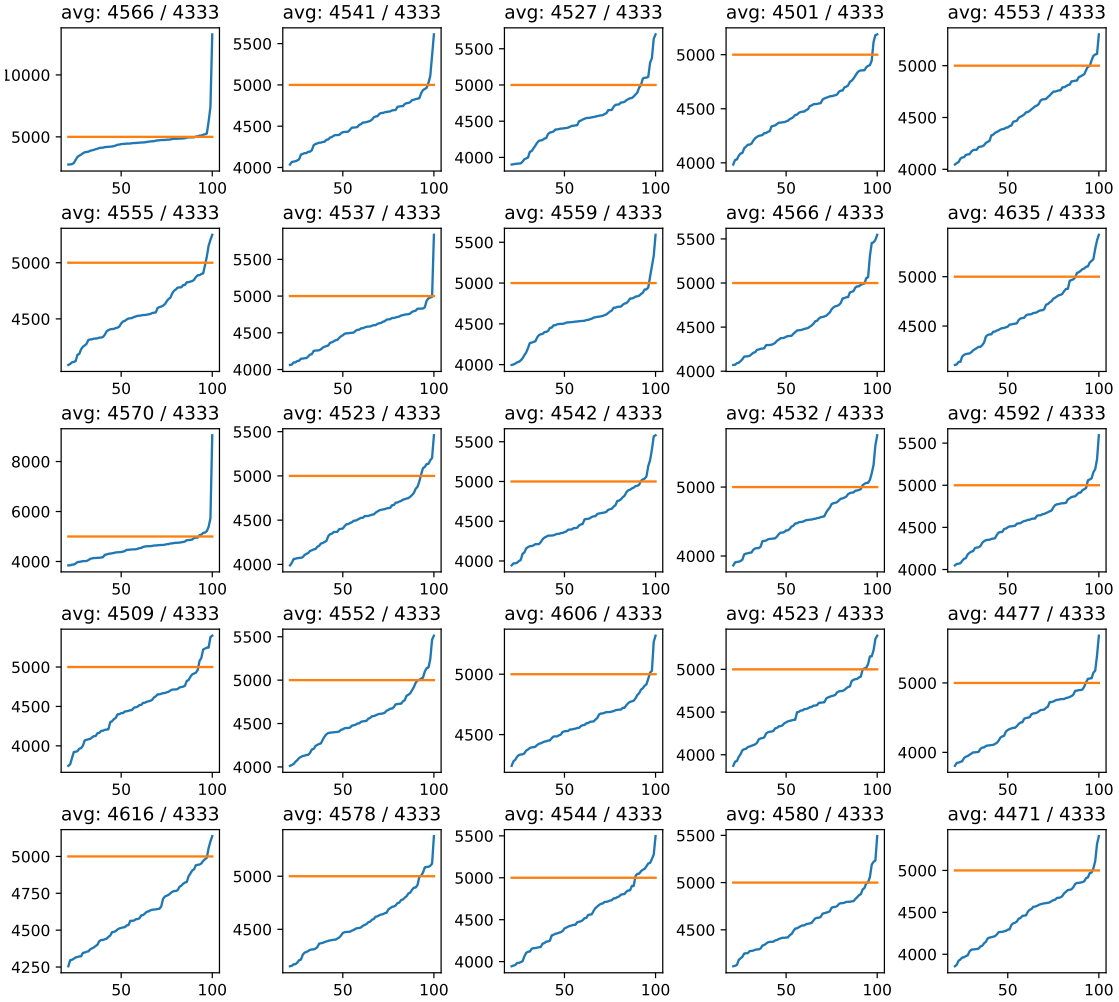

Figure 5.3: Batches of 100 Estimates compared to Static Bandwidth Limit (cutoff 20%)
x-Axes are the indices of values
y-Axes is the Bandwidth in kbps

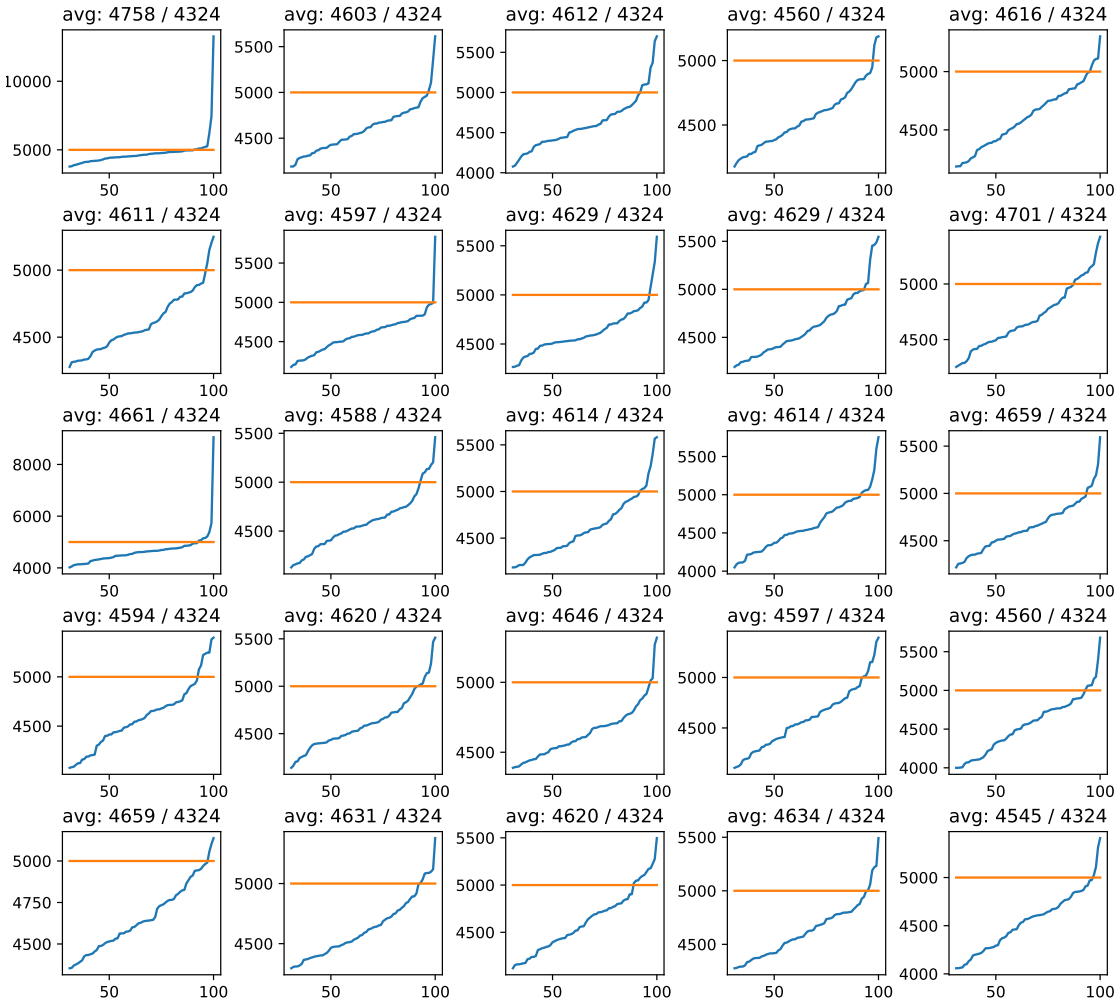

Figure 5.4: Batches of 100 Estimates compared to Static Bandwidth Limit (cutoff 30%)
x-Axes are the indices of values
y-Axes is the Bandwidth in kbps

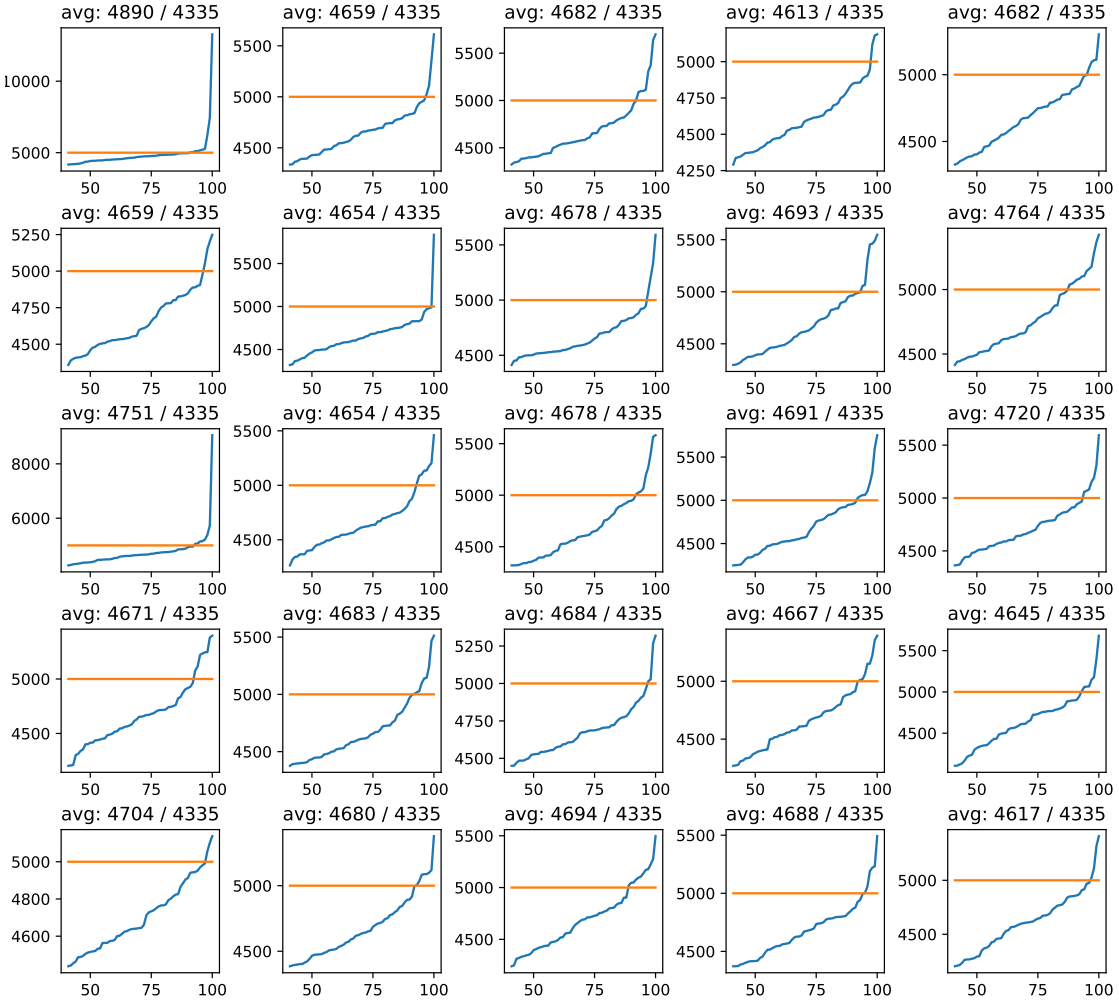

Figure 5.5: Batches of 100 Estimates compared to Static Bandwidth Limit (cutoff 40%)
x-Axes are the indices of values
y-Axes is the Bandwidth in kbps

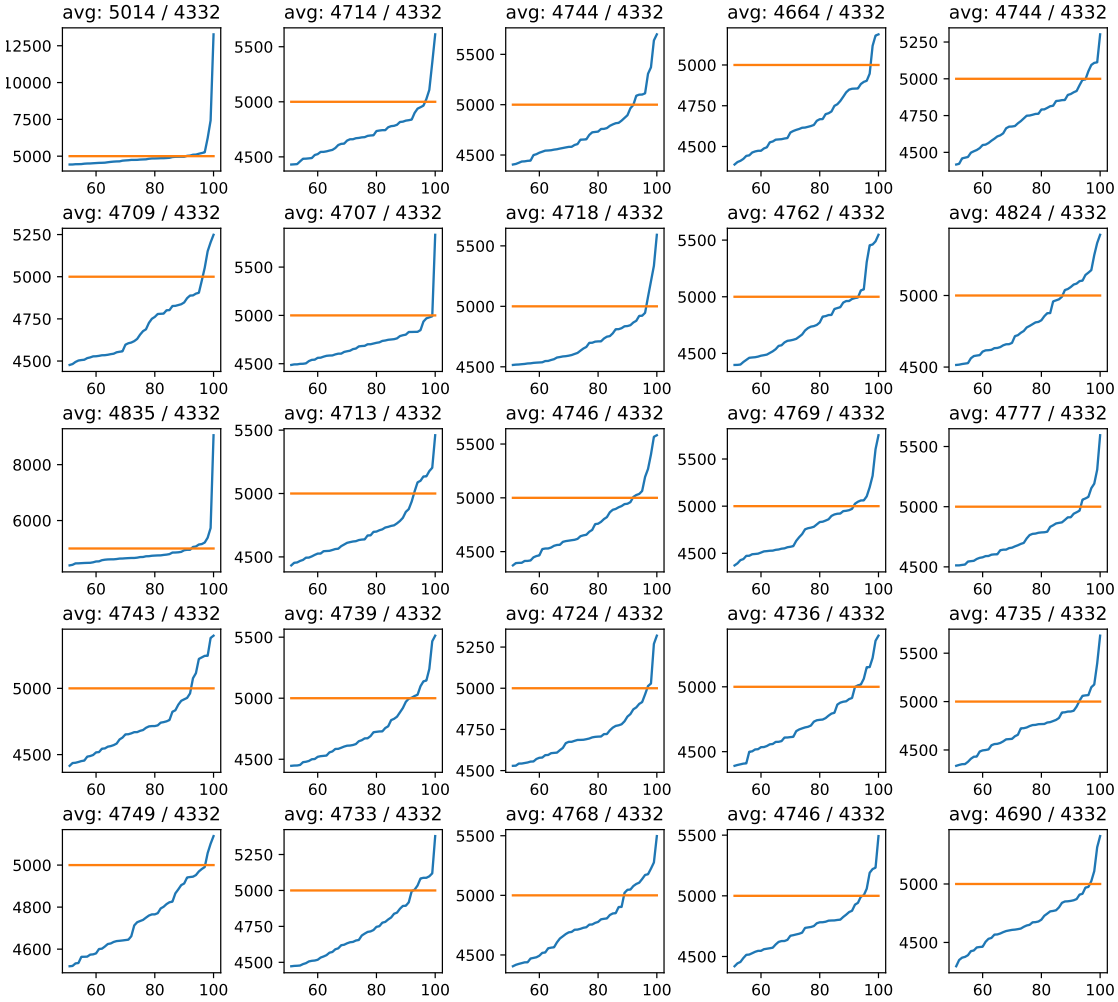

Figure 5.6: Batches of 100 Estimates compared to Static Bandwidth Limit (cutoff 50%)
x-Axes are the indices of values
y-Axes is the Bandwidth in kbps

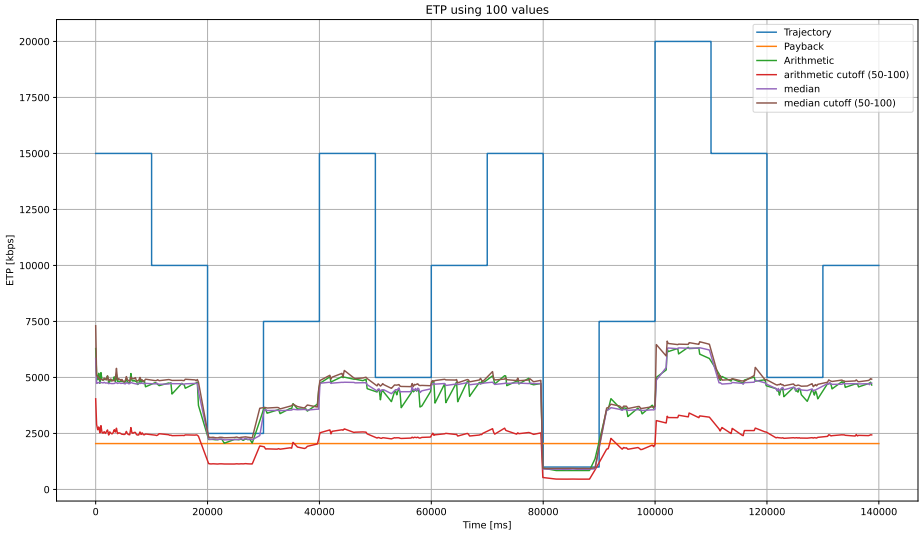

Figure 5.7: Trajectory ETP Estimate playing 720p

In fig. 5.7 the result of testing 100 value batches with the following strategies is shown

- Arithmetic Mean
- Arithmetic Mean (cutting off the lower 50%)
- Median
- Median (cutting off the lower 50%)

It can be observed that, with the exception of the cutoff arithmetic mean, all strategies perform roughly the same within margin. This plot also shows the futility of my previous experiments using a static limit of $5,000kbps$ since the estimates seem to be hard-capped at $5,000kbps$ with the only exception being the section where the limit was set to $20,000kbps$. Anything below $5,000kbps$ like $2,500kbps$ from $20s$ to $30s$ and $1,000kbps$ from $80s$ to $90s$ were quite accurate.

I had the assumption that the video playback bit rate was perhaps too low, since I was using a $720p$ ($\approx 2,000kbps$) video for this testing. Next up I ran the very same test but change the video bit rate to $4k$ quality at $\approx 17,500kbps$, shown in fig. 5.8. However, this resulted in nearly the same outcome. The only difference being unusual fluctuations in the two section where the limit was set to $\approx 7,500kbps$ in the periods $30s$ to $40s$ and $90s$ to $100s$.

Any further experiment resulted in the same outcome, which lead me to the decision to declare the pursuit of estimating the server-side throughput a failure for the time being.

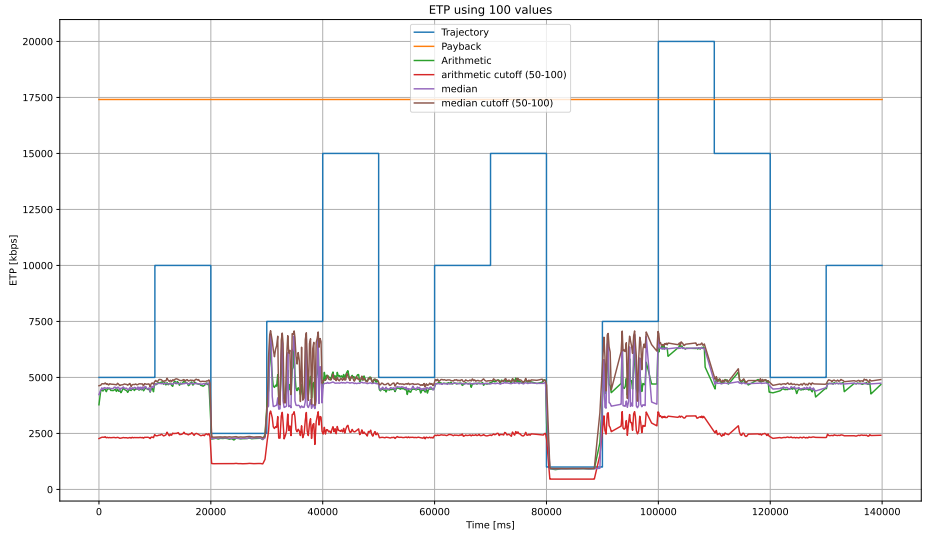

Figure 5.8: Trajectory ETP Estimate playing 720p

5.2.2 Persistent QUIC Streams

Previously in Go, I was able to open a single stream for audio, video segments and messages each and save these on a struct to reuse them for every transmission. This way I didn't need to open a new stream for every payload I wanted to send. However, in Rust using Quinn I wasn't able to replicate this.

Whenever I opened a new stream and saved it to a struct the stream would close as soon as the function was done, resulting in the send function of the stream to throw an error when trying to send the next payload.

To circumvent this issue I am now opening a new stream for every transmission. No changes were required on the Client to make it support individual streams instead of singular persistent streams.

5.2.3 Interior Mutability

In Go it is very easy to create concurrently running function by simplify adding the keyword `go` before a function call, which would start executing this function in a so called `Goroutine`, a word play on `"Go"` and `Coroutine`. In addition Go is very lenient on memory accesses using `Goroutines`, where it possible to easily start these additional threads and be able to access anything that would also be accessible when running the same function outside of a `Goroutine`.

However, even during my usage of them I came across the problem of having multiple accesses on the same data simultaneously, which required the use of wrapping these data accesses inside a Mutual Exclusion (`Mutex`). A Mutex is common programming

paradigm when working with concurrency, which allows certain sections of the memory to be `locked`, which prevents anyone else from accessing it, while the data is accessed and `unlocked` to release the lock and allow the next access.

In this case I used a special type of Mutex called `RwLock`, Read/Write Lock, which instead of generally locking the access to data, allows to specify which type of access one wants of the data, read or write access. Since only writing to memory can great complication if someone else wants to read from the same memory in the same moment, this `RwLock` allows to specifically lock the memory using either a `ReadLock` or `WriteLock`. The advantage of this is that it is possible to create as many `ReadLocks` as you want, since reading does not change the data in the memory. However, if there is just one `WriteLock` present, no other lock can be opened until the `WriteLock` has been lifted.

This comes into a much bigger play in Rust, especially when using asynchronous execution or multi-threaded programs.

Since the entirety of my Rust servers use the asynchronous environment of Tokio and Rust being a lot more stricter with memory management to achieve its excellent memory safety, I was required to use Tokio's implementation of the `RwLock` Mutex, as well as the smart pointer `Arc`.

The smart pointer `Arc` stands for "Atomically Reference Counted" and is a thread-safe pointer counting references of the encapsulated data. Usually in Rust when when passing a variable to a new scope (e.g. as parameter to a function) one has to either give ownership to the new owner (e.g. the new function), which would make the variable inaccessible after the function or clone/copy the variable, which would create a separate instance of the same data, which can become very memory intensive with large data types. However, with an `Arc` pointer you can enclose data inside of it and very efficiently pass this data around by reference. This way instead of cloning the entire dataset and doubling the memory of that data, one can just clone the pointer and pass the pointer to the function, which only clones the address, which still points to the same data, even across threads. See an example of using `Arc` in listing 5.1.

```
1  // bring Arc into scope
2  use std::sync::Arc;
3
4  // placeholder struct which takes up a lot of memory
5  let data = VeryLargeStruct::new();
6
7  // create a Arc pointer to data
8  let data_arc = Arc::new(data);
9
10 // clone pointer to use in a new tokio runtime
11 let data_clone = data_arc.clone();
12 tokio::spawn(async move {
13   // processing data on a new thread
14   println!("{}", data_clone);
15 });
16
17 // data can still be used
```

```
18 println!("{}", data_arc);
```

Listing 5.1: Using Arc Smart Pointer

If I were to skip the step of encapsulating `data` in an `Arc` (line 8) and `VeryLargeStruct` is unlikely to implement the `Copy` Trait, we would need to create a second copy of `data` using a lot of memory. If `VeryLargeStruct` would have implemented the `Copy` Trait and one would skip line 8 and 11 and directly use the `data` inside the Tokio thread, `data` would have been moved into the thread and be inaccessible in line 18, but Rust's compiler would not compile and point this very issues out.

The `Arc` is highly integral to the design pattern in Rust called Interior Mutability, which describes the possibility to mutate data without having mutable references to this data. This can be achieved with the previously mentioned smart pointers and Mutexes. There are several types of smart pointers and Mutexes in Rust, but I will only be using the aforementioned `Arc` and `RwLock`.

All we got to do is encapsulate the data, we want to access and mutate on different threads, inside an `RwLock` which we encapsulate again inside an `Arc`. The `Arc` pointer allows us to pass references of our data around as want and it also doesn't change the way we are able to use the encapsulated data. For example having an unsigned integer with bit size 64 `u64` and the same inside an `Arc Arc<u64>` is completely equal to the coder, with the only expectation being the underlying procedure when cloning these variables and of cause being able to pass a reference to the data to another thread. The `RwLock` is what enables us to mutate the data even though we are still working with immutable references. See an example in listing 5.2.

```
1  // bring Arc and RwLock into scope
2  use std::sync::Arc;
3  use tokio::sync::RwLock;
4
5  // instantiate immutable data
6  let data = 10;
7
8  // create the special construct for Interior Mutability
9  let data = Arc::new(RwLock::new(data));
10
11 // get a read lock
12 let data_read = data.read().await;
13 println!("{}", data_read); // prints 10
14
15 // get a second read lock
16 let data_read2 = data.read().await;
17 println!("{}", data_read2); // prints 10
18
19 // unlock both read locks to allow following write lock
20 drop(data_read);
21 drop(data_read2);
22
23 // get a write lock
24 let mut data_write = data.write().await;
25
```

```
26 // mutate data
27 *data_write *= 2;
28 println!("{}", data_read2); // prints 20
29
30 // going out of scope will also unlock the lock (equals to the drop call
       from above)
```

Listing 5.2: Interior Mutability in Action

If we had skipped the `drop()` in line 20 and 21, line 24 would have started block the execution until the two previous read locks would have been unlocked, however since that would not have been possible this linear example, this would have resulted in a dead lock. A dead lock is a unresolvable blocking of execution.

5.2.4 Communication between Code

To process the memory cache for low-latency streaming, I was using event emitter callbacks in the form of `.on(event, callback)` to define the procedure to execute on a specific event and `.emit(event, ...data)` to throw a specific event making the callback run where it was defined.

The Rust programming language does have functional programming and technically is able replicate this event emitting as well. However, since Quinn and Hyper are both asynchronous this made this problematic, because asynchronous closures are strictly speaking not allowed in Rust. There are workarounds to be able to use asynchronous closures, however, I was unable to get them to work.

I needed an alternative and was able to find one directly in the Tokio crate. Tokio has various forms of `Channels`. Similar to Channels in Go, Channels allow direct communication between threads.

For this use case I used `broadcast` channels, which allow having multiple producers and multiple consumers. A channel consists of a transceiver (the producer) and a receiver (the consumer) and in the case of a `broadcast` both can be cloned as many times one needs.

5.2.5 Stream Messages

Finally, a minor issues arose from Rust's strict type system. In Go and especially TypeScript and JavaScript there is a catch-all `Any` type, which as the name implies can be whatever data one wants.

However, Rust does not have that luxury, for my use case at least. Before I had the `Message` object (listing 4.3) with the field `value` of type `Any` which allowed me to send whatever, I wanted, numbers, string, booleans, even entire objects, anything went.

Rust does technically have the option to work with a special `Any` type, however, on first glance this looked very complicated and from what I could gather this wouldn't've been compatible with the JSON parsing crate `serde` and `serde_json` anyways.

This is why I changed the `value` type to `String` which is the safest compromise allowing me to send anything using these Messages that can be converted to String and so far

there hasn't been anything that wasn't.

6 Evaluation

In this chapter will evaluate the implementation of the WebTransport media streamer and present the benchmarks along with their results and observations.

6.1 Test Environment

I ran the benchmarks on the Work Laptop shown in table 5.1 in a local Docker Container. I ran two benchmarks in total to evaluate the performance of the implementation.

6.1.1 Startup Time Benchmark

The first benchmark is testing the Startup Time, the time it takes for a media stream to start playback, comparing WebTransport and traditional HTTP requests.

There are many parameters to look at when streaming a video, the quality of the video in form of the bitrate, the segment duration, the initial buffer length target and the condition of the network connection.

All those factors play a part in how fast a video can start playing. However, it would take a lot of time to benchmark all parameters and in the end they would all roughly affect the startup time linearly. A lower bitrate allows sending more segments faster, while a higher bitrate results is the opposite. Shorter segments would be smaller to have playable data more quickly and most likely benefit WebTransport over HTTP requests by not needing more requests, again exactly the opposite would be the case for larger segments. Along the same lines, having a shorter buffer target requires less data to start playing and a larger target would take longer. Finally, the network condition naturally has a impact on the transmission of the stream, more throughput equals faster transmission.

To simplify the testing of Startup Time, I limited the benchmark to just test how the network condition, in form of a synthetic bandwidth limit using `tc`, affects the Startup Time. For every other parameter I decided on fixed values.

- Quality: `720p` ($\approx$ `2,000 kbps`)
- Segment Duration: `1,6s`
- Buffer Target: `1,6s`

I have decided on $720p$ as quality to test, because it has a bitrate of roughly $2,000 kbps$ which I see as a good balance between quality and data usages. I settled on $1,6s$ as segment duration because it is the shortest one I had available and generally it is best to

keep it on the shorter side to improve latency. As initial buffer target I chose the length of exactly one segment which is the shortest that would have been possible, since only full segments can be played back in a non-low-latency scenario.

Finally, I chose bandwidth limits that are in relation to the video bitrate, being $500kbps$ (quarter), $1,000kbps$ (half), $1,500kbps$ (three-quarters), $2,000kbps$ (full), $3,000kbps$ (one half over) and $4,000kbps$ (double).

The Startup Time benchmark's order of operation is as follows. Every iteration of the benchmark opens a new browser tab and initializes my MSE Player. Since this is a benchmark the first act of the MSE Player is to send a report to the HTTP server with the current timestamp as start time. Then the MSE Player starts its normal operation until the desired buffer has been reached, which would normally start the playback of the video. However, since this is a benchmark, the MSE Player again sends a report to the HTTP server with another timestamp, signally the end of this iteration. After the report has been sent, the MSE Player will close the tab and the next run will start.

For every bandwidth limit the benchmark will run 50 times and in the end take the arithmetic mean of those 50 values to get the result displayed in table 6.1 and table 6.2.

With two protocols to benchmark, six bandwidth limits to evaluate and 50 samples for each, there is a total 600 runs which make up the entire benchmark. The results will be shown later in section 6.2.

6.1.2 Server-side Vs. Client-side ETP Benchmark

The second benchmark would've been the comparison of server-side ETP using Quinn and client-side ETP using Dash.js. However, since I wasn't able to come up with a working ETP calculation with the tools provided by the Quinn crate, as detailed in section 5.2.1, I wasn't able to actually run this benchmark for the Rust implementation.

The ETP benchmark would have been similar to the Startup Time benchmark, starting by opening a new browser tab. The MSE Player would playback the video as normal. However, since this would've been a ETP benchmark the MSE Player would additionally receive ETP value reports from the WebTransport server and then send them to the HTTP server to save them.

The next run would've done the same but using Dash.js instead. Using Dash I would've used the event `fragmentLoadingCompleted` by Dash, which is thrown whenever a segment has finished downloading from the server. Then I use Dash's function `dash.getAverageThroughput(mediaType)` to get the currently estimated throughput and again send it to the HTTP server for later plotting.

Both iterations were to run for $60s$, collect the ETP values and finally would have been plotted in a line graph and displayed on the website. Since this benchmark was not possible using Quinn section 6.2 will show the previous results gathered using webtransport-go.

6.2 Performance Measurements

In this section I will present the results of the previously mentioned benchmarks.

Bandwidth [kbps]	Bitrate [kbps]	Segment Duration [s]	Buffer Target [s]	WebTransport [ms]	HTTP/1.1 [ms]	WT faster by	Samples
500	2,048	1,6	1,6	6,067	8,261	26,56%	50
1,000	2,048	1,6	1,6	2,971	3,948	24,75%	50
1,500	2,048	1,6	1,6	1,966	2,709	27,43%	50
2,000	2,048	1,6	1,6	1,503	2,095	28,26%	50
3,000	2,048	1,6	1,6	1,004	1,400	28,29%	50
4,000	2,048	1,6	1,6	777	1,064	26,97%	50
						27,04%	average

Table 6.1: Startup Time Benchmark using Go (WebTransport) and Express.js (HTTP/1.1)

Bandwidth [kbps]	Bitrate [kbps]	Segment Duration [s]	Buffer Target [s]	WebTransport [ms]	HTTP/1.1 [ms]	WT faster by	Samples
500	2,048	1,6	1,6	4,903	8,211	40,29%	50
1,000	2,048	1,6	1,6	2,595	3,935	34,05%	50
1,500	2,048	1,6	1,6	1,768	2,613	32,34%	50
2,000	2,048	1,6	1,6	1,375	2,081	33,93%	50
3,000	2,048	1,6	1,6	973	1,369	28,93%	50
4,000	2,048	1,6	1,6	767	1,056	27,37%	50
						32,82%	average

Table 6.2: Startup Time Benchmark using Rust Quinn (WebTransport) and hyper (HTTP/1.1)

Startup Time Benchmark

In table 6.1 the previous results using Go for WebTransport and Express.js for HTTP/1.1 are shown. On average WebTransport shows an improvement of Startup Time of about 27% over HTTP/1.1.

Looking at table 6.2, which shows the same benchmark as table 6.1 but now using Rust with Quinn for WebTransport and Hyper for HTTP/1.1, it can be observed that HTTP/1.1 has stayed exactly the same within margin. However, WebTransport has again improved. Interesting to note here is that previously in table 6.1 the improvement was roughly the same across all bandwidth limits, using Rust, though, this is not the case. Rust is showing a much greater improvement at lower bandwidth limits and as the limit increases the improvement decreases. However, on average WebTransport using Rust still increased the improvement of Startup time to just under 33% in comparison to HTTP.

WebTransport's faster Startup time can accredited to QUIC's much faster opening handshake by combining the protocol and TLS handshake into one and allowing the

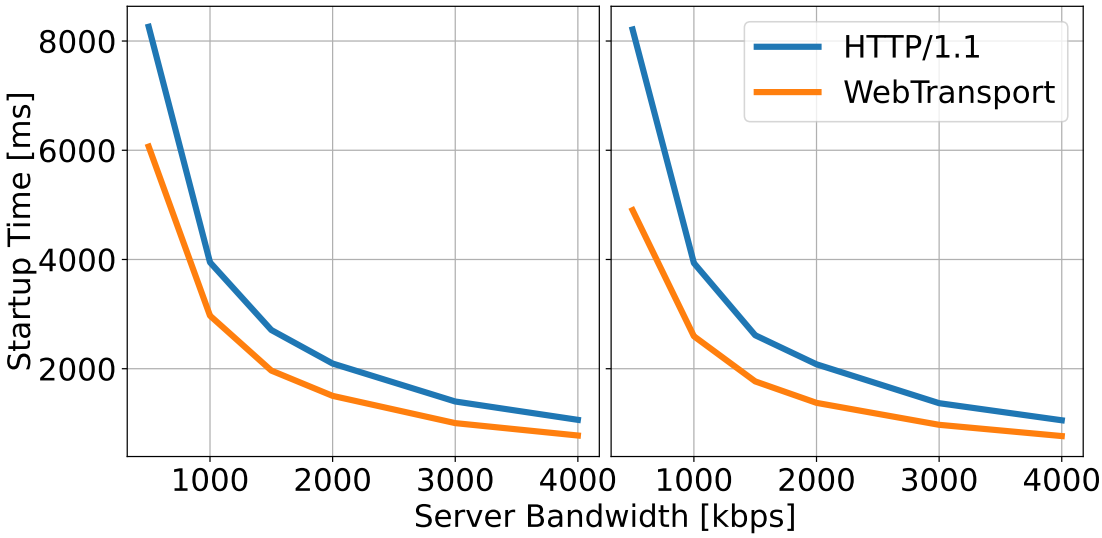

Figure 6.1: Startup Time Benchmark Plots, left: Express (HTTP/1.1) and webtransport-go (WebTransport), right: Hyper (HTTP/1.1) and Quinn (WebTransport)

server to send data on its own accord, rather than having to a TCP handshake, followed by a TLS handshake, followed by a HTTP GET request to get the first segment in case of HTTP. This can be summarized to HTTPS needing a 3-RTT instead of a 1.5-RTT in QUIC's case.

6.2.1 Server-side Vs. Client-side ETP Benchmark

Since ETP calculation wasn't possible using Quinn fig. 6.2 and fig. 6.3 show the previous results compiled using webtransport-go. In the scenario of VoD streaming (fig. 6.2) both WebTransport and Dash are both showing excellent estimations. However, both are showing advantages and disadvantages.

Dash's estimations are more accurate but only once the moving window completely encloses values of the same expected limit. On the other hand, WebTransport is better at detecting changes in throughput more quickly, but are not quite as accurate as Dash once the available bandwidth greatly exceeds the required throughput in accordance of the video bitrate.

Out of curiosity I re-ran the benchmark with a altered version of Dash.js where I changed the window size of the bandwidth estimation algorithm from the default of 4 to just 1. This way it can be observed that Dash.js completely negates the one advantage WebTransport had over it by more quickly detecting changes in throughput, as shown in fig. 6.4.

Finally, I ran this benchmark one more time, this time using a low-latency stream. This is where WebTransport shines. In fig. 6.3 it can be seen that Dash greatly overestimates the throughput, which is expected by the implementors of the LoL+ algorithm Dash is

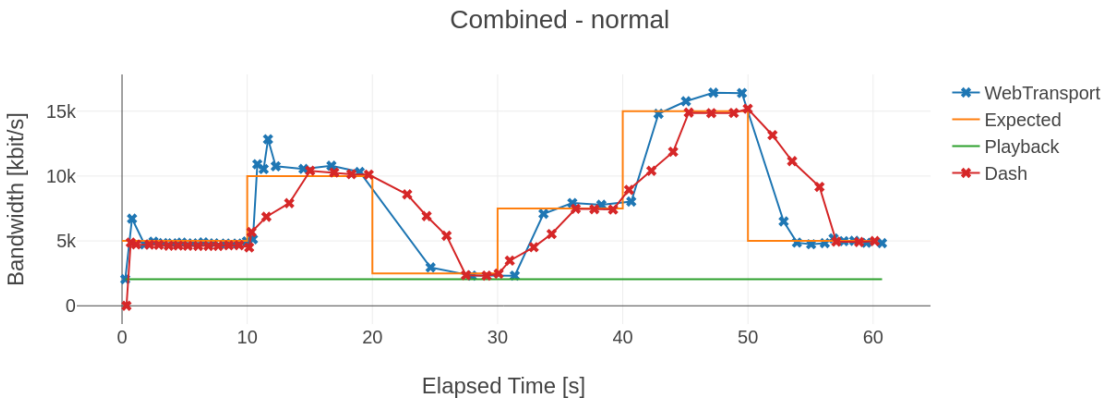

Figure 6.2: ETP - webtransport-go Vs. Dash.js using VoD

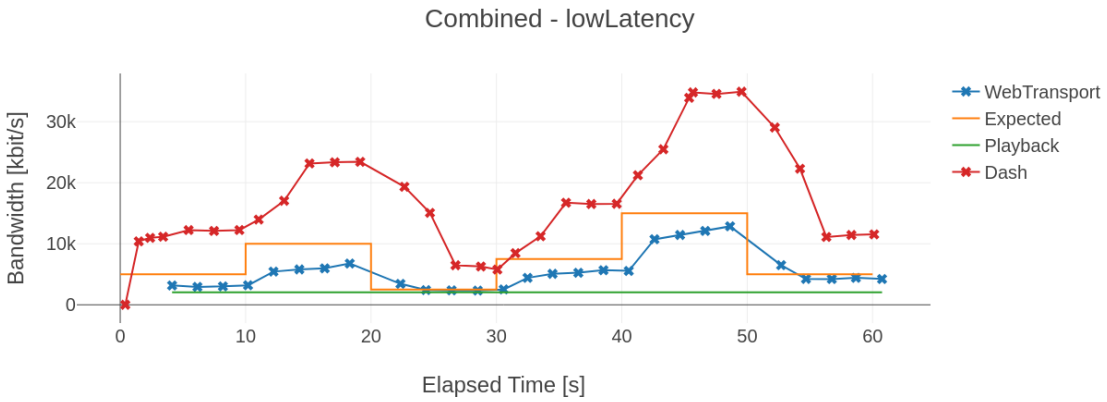

Figure 6.3: ETP - webtransport-go Vs. Dash.js using Low-Latency

using, [54]. In contrast to Dash's overestimation, WebTransport slightly underestimates the throughput.

In terms of ABR algorithms underestimation is the lesser evil and is desired over overestimation. If the throughput is overestimated, the ABR algorithm will work with a greater ETP value assumption and will recommend a bitrate and segment frequency which the current network conditions cannot sustain. Meanwhile, an underestimated ETP value will allow the ABR algorithm to recommend a bitrate and segment frequency which the network conditions can easily maintain. This may not use the network to its full capabilities, but will not lead to frequent stalling as it would when working with overestimation.

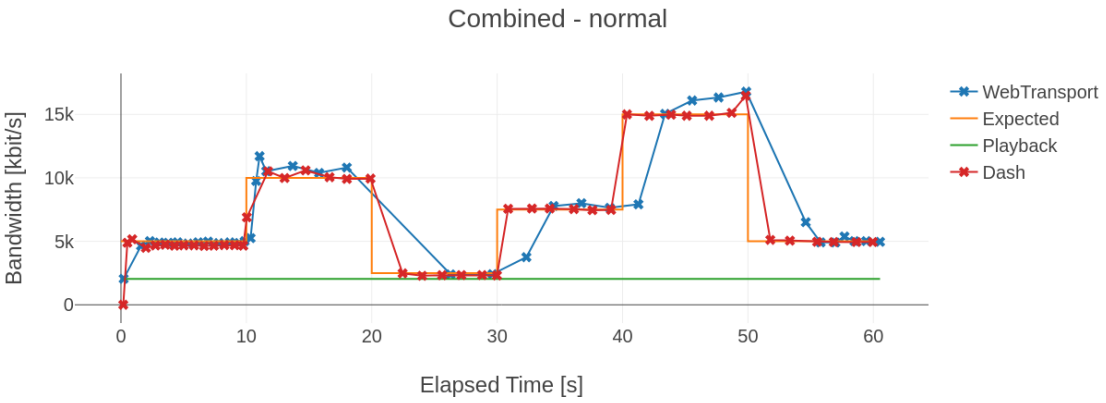

Figure 6.4: ETP - webtransport-go Vs. Dash.js (window size 1) using VoD

7 Conclusion

To come to a conclusion I will begin this chapter with a summary of this thesis, followed by a dissemination of where media streaming using WebTransport fits into todays industry. Finally, I'll recap the problems I came across, solved and which remain unsolved and give a outlook on where this research can lead and what it can be used for.

7.1 Summary

At the beginning of this thesis I had a running WebTransport media streaming setup using webtransport-go from the Go programming language as WebTransport server and Express.js using TypeScript as HTTP/1.1 server and host for the Client and the task was to transfer this implementation to the Rust programming language, a coding language I had no previous experience with, with the goal of improving the performance of the WebTransport server and overall show that WebTransport media streaming is the next big thing after using HTTP for the past 10 to 15 years.

I started by learning the basics of Rust by going through "The Rust Book" [35] as well as doing practical training using "Rust by Example" [36], followed by research on which crates to use to implement the WebTransport and HTTP server. Most of the crates I found were just raw QUIC implementations and would have required additional work extending them to support the WebTransport standard, however, I was made aware that Luke Curely and Mike English have already done this for the Quinn [40] crate as part of a project for the Media over QUIC IETF working group [30]. On the other side for HTTP I settled on the most popular crate according to download numbers on crates.io, Hyper [39].

With the right tools and the knowledge to use them I started with the actual work of converting the Go and TypeScript servers over to pure Rust. Along the way I came across a few hurdles which I had to overcome.

The biggest one was Quinn missing a working method of estimated the server-side throughput. I tried numerous way to come up with a solution of estimating the throughput but in the end all failed and came to the conclusion that right now it is not possible to estimate the throughput using Quinn in its current state. I am assuming that the culprit of this issue is the experimental implementation of the BBR congestion control algorithm. The documentation of the BBR struct mentioned that their implementation is based on Google's Quiche implementation but is currently experimental and it seems there are still bugs in Quinn's BBR implementation.

Another issue I happened upon was enabling encryption on the Hyper server as well as serving static files to host the Client website. Later on I found a complementary crate to

Hyper which would have enabled HTTPS with very little changes to the server, however, that would have still not addressed the problem of hosting static files. To overcome this problem I added an NGINX HTTPS reverse proxy between client and Hyper server, which hosts the static files of the Client implementation and redirects all HTTP requests intended for the Hyper server. This offloads the cheap task of serving static files to the NGINX server and leaves Hyper more resources to host the more important media segments. Additionally, NGINX is highly efficient at load balancing a large number of client, making this setup highly scalable if deployed on a public server.

Once the servers had been converted to Rust, I had to make minor changes to the Client webpage to make benchmarking a little easier and then I just had to run these benchmarks and collect the results.

As discussed in section 6.2, WebTransport in general shows noticeable improvements over using HTTP for media streaming by making videos start to play about 30% faster. The Rust server in specific is performing slightly better than the Go server.

7.2 Dissemination

Media streaming is the favorite past time activity of many people and it shows in the internet traffic, more than half of which is taken up by media streaming [1]. This leads to the search of possibilities to make media streaming as efficient as possible.

Using QUIC and WebTransport is one of these possibilities by streamlining the opening handshakes and reducing overhead by allowing the server to control the distribution of media segments instead of requiring the client to periodically making numerous HTTP requests.

This has lead to the IETF working group Media over QUIC [30], which member's are eagerly working on a standard for streaming media over QUIC and WebTransport.

7.3 Outlook

Ultimately, WebTransport and QUIC show high promise for a large scope of application. In this thesis I have looked at media streaming as one type of application and in this case WebTransport is showing a sizeable improvement over traditional HTTP media streaming. This work can be further expanded by adapting to the recently released Media over QUIC Transport (MOQT) standard [34] and integration into existing media players. For example one possibility would be the Dash.js player which I have referenced multiple times in this thesis.

Important to note regarding Dash, though, is that the whole schtick of Dash is the manifest file, which Dash is using to acquire all information for the specific media stream and with WebTransport the client wouldn't need to do this anymore, making the current Dash.js player superfluous. One way would be to split the Dash player up into a server- and client-side application. The client-side would be responsible for establishing a WebTransport connection and maintaining playback using MSE and the server-side would be

taking care of handling the manifest file and relaying the media segments to the client using the existing WebTransport connection.

Another possibility would be a replacement to the WebRTC API offering a more open and customizable alternative, allowing a multitude of low-latency applications, as recently outlined in the blog post "Replacing WebRTC" by Luke Curely [55].

In conclusion, QUIC and WebTransport are on the verge of revolutionizing communication in the web, especially low-latency communication by using a polished extension of UDP, turning it from an unreliable and insecure transport protocol to a reliable and secure one, while still maintaining low-latency and offering a quicker opening handshake than HTTPS and giving the server more control.

List of Acronyms

MoQ	Media over QUIC
MOQT	Media over QUIC Transport
HAS	HTTP adaptive streaming
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
HoL	Head-of-Line Blocking
MSE	MediaSourceExtension
HLS	HTTP Live Streaming (Apple)
DASH	Dynamic Adaptive Streaming over HTTP
ABR	Adaptive Bitrate Algorithm
MPD	Media Presentation Description
TLS	Transport Layer Security
RTT	Round-Trip-Time
RTP	Real-Time Transport Protocol
RTMP	Real-Time Messaging Protocol
ALPN	Application-Layer Protocol Negotiation
LL	Low-Latency
ETP	estimated throughput
JSON	JavaScript Object Notation
ECDSA	Elliptic Curse Digital Signature Algorithm
tc	Traffic Control
VoD	Video-on-Demand

Bibliography

[1] T. Bianchi, "Media internet traffic usage."

[2] quic go, "Github - webtransport-go."

[3] D. W. Mikeal Rogers, "Npm - express.js."

[4] FFmpeg, "Ffmpeg home page."

[5] M. . M. Contributors, "Media source api."

[6] R. Pantos and W. May, "HTTP Live Streaming." RFC 8216, Aug. 2017.

[7] Apple, "Http live streaming apple developer."

[8] "Dash.js iso/iec 23009-1:2012 release."

[9] Dash-Industry-Forum, "Dash.js github."

[10] W. Eddy, "Transmission Control Protocol (TCP)." RFC 9293, Aug. 2022.

[11] C. Westphal, S. Lederer, C. Mueller, A. Detti, D. Corujo, J. Wang, M.-J. Montpetit, N. Murray, C. Timmerer, D. Posch, A. Azgin, and W. S. LIU, "Adaptive Video Streaming over Information-Centric Networking (ICN)." RFC 7933, Aug. 2016.

[12] E. Rescorla, "HTTP Over TLS." RFC 2818, May 2000.

[13] W3Techs, "Usage statistics of default protocol https for websites."

[14] M. Belshe and R. Peon, "SPDY Protocol," Internet-Draft draft-mbelshe-httpbis-spdy-00, Internet Engineering Task Force, Feb. 2012. Work in Progress.

[15] M. Belshe, R. Peon, and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)." RFC 7540, May 2015.

[16] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport." RFC 9000, May 2021.

[17] M. Bishop, "HTTP/3." RFC 9114, June 2022.

[18] "User Datagram Protocol." RFC 768, Aug. 1980.

[19] V. Vasiliev, "Webtransport."

[20] A. F. et al., "Webtransport over http/2."

[21] M. . M. Contributors, "Webtransport browser support."

[22] M. . M. Contributors, "Streams api."

[23] J. Posnick, "Using webtransprot."

[24] H. Schulzrinne, S. L. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications." RFC 3550, July 2003.

[25] Adobe, "Adobe flash player end-of-life."

[26] W. Traci Ruether, "2021 video streaming latency report by wowza."

[27] M. . M. Contributors, "Webrtc api."

[28] M. . M. Contributors, "Introduction to the real-time transport protocol (rtp)."

[29] W. Traci Ruether, "History of streaming media [infographic] by wowza."

[30] T. H. Alan Frindell, "Media over quic (moq) ietf working group."

[31] M. E. Luke Curley, "Media over quic (moq-rs) by luke curley and mike english."

[32] M. W. Group, "Moq meeting minutes 118."

[33] M. . M. Contributors, "Webcodecs api."

[34] L. Curley, K. Pugin, S. Nandakumar, and V. Vasiliev, "Media over QUIC Transport," Internet-Draft draft-ietf-moq-transport-01, Internet Engineering Task Force, Oct. 2023. Work in Progress.

[35] S. Klabnik and w. c. f. t. R. C. Carol Nichols, "The rust programming language."

[36] R.-L. Team and Community, "Rust by example."

[37] S. Leffler, "Rust's type system is turing-complete."

[38] R.-L. Team, "crates.io - the rust community's crate registry."

[39] h. Sean McArthur, "crates.io - hyper."

[40] B. S. Dirkjan Ochtman, "crates.io - quinn."

[41] A. G. cloudflare, "crates.io - quiche."

[42] Mozilla, "Github - neqo."

[43] L. Curely, "crates.io - webtransport-quinn."

[44] F. Li, J. W. Chung, X. Jiang, and M. Claypool, "Tcp cubic versus bbr on the highway," in *Passive and Active Measurement* (R. Beverly, G. Smaragdakis, and A. Feldmann, eds.), (Cham), pp. 269–280, Springer International Publishing, 2018.

[45] P. Hurtig, H. Haile, K.-J. Grinnemo, A. Brunstrom, E. Atxutegi, F. Liberal, and Å. Arvidsson, "Impact of tcp bbr on cubic traffic: A mixed workload evaluation," in *2018 30th International Teletraffic Congress (ITC 30)*, vol. 01, pp. 218–226, 2018.

[46] A. R. Carl Lerche and tokio rs, "crates.io tokio."

[47] D. O. Joe Birr-Pixton and rustls publishers, "crates.io rustls."

[48] S. Friedl, A. Popov, A. Langley, and S. Emile, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension." RFC 7301, July 2014.

[49] M. . M. Contributors, "Javascript crypto: randomuuid() method."

[50] B. Hubert, "tc(8) linux manual page."

[51] Akamai, "Ffmpeg command line builder (r0.3h)."

[52] R. Marx, L. Niccolini, M. Seemann, and L. Pardue, "Main logging schema for qlog," Internet-Draft draft-ietf-quic-qlog-main-schema-07, Internet Engineering Task Force, Oct. 2023. Work in Progress.

[53] G. U. Ralith, "Github issues - quinn qlog."

[54] A. Bentaleb, M. N. Akcay, M. Lim, A. C. Begen, and R. Zimmermann, "Catching the moment with lol$^+$ in twitch-like low-latency live streaming platforms," *IEEE Transactions on Multimedia*, vol. 24, pp. 2300–2314, 2022.

[55] L. Curely, "Replacing webrtc."

Annex

```rust
use std::io::Read;

#[tokio::main]
async fn main() {
  tokio::spawn(async move { serve_fingerprint().await });

  let addr = "127.0.0.1:5000".parse().unwrap();

  // get TLS certificate and key
  let (key, cert) = get_tls_cert();

  let server_config = rustls::ServerConfig::builder()
    .with_safe_default_cipher_suites()
    .with_safe_default_kx_groups()
    .with_protocol_versions(&[&rustls::version::TLS13])
    .unwrap();
  let mut server_config = server_config
    .with_no_client_auth()
    .with_single_cert(cert, key)
    .unwrap();
  server_config.max_early_data_size = u32::MAX;
  server_config.alpn_protocols = vec![webtransport_quinn::ALPN.to_vec()];

  let server_config = std::sync::Arc::new(server_config);

  // create QUIC serer config using TLS config
  let mut config = quinn::ServerConfig::with_crypto(server_config);

  // enable BBR
  let mut transport_config = quinn::TransportConfig::default();
  transport_config.congestion_controller_factory(std::sync::Arc::new(
    quinn::congestion::BbrConfig::default(),
  ));
  transport_config.mtu_discovery_config(None); // disable MTU discovery
  let transport_config = std::sync::Arc::new(transport_config);

  config.transport_config(transport_config);

  // create server endpoint using config and address
  let endpoint = quinn::Endpoint::server(config, addr).unwrap();
  println!("Server listen on https://localhost:5000");

  // start server loop
  loop {
    // accept incoming connections
```

```
46      let conn = endpoint.accept().await.unwrap();
47
48      // wait for handshake to complete
49      let conn = conn.await.unwrap();
50
51      // upgrade QUIC connection to WebTransport
52      let request = webtransport_quinn::accept(conn).await.unwrap();
53
54      // complete WebTransport handshake
55      let session = request.ok().await.unwrap();
56
57      // spawn new thread to process session
58      tokio::spawn(async move {
59        // handle incoming datagrams
60        let s = session.clone();
61        let datagrams = tokio::spawn(async move {
62          loop {
63            let data = s.read_datagram().await.unwrap();
64
65            println!(
66              "Received Datagram: {}",
67              String::from_utf8(data.to_vec()).unwrap()
68            );
69
70            s.send_datagram(data).await.unwrap();
71          }
72        });
73
74        // handle incoming unidirectional streams
75        let s = session.clone();
76        let uni_stream = tokio::spawn(async move {
77          loop {
78            let mut stream = s.accept_uni().await.unwrap();
79
80            let data = stream.read_to_end(64 * 1024).await.unwrap();
81
82            println!(
83              "Received Data on Unidirectional Stream: {}",
84              String::from_utf8(data.to_vec()).unwrap()
85            );
86
87            let mut stream = s.open_uni().await.unwrap();
88
89            stream.write(&data).await.unwrap();
90          }
91        });
92
93        // handle incoming bidirectional streams
94        let s = session.clone();
95        let bidi_stream = tokio::spawn(async move {
96          loop {
97            let (mut tx, mut rx) = s.accept_bi().await.unwrap();
98
```

```
99            let data = rx.read_to_end(64 * 1024).await.unwrap();
100
101           println!(
102             "Received Data on Bidirectional Stream: {}",
103             String::from_utf8(data.to_vec()).unwrap()
104           );
105
106           tx.write(&data).await.unwrap();
107         }
108       });
109       tokio::try_join!(datagrams, uni_stream, bidi_stream).unwrap();
110     });
111   }
112 }
```

Listing 1: WebTransport Echo Server using Quinn and webtransport-quinn (Rust)

```
1  const button_data = document.getElementById("send_btn_data");
2  const button_uni = document.getElementById("send_btn_uni");
3  const button_bidi = document.getElementById("send_btn_bidi");
4  const input = document.getElementById("input");
5  const output = document.getElementById("output");
6
7  const encoder = new TextEncoder();
8  const decoder = new TextDecoder("utf-8");
9
10 async function main() {
11   // fetch WebTransport options, notably the cert fingerprint
12   const options = await get_options();
13
14   // establish a new WebTransport connection
15   const wt = new WebTransport("https://localhost:5000/test", options);
16
17   // optionally handle connection closure
18   wt.closed.then(() => {
19     console.log("WebTransport closed gracefully");
20   }).catch((error) => {
21     console.error("WebTransport closed with error: ", error);
22   });
23
24   // await the connection being established
25   await wt.ready;
26   console.log("WebTransport connection established");
27
28   button_data.onclick = async () => {
29     const payload = encoder.encode(input.value);
30
31     const writer = wt.datagrams.writable.getWriter();
32     const reader = wt.datagrams.readable.getReader();
33
34     writer.write(payload);
35
36     while ( true ) {
37       const { value, done } = await reader.read();
```

```
38        if ( done ) break;
39
40        const res = decoder.decode(value);
41        output.innerHTML += `Datagram: ${res}\n`;
42      }
43          input.value = "";
44    }
45
46    button_uni.onclick = async () => {
47      const payload = encoder.encode(input.value);
48
49      const stream = await wt.createUnidirectionalStream();
50
51      const writer = stream.writable.getWriter();
52
53      writer.write(payload);
54
55      const reader = wt.incomingUnidirectionalStreams.getReader();
56      while ( true ) {
57        const { value, done } = await reader.read();
58        if ( done ) break;
59
60        const rdr = value.getReader();
61        while ( true ) {
62          const { value, done } = await rdr.read();
63          if ( done ) break;
64
65          const res = decoder.decode(value);
66          output.innerHTML += `Uni Stream: ${res}\n`;
67        }
68      }
69          input.value = "";
70    }
71
72    button_bidi.onclick = async () => {
73      const payload = encoder.encode(input.value);
74
75      const stream = await wt.createBidirectionalStream();
76
77      const reader = stream.readable.getReader();
78      const writer = stream.writable.getWriter();
79
80      writer.write(payload);
81
82      while ( true ) {
83        const { value, done } = await reader.read();
84        if ( done ) break;
85
86        const rdr = value.getReader();
87        while ( true ) {
88          const { value, done } = await rdr.read();
89          if ( done ) break;
90
```

```
91          const res = decoder.decode(value);
92          output.innerHTML += `Bidi Stream: ${res}\n`;
93        }
94      }
95          input.value = "";
96    }
97  }
```

Listing 2: WebTransport Echo Client (JavaScript)

```
1  use hyper::body::Body;
2
3  #[tokio::main]
4  async fn main() {
5    let addr = "127.0.0.1:5001".parse::<std::net::SocketAddr>().unwrap();
6
7      let endpoint = tokio::net::TcpListener::bind(addr).await.unwrap();
8
9      loop {
10         let (stream, _) = endpoint.accept().await.unwrap();
11
12         let io = hyper_util::rt::TokioIo::new(stream);
13
14         tokio::spawn(async move {
15             hyper::server::conn::http1::Builder::new()
16                 .serve_connection(io, hyper::service::service_fn(handle))
17                 .await
18                 .unwrap();
19         });
20     }
21 }
22
23 async fn handle(
24     _req: hyper::Request<hyper::body::Incoming>,
25 ) -> Result<hyper::Response<http_body_util::Full<bytes::Bytes>>, hyper::
       Error> {
26     let mut req = req;
27
28     match (req.method(), req.uri().path()) {
29         (&hyper::Method::POST, "/echo") => {
30             let mut body = Vec::new();
31
32             // read incoming body by chunks
33             while let Some(chunk) = futures::future::poll_fn(|cx| {
34                 let c = std::pin::Pin::new(req.body_mut());
35                 c.poll_frame(cx)
36             })
37             .await
38             {
39                 let chunk = chunk.unwrap();
40                 let chunk = chunk.data_ref().unwrap();
41
42                 // append chunk to body
43                 body.append(&mut chunk.to_vec());
```

```
            }

            // send body back to client
            return Ok(hyper::Response::new(http_body_util::Full::new(
                bytes::Bytes::from(body),
            )));
        }
        _ => println!("unsupported -> handle Error.."),
    }
}
```

Listing 3: HTTP Echo Server using Hyper (Rust)

```
const button_http = document.getElementById("send_btn_http");
const input = document.getElementById("input");
const output = document.getElementById("output");

button_http.onclick = async () => {
  const res = await fetch("/api/echo", {
    method: "POST",
    body: input.value,
  });

  const data = await res.text();
  output.innerHTML += `HTTP: ${data}\n`;
  input.value = "";
}
```

Listing 4: HTTP Echo Client (JavaScript)

```
{
  "speed": 5_000,
  "profile": "NONE",
  "trajectory": []
}
```

Listing 5: Static Bandwidth Limit of 5,000 kbps

```
{
  "speed": 12345,
  "profile": "custom trajectory example",
  "trajectory": [
    {
      "speed": 10_000,
      "duration": 10_000,
      "latency": 50
    },
    {
      "speed": 5_000,
      "duration": 15_000,
      "latency": 50
    },
    {
      "speed": 15_000,
```

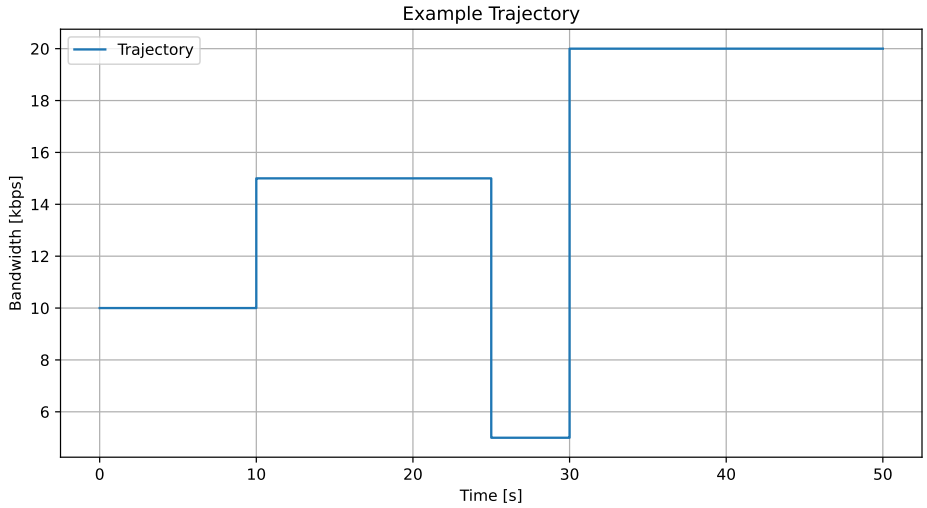

Figure .1: Trajectory listing 6 visualized

```
      "duration": 5_000,
      "latency": 50
    },
    {
      "speed": 5_000,
      "duration": 20_000,
      "latency": 50
    }
  ]
}
```

Listing 6: Trajectory Bandwidth Limit