Evaluating Performance of HTTP/3 for Video Streaming: A Comparative Study with Previous Versions of HTTP

Thomas Daniel Galligan

Final Year Project
BSc in Computer Science

Supervisor: Dr. Cormac Sreenan
Second Reader: Dr. Ahmed Zahran

Department of Computer Science
University College Cork

April 2023

Abstract

Standardized in July of 2022, HTTP/3 aims to improve over previous versions of HTTP. The project aims to investigate the protocol's performance concerning video streaming and compare it with earlier versions of HTTP. To investigate this, tests run from multiple clients were tested against a web server that supports HTTP/1.1, HTTP/2, and HTTP/3. These tests were conducted under different network environments that could be encountered in the real world while downloading files and streaming video over HTTP. Network variables tested against were latency, packet loss rates, and egress bandwidth from the data sender. The results from these tests showed a slight decrease in overall throughput via HTTP/3 compared to the other protocols when placed under the same network conditions. HTTP/3 performed significantly better under high packet loss environments, but worse in high latency environments. Under differing bandwidth conditions, HTTP/3 performed worse than HTTP/1.1 and HTTP/2, but the difference was negligible. Under high latency conditions, HTTP/3 performed poorly and was the worst-performing protocol of the three. The results suggest that HTTP/3 has noticeably better video streaming performance than the other two. That performance gain is offset, however when the latency is sufficiently high.

Declaration of Originality

In signing this declaration, you are conforming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award.

I hereby declare that:

- this is all my own work unless clearly indicated otherwise, with full and proper accreditation;
- with respect to my own work: none of it has been submitted at any education institution contributing in any way to an educational award;
- with respect to anothers' work: all text, diagrams, code, or ideas, whether verbatim, paraphrased, or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Signed: Thomas Daniel Galligan

Date: 24th April, 2023

Acknowledgements

I want to express my sincere gratitude to my project supervisor, Dr. Cormac Sreenan, for his guidance and support throughout this project. His expertise, encouragement, and insightful feedback have been instrumental in shaping my work.

I would also like to thank Dr. Jason Quinlan for introducing me to the QUIC transport protocol and sharing his knowledge and expertise in the field. His prior mentorship and support have been invaluable in helping me understand the intricacies of this complex protocol.

I want to extend my appreciation to Liam Crilly from NGINX for providing me with valuable insights into NGINX's QUIC and HTTP/3 implementation. His contributions have been constructive in deepening my understanding of this cutting-edge technology.

Contents

List of Figures

Listings

Chapter 1

Introduction

The introduction of video streaming in recent years [1] has revolutionized how we consume media, from entertainment to education and communication. With the increasing internet traffic for video content [2], the need for efficient video streaming technologies has also risen due to increasing bitrate requirements of video over time [2]. Hypertext Transfer Protocol (HTTP) (Hypertext Transfer Protocol) is widely used for video streaming. Hypertext Transfer Protocol Version 3 (HTTP/3) is the latest version of HTTP, designed to address previous versions' performance limitations, such as head-of-line blocking and connections only consisting of a single stream (further explained in Chapter 2).

In this project, the performance of HTTP/3 for video streaming will be evaluated using Dynamic Adaptive Streaming over HTTP (Dynamic Adaptive Streaming over HTTP (DASH)), a popular method for streaming video. This project will investigate the performance of HTTP/3, which will be compared with previous versions of the protocol, and investigate whether HTTP/3 improves video streaming performance.

1.1 Motivation

The performance of video streaming is critical for User Experience (UX). Bad Quality of Experience (QoE) often leads users to abandon video playback [3]. Therefore, it is essential to ensure that video QoE is sufficiently high to keep users engaged.

According to Akamai, a Content Delivery Network (Content Delivery Network (CDN)) provider, users tend to abandon videos after 2 seconds of initial load time. Each additional second of load time increases the video abandonment rate by 5.8% incrementally [3]. Netflix also quantifies the delay until video playback starts as a QoE metric [4]. From this, the performance of video streaming is vital for UX. If a new protocol improved the performance of video streaming to replace existing ones easily, it would benefit both users and service providers.

1.2 Objectives

The objectives of this project are to determine the following:

- Is HTTP/3 a viable alternative to previous versions of HTTP for video streaming?
- How does HTTP/3 compare to the previous versions of HTTP under different network conditions?
- Does the initial load time of video streamed over HTTP/3 improve under any conditions compared to previous versions of HTTP?

The first objective is to determine if HTTP/3 is a viable alternative to previous versions of HTTP for video streaming. If HTTP/3 can be used as an alternative to previous versions without significant performance regressions, it may be worth replacing previous HTTP versions for video streaming with the new version.

The second objective is to determine how HTTP/3 compares to previous versions of HTTP under different network conditions. This can be important for mobile users on 4G networks who may be subject to varying network conditions [5].

Finally, the third objective is to determine if the initial load time of streamed video over HTTP/3 improves under any conditions compared to previous versions of HTTP. The initial load time of video streaming is essential for QoE, as detailed by the Akamai study [3] and Netflix [4].

Chapter 2

Background

This chapter will discuss the background of video streaming, HTTP, its underlying transport protocols, and HTTP security. Different HTTP versions will be compared in this project and briefly discussed to get important context on the components.

2.1 Video Streaming

Video-on-demand streaming is the process of delivering video content over the internet in a continuous flow. It enables users to watch video content without downloading the entire video file before playback can start. Video streaming has gained immense popularity in recent years, with the rise of platforms such as YouTube and Netflix [1], among the most visited websites in the world [6].

2.1.1 Dynamic Adaptive Streaming over HTTP (DASH)

DASH, sometimes called MPEG-DASH [7], is a popular method for video streaming [8] that has gained widespread adoption in recent years. DASH enables the delivery of video content in small segments. These video segments can be encoded into different quality profiles, allowing the client to dynamically select the quality profile that best suits the current network conditions. This approach allows an algorithm to fetch video segments at whichever quality profile is best for the situation [9].

DASH content must be pre-encoded into segments and containerized in a format called m4s [7]. The m4s files are MP4-encoded binary files containing multimedia (can be video, audio, or audio and video together) data.

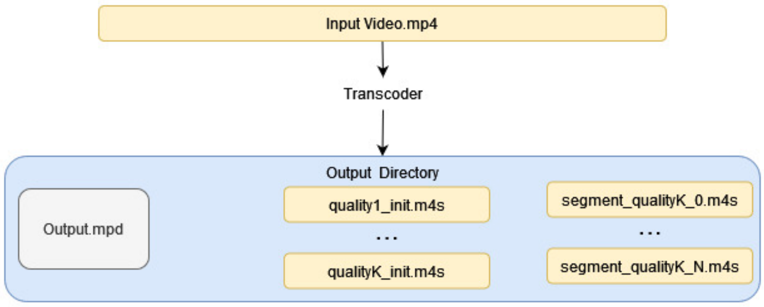


Figure 2.1: Diagram of DASH video encoding process

After encoding, a manifest file is generated, which includes information on the segments and informs a client of the naming convention for segments. It also includes different quality profiles and the bitrate of each quality profile. The client then uses this information to download the segments and play the video.

```
<Representation id="320x240 45.0kbps" mimeType="video/mp4"
    codecs="avc1.42c00d" width="320" height="240" frameRate
    ="24" sar="1:1" startWithSAP="1" bandwidth="45226" />
```

Listing 2.1: Example DASH manifest quality profile

2.1.2 HTTP Live Streaming (HLS)

HTTP Live Streaming (HLS) is a video streaming method initially drafted by Apple Inc. and later standardized by the IETF [10]. HLS is similar to DASH in that it also uses video split into segments and a manifest file to inform the client of the available segments available and their quality profiles. The main difference between DASH and HLS is that HLS uses the MPEG-2 Transport Stream (MPEG-TS) container format, while DASH uses the MP4 container format. MPEG-TS is a container format that stores video, audio, and other metadata [11].

Twitch uses HLS [12], a major live-streaming platform, to deliver live video content. HLS is also a fallback for YouTube on iOS devices [13], which natively supports the technology [14].

```
#EXT-X-STREAM-INF:BANDWIDTH=45226,CODECS="avc1.42c00d",
    RESOLUTION=320x240
320x240_45.0kbps.m3u80
```

Listing 2.2: Example HLS manifest quality profile

2.1.3 Progressive Streaming

Progressive video streaming is downloading a video file in byte ranges and playing the video as it is downloaded. To stream a video, the video file's header must be downloaded, containing metadata about the video, including bitrate, length, and framerate. The client can then use this information to request bytes from the video file in chunks using the HTTP *range* HTTP header. This HTTP video streaming method is not standardized and inflexible, so it was not considered for this project.

2.2 HTTP Versions

HTTP is an application-layer protocol that transfers data over the World Wide Web [15]. HTTP is a request-response protocol, meaning that a client sends a request to a server, and the server responds. The protocol was initially designed by Tim Berners-Lee in 1989 at CERN [15] and was created to transfer hypertext documents over the World Wide Web (which was referred to as *Mesh* at the time [15]). In modern applications, however, HTTP is used for much more than just hypertext documents [2].

HTTP version standards aim to implement HTTP semantics [16]. They do this by implementing an application-layer protocol that satisfies the HTTP semantics [16] and transferring this application data over existing transport layer protocols. HTTP/0.9, HTTP/1.X, and Hypertext Transfer Protocol Version 2 (HTTP/2) are all transported by Transmission Control Protocol (Transmission Control Protocol (TCP)) over IP. HTTP/3, on the other hand, is transported by QUIC over IP.

2.2.1 HTTP/0.9

Version 0.9 of HTTP is referred to as *the one-line protocol* [15] as its header only contains the "GET" method and the path to the document in one line before a line feed [17]. This protocol was designed to be idempotent[1] and be forward-compatible with future versions of the protocol. This version of HTTP was too simple for modern use and was replaced by HTTP/1.0 in 1996 [15].

[1]Idempotent means that the same request can be sent multiple times without changing the request result.

2.2.2 HTTP/1.0

Hypertext Transfer Protocol Version 1.1 (HTTP/1.0)/1.0 was the first version of HTTP/1.0 to be standardized by the IETF [18]. It was designed to support requests using version 0.9 of the protocol entirely. It was also designed to be extensible [18] by way of the header fields, allowing arbitrary header fields to be added to requests and responses. With this version of HTTP, users extended the protocol to add implementation-defined headers to add features to the communication between client and server.

2.2.3 HTTP/1.1

Hypertext Transfer Protocol Version 1.1 (HTTP/1.1) was the first HTTP version to enforce the *Host* header, which standardized the ability to use the same machine and IP address for multiple websites [19]. This was important, as IP address space is limited. The *Host* header also allows multiple distinct websites to share an IP address. This also allowed for machines to be used as web proxies. The externally-connected proxy machine could tell what website is requested based on the *Host* to proxy the connection to the appropriate web server. HTTP/1.1 also introduced *Persistent Connections* [19], which allows a TCP connection to stay open between the client and server for a server-specified length of time. Therefore, multiple requests and responses could occur over one TCP connection without creating a new connection for each request. This removed the overhead of a TCP handshake for each request and made the protocol considerably more efficient.

2.2.4 HTTP/2

HTTP/2 was initially designed by Google and called *SPDY* [20], initially drafted in 2009 [21]. It aimed to reduce the latency of web pages.

The standard introduced header compression called *HPACK* [22], meaning the headers can be sent in a single frame, which reduces the number of frames that need to be sent for the same data. This makes the protocol more efficient, as fewer TCP packets need to be sent.

The standard also introduced the concept of connection multiplexing to the protocol, allowing multiple traffic streams to take place over the same TCP connection [23]. This allowed a client to download multiple files concurrently.

server push was another feature specific to HTTP/2, which allows the server to push resources to the client before the client has requested them.

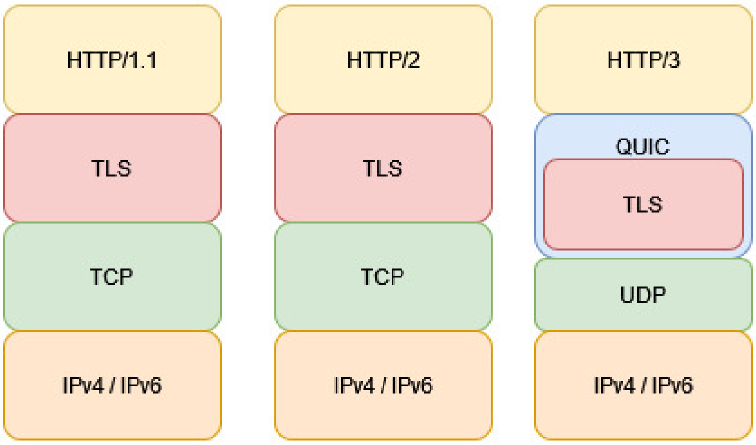


Figure 2.2: Diagram showing the different transport protocols used by the different HTTP versions

This allows the server to push resources the client will likely request shortly after an initial request, such as images or stylesheets. This feature of HTTP/2 is not widely used and will not be incorporated into testing throughout this project.

2.2.5 HTTP/3

HTTP/3 is the newest iteration of HTTP, standardized in July of 2022 [24]. It aims to re-implement HTTP/2's new features to use client-server connections efficiently. Some of the features that this project makes use of include; connection multiplexing and header compression. HTTP/3 was created to fix some shortcomings of HTTP/2, one of which is head-of-line blocking, explained in Subsection 2.4.3.

2.3 Head of Line Blocking

Head-of-line blocking occurs when packet loss occurs, causing the download of one file to delay the download of another — two factors in HTTP cause this: HTTP head-of-line blocking and TCP head-of-line blocking.

2.3.1 HTTP Head-of-Line-Blocking

In HTTP/1.1 and previous versions of HTTP, each file download requires a dedicated TCP connection [19, 22]. Browsers will only open a limited number of connections to avoid using excessive resources [25]. For most modern browsers, this TCP connection limit is six connections. The browser must enqueue the remaining requests if more than six files are requested. If packet loss occurs on one of the active file downloads, the queued files will be delayed until a connection is free.

2.3.2 TCP Head-of-Line Blocking

Head-of-line blocking can also occur due to the semantics of TCP. If multiple files are downloaded over a single connection, if packet loss occurs on one of the file downloads, the entire connection will be blocked until the packet is retransmitted. This is because TCP congestion control does not account for multiplexed TCP connections.

2.4 Transport Protocols

2.4.1 Transmission Control Protocol (TCP)

TCP is a connection-based transport protocol atop the IP network layer protocol [26] (see Figure 2.2). Connections are initiated via a handshake (see Figure 2.3), which takes one round trip to create a TCP connection. The handshake is used to establish a connection between the client and server and to synchronize the sequence numbers of the data being sent. Once the connection is established, data in packets called TCP segments can be sent in either direction.

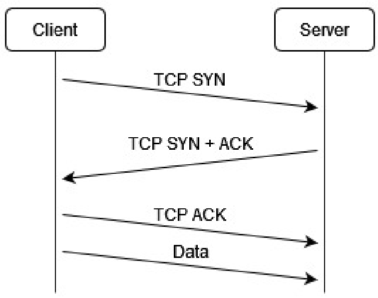


Figure 2.3: Diagram showing the TCP handshake

2.4.2 TCP over Transport Layer Security (TLS)

Transport Layer Security (TLS) is a cryptographic protocol that provides security to the data sent over a transport protocol (e.g., TCP) connection [27]. TLS is initiated with a secondary handshake (see Figures 2.4 and 2.5), wherein public key cryptography establishes a shared secret between the client and server. This shared secret encrypts the data sent over the connection without it ever being exposed to the network.

Adding TLS to a TCP connection adds additional round trips to the handshake. This addition may cause significant overhead in the time to transfer encrypted information. With version TLSv1.2 or lower [27], the TLS handshake requires two additional round trips to complete, whereas, for Transport Layer Security Version 1.3 (TLSv1.3), the TLS handshake only requires one additional round trip to complete.

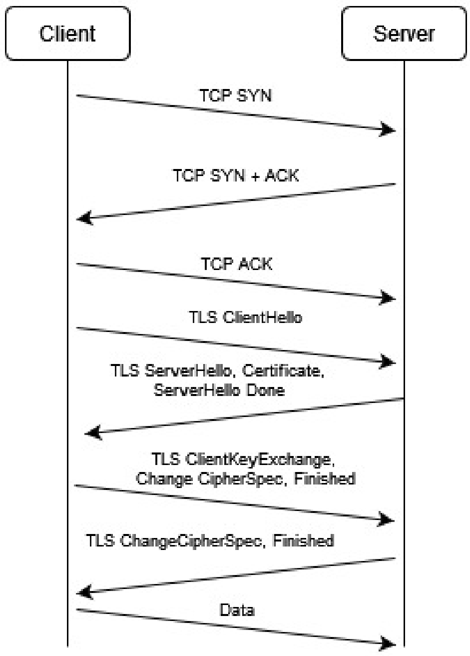


Figure 2.4: Diagram showing the TCP handshake with TLSv1.2 or lower

2.4.3 Congestion Control in TCP

Congestion control is a mechanism that is used to regulate the amount of data sent between the server and the client [28]. This is done to prevent any one

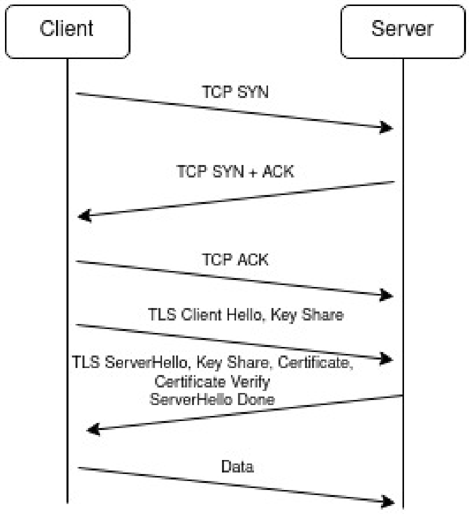


Figure 2.5: Diagram showing the TCP handshake with TLSv1.3

sender from overwhelming the network, which can cause packet loss and latency. Examples of how a network can be overwhelmed are if a sender is sending data at a rate that the receiver's buffer cannot handle or if the buffer of an intermediary device (for example, a router or switch) is complete. Congestion control is implemented via a Congestion Window (CWND), which starts small and increases as the data receiver sends acknowledgment packets (ACKs) to the sender to indicate that the data has been received in good condition.

TCP NewReno is a widely used congestion control algorithm in modern TCP implementations [29]. During the testing stages of this project, TCP NewReno was used. This algorithm adjusts its congestion window, which limits the number of unacknowledged packets that can be sent by the sender when packets are lost or delayed in the network.

The congestion control algorithm in TCP NewReno detects packet loss when the sender does not receive an acknowledgment (ACK) for a transmitted packet within a specified timeout period [29]. In response to this loss, the congestion window is reduced by setting it to a fraction of its previous value, called the congestion window threshold, which signals the sender to slow down and reduce transmitted data to prevent further congestion.

TCP NewReno implements a fast retransmit and recovery mechanism when congestion is detected [29]. During this phase, the sender immediately retransmits the lost packet without waiting for a retransmission timeout and increases the congestion window by a small amount for every ACK received for the outstanding packets. This enables the sender to recover quickly from the loss and continue transmitting data faster. If further losses occur during recovery, TCP NewReno enters a timeout phase. In that case, TCP reduces its congestion window size to its initial value and slows down its transmission rate to prevent further congestion.

TCP NewReno's congestion control algorithm optimizes the connection's throughput [29] while minimizing network congestion and packet loss. By identifying packet loss and modifying its congestion window, TCP NewReno can adapt to changing network conditions and ensure reliable and efficient data transfer.

2.4.3.1 Packet loss effect on HTTP/2

HTTP/2 uses a single TCP connection to send and receive data across multiple streams. This means the entire connection is slowed if multiple files are downloaded simultaneously, and a packet is lost for any stream. This is by congestion control rectifying the connections' instability and slowing down the transmission rate to prevent further congestion. For HTTP/1.1, this is not an issue, as if multiple files are being downloaded, they are doing so over separate TCP connections. This means that if a packet is lost for one of the connections, the other connections are unaffected.

2.4.4 User Datagram Protocol (UDP)

User Datagram Protocol (UDP) is a connectionless transport protocol atop the IP network layer protocol [30]. UDP is a simple protocol that does not implement many of the features that TCP does. It does not implement flow control, error checking, or retransmission of lost data. UDP is often used for applications that do not require these features, such as streaming real-time applications like video games. UDP is also used for DHCP (Dynamic Host Configuration Protocol) [31], as UDP does not require a client-server connection, unlike TCP, which cannot broadcast to multiple machines to accept an IP address.

For data to be transferred, it is first broken down into smaller chunks called datagrams [30]. Each datagram is labeled with header data detailing the source and destination ports and the length of the datagram payload. There is no guarantee that the destination will receive a datagram or receive it in the same order it was sent.

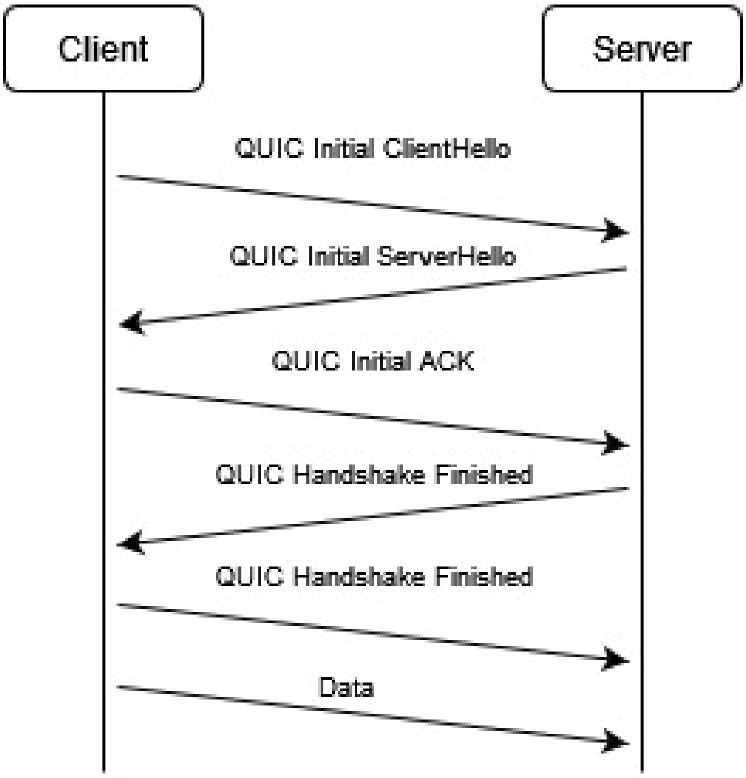


Figure 2.6: Diagram showing the initial QUIC handshake with TLS

2.4.5 QUIC

Initially developed by Google in 2012, QUIC is a general-purpose transport layer protocol that sits atop UDP [32]. QUIC aimed to be a plug-in replacement for HTTP/2, to improve user experience concerning page load times [32]. QUIC is a connection-oriented protocol [33], similar to TCP, to provide many of TCP's features. QUIC does this while attempting to avoid some of the pitfalls associated with TCP, such as the requirement for a client-server connection to be established before data can be sent, head-of-line blocking, and lack of seamless connection migrations.

2.4.5.1 Standardization of QUIC and HTTP/3

Google submitted QUIC to the IETF as a draft in 2016 [34], standardized in 2021 [33]. QUIC was initially intended to encapsulate both the transport protocol and the HTTP binding (HTTP-over-QUIC) [34]. During the

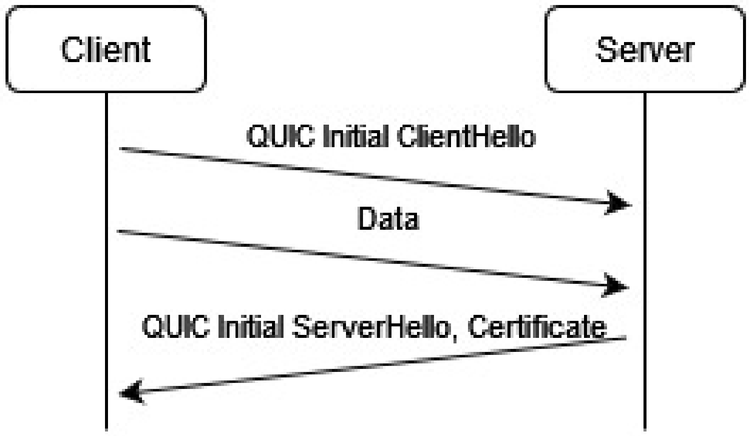


Figure 2.7: Diagram showing the QUIC handshake with 0-RTT

drafting process, however, the IETF distinguished a clarification between the two, designating the transport protocol as QUIC and the HTTP binding as HTTP/3. This was done to avoid confusion and communicate that QUIC is a general-purpose transport protocol, not just a transport protocol for HTTP [35].

2.4.5.2 Streams

As stated in RFC 9000, QUIC: *A UDP-Based Multiplexed and Secure Transport*, QUIC transmits data over a connection through *streams*. These streams send data in the form of QUIC packets in sequential order. QUIC supports both unidirectional and bidirectional streams. Unidirectional streams send data from one endpoint to another, while bidirectional streams send data in both directions. This allows a single QUIC connection for multiple data streams, such as multiple files being downloaded concurrently.

Stream prioritization is a fundamental feature of the QUIC protocol that enables the QUIC implementation to organize streams based on their relative importance [33]. This can improve the overall performance of the data transfer by ensuring that critical data is handled as quickly as possible.

In addition, to stream prioritization, QUIC supports stream multiplexing [33], which enables multiple streams to be transmitted over a single connection simultaneously. This reduces the overhead of opening and closing multiple connections, making data transfer more efficient. Stream multiplexing and prioritization work together to provide a flexible and efficient mechanism for data transfer in QUIC.

2.4.5.3 Connection Migration

QUIC allows for connection migration by way of using client & server connection IDs [33]. Connection IDs are 64-bit identifiers used on both server and client to identify a connection and hold context on each. They can be used to identify a connection when a client migrates to a new network, such as switching from an LTE mobile network to a public Wi-Fi network. This allows a connection to persist through switches in the IP address. This can allow users to keep a connection ongoing while moving between networks seamlessly without interruptions. In video streaming, this may cause a downloading segment to be lost over TCP, but with QUIC, the download should continue over the new network.

2.4.6 Congestion Control in QUIC

The UDP transport protocol does not implement congestion control [30], so it is up to QUIC to implement this feature.

QUIC's modular congestion control can be swapped out for other congestion control algorithms [36]. According to the standard [36], the default congestion control algorithm is based on NewReno and is referred to in the document under the same name.

One significant change is that congestion control is on a per-path basis [36]. This means that if a connection is slowed down due to packet loss, all other streams on different paths are unaffected, as they are on separate paths. This is a significant improvement over TCP. If a packet is lost in a connection, the entire connection is affected, which is significantly detrimental for HTTP/2, which downloads multiple files over a single connection (see 2.4.3.1).

2.4.7 Transport Layer Security (TLS) and QUIC

Instead of the transport protocol being transported over TLS v1.3 and is incorporated as part of QUIC [33] (see Figure 2.2). During a QUIC connection initiation with a previously unseen server, a client can send encrypted data after just two round trips instead of the three round trips necessary for TCP+TLS. For a server the client has previously exchanged cryptographic details for, encrypted data can be sent without waiting for a round trip to complete (see Figure 2.7). Alternatively, TCP requires an additional round trip to initiate a TCP connection.

This version of TLS will not be optimized by kTLS (Kernel Transport Layer Security), as QUIC is not run in kernel space. This may pose a significant advantage for TCP over TLS, as Netflix suggests it gives considerably faster handshakes [37].

Chapter 3

Analysis

HTTP/3 and QUIC are relatively new protocols, and up until recently, they were still in a draft state [24, 33] and were not recommended for production systems. However, with the release of the IETF standard RFC 9114, HTTP/3 is now considered stable and ready for production use. Due to its recent standardization, a few full implementations of HTTP/3 and QUIC are available together. That being said, some implementations do exist. This chapter will discuss some of the implementations of HTTP/3 and QUIC, their implementation details, and any issues faced by the student while using them.

3.1 Server Implementations

3.1.1 NGINX

NGINX is an open-source, high-performance web server that is a popular choice in the open-source community [38]. Among the web servers in use today, NGINX has been the most-used web server for the past four years [39].

NGINX announced adding experimental support for QUIC and HTTP/3 on a separate branch of the mercurial NGINX source code repository in 2020 [40]. This branch still needs to be merged into the main branch, but NGINX plan on merging it into version 1.25 of the software by 04/11/24 [41]. The student compiled NGINX from source and linked it to Google's BoringSSL library [42], a fork of OpenSSL that supports TLSv1.3[1]. The stu-

[1]OpenSSL's maintainers have decided not to support QUIC for the foreseeable future [43]. As a result, QUIC implementations use a fork of the OpenSSL library to add the

dent then compiled NGINX with the QUIC stream, HTTP/3 module, and HTTP/2 module.

NGINX's implementation of QUIC and HTTP/3 is based entirely on the IETF standards [33] [24]. It does not use proprietary extensions, needs few dependencies, and is fully compatible with other QUIC and HTTP/3 implementations.

3.1.2 Cloudflare Quiche + NGINX

Cloudflare is a large CDN provider that provides an HTTP/3 proxy as a service [44]. They have open-sourced a part of the underlying code for implementing QUIC and HTTP/3 that runs their proxy. This library is called *quiche* [45] [2]. Cloudflare *quiche* is a library that can implement both a QUIC server and a client. *quiche* is written in the Rust programming language and is designed to be highly performant.

Cloudflare *quiche* has a built-in patch for NGINX that adds QUIC & HTTP/3 support to the web server. The student compiled Cloudflare *quiche* from its source and applied *quiche*'s patch to NGINX. Using the *quiche* patch, the student could compile NGINX with HTTP/3 support, once again linking it against Google's BoringSSL library.

The Cloudflare *quiche* implementation of HTTP/3 and QUIC is based on the IETF standards but includes several extensions. The extensions noted by the student were that the default congestion control algorithm employed by Cloudflare *quiche* is a non-standard version of the Cubic algorithm, which is different from the NewReno default suggested in the IETF standard [33]. As well as this, there are multiple GitHub issues on the repository, pointing out differences between the implementation and the official IETF Standard [46, 47].

3.1.3 Google Quiche + Envoy

Google owns many websites ranking in the top 100 visited [6]. They also created the initial draft of the QUIC protocol and released an open-source implementation of QUIC and HTTP/3 called *quiche* [48]. Google *quiche* is a library that can implement both a QUIC server and a client. It is written in C++ and is the underlying library in the Chromium client implementa-

support themselves

[2] Google also has an open-source QUIC and HTTP/3 compliant library called quiche so that they will be referred to as Cloudflare *quiche* and Google *quiche*, respectively

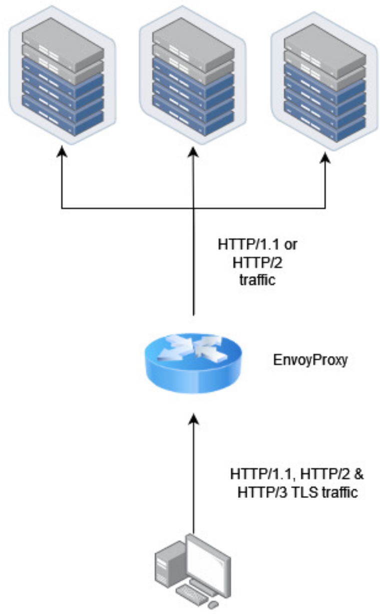


Figure 3.1: Diagram of Envoy's use of HTTP/3

tion. Envoy is an open-source proxy server used by Google to route traffic between their services [49].

As the HTTP/3 implementation for Envoy is intended just to be a TLS and HTTP/3 terminating proxy to underlying web servers, the student did not consider this implementation for testing, as it is intended to be used as a web proxy, and not a web server, unlike the other implementations. However, the student did compile Envoy with Google's *quiche* library and successfully connected to the server using the Chromium client implementation and proxied web traffic through the server to a remote HTTP/1.1 server.

3.1.4 QUIC-Go + Caddy

QUIC-Go is a QUIC and HTTP/3 implementation written in the Go programming language [50]. It is a library that implements a QUIC server and a client. It is designed to closely represent the official RFC 9000 [33] and RFC 9114 [24]. However, at the time of writing, due to Go's garbage collector[3], when the library's API allocates memory, a significant amount of CPU time is required to free the memory. This can cause significant performance degradation compared to other implementations, as the Go garbage collector

[3] free up memory after it has been allocated and subsequently no longer needed

performance issues at runtime [51]. There is ongoing work to improve the library's performance, but at the time of writing, it is not yet complete [51]. For this reason, this library implementation was not considered for testing.

Caddy is a lightweight web server written in the Go programming language and has HTTP/3 support built-in using the QUIC-Go library [52]. The student downloaded the official binary package available from the GitHub repo [52], which had out-of-the-box support for HTTP/3.

3.2 Client Implementations

3.2.1 Chromium

Google Chrome and other Chromium-based web browsers [53] make up over 62% of the web browser market share [54]. Chromium is an open-source project that Google maintains [55]. It is written in C++ and is the underlying codebase for Google's QUIC and HTTP/3 implementation, Google *quiche*. As Google is the project's maintainer, the QUIC implementation is likely highly optimized since they drafted the protocol. This would show they have much experience with it, making it a good choice for testing, as some other implementations may not be as optimized yet. Most people streaming video will likely do so using Google Chrome [54].

3.2.2 QUIC-Go

QUIC-go, as was discussed in subsection 3.1.4, QUIC-go is unoptimized and will not be considered for testing as a client. Its performance is not comparable to other implementations, and testing it against the other HTTP versions would be unfair.

3.2.3 Quiche + cURL

cURL is a command-line utility for transferring data using various protocols. It is written in C and is available on most Linux distributions [56]. The Cloudflare *quiche* library can be used to add HTTP/3 support to cURL [45]. The student compiled Cloudflare *quiche* from its source and linked the static object files to cURL while compiling it. The student successfully connected to previously-mentioned server implementations using the custom-built cURL and downloaded files using HTTP/3.

Chapter 4

Implementation

4.1 Overview

This chapter explores the implementations of the tests carried out in this project. The network environment used for the tests was discussed first, and how it was altered to simulate different conditions.

4.2 Traffic Shaping

To simulate different network conditions, a method called *traffic shaping* is used to alter the network conditions [57]. Traffic shaping is a technique used to simulate different network conditions by altering the network characteristics [58]. The tool used in the project to implement traffic shaping was *tc* (traffic control), a Linux utility that allows for the manipulation of network traffic. *tc* allows for the manipulation of latency, random packet loss rate, and bandwidth. Multiple *class*es of traffic can be manipulated, but in this project, we only modify the *root* class [58]. The *root* class refers to egress traffic, which was the only traffic that was required to be altered for the tests carried out on both client and server (See Figure 4.1).

4.2.1 Bandwidth

Bandwidth was altered using the *tbf* (Token Bucket Filter) module of *tc*. The *tbf* module allows for the shaping of the bandwidth of a network interface card (NIC) [58]. The *tbf* module adds a token bucket to the NIC, which is filled at a user-defined rate. When a packet is sent, a token is removed from the bucket. If there are no tokens in the bucket, the packet is dropped.

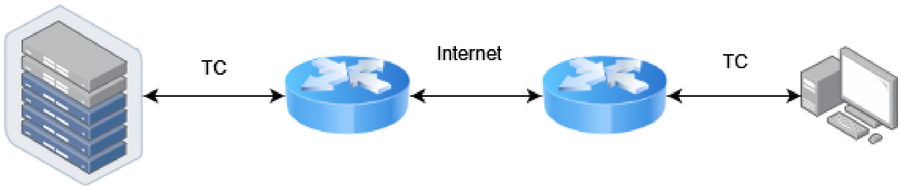


Figure 4.1: Traffic Shaping

This allows for the shaping of the bandwidth of the NIC. Bidirectional bandwidth shaping was not necessary, as the data the client sent to the server was negligible compared to the bandwidth constraint applied to the server.

```
#!/bin/bash
tc qdisc add dev eth0 root tbf rate 50mbit burst 32kbit
    latency 200ms
```

Listing 4.1: Adding bandwidth constraints with TBF

4.2.2 Latency

To shape the network latency between the client and server, we used the *netem* (Network Emulation) module of *tc* to shape egress latency. The *netem* module adds a packet queue to hold packets for a user-defined amount of time before sending the data to the Network Interface Card (NIC) [58], which then sends the packet to the network. This allows for the addition of a delay to the packets, which simulates network latency. The methodology mentioned above was applied on both client and server machines to add bidirectional latency to the connection, ensuring the latency was the same in both directions.

```
#!/bin/bash
tc qdisc add dev eth0 root netem delay 100ms
```

Listing 4.2: Adding latency with Netem

4.2.3 Packet Loss

The Linux tool *tc*'s *netem* module is also used to add random packet loss. It does this by adding a packet queue to the NIC, randomly dropping a user-defined percentage of packets.

```
#!/bin/bash
tc qdisc add dev eth0 root netem loss 3%
```

Listing 4.3: Adding packet loss with Netem

4.2.4 Combining TBF with Netem

All tests conducted constrained the server to a 50 Mbps bandwidth (using *tc* and *tbf*) and 23ms of delay (added by natural network latency). To apply the packet loss rate or network latency for testing purposes, bandwidth, and either latency or loss would have to be applied to the NIC simultaneously [58]. This was done by using *tc* filters. Multiple network conditions can be applied with *tc* simultaneously using filters.

```
#!/bin/bash
tc qdisc add dev eth0 root tbf rate 50mbit burst 32kbit
    latency 200ms
tc qdisc add dev ens3 parent 1:1 netem $NETWORK_CONDITION
```

Listing 4.4: Traffic Shaping Script with Netem and TBF

4.3 Web Server

The student chose to use the *nginx* web server for the project, for the reasons outlined in Chapter 3, due to how closely it represents the IETF standard of HTTP and QUIC [24, 33].

When the student first began testing the web server, it was necessary to compile the web server from its source, using the methodology discussed in Chapter 3. However, in February of 2022, NGINX announced pre-built packages for a version of *nginx*, supporting QUIC and HTTP/3 that was available for Ubuntu 22.04. This allowed for a more straightforward orchestration of the test environment, as the web server could be installed using the package manager instead of manually compiling the web server from the source.

The server configuration was mainly left to the default configuration, except for *quic-gso*, *ssl-early-data*, and *quic-retry* [59], which were all set to *on*. The student used the Let'sEncrypt Certificate Authority to get a valid SSL cert, which was used to set up TLSv1.3 for each NGINX listener. HTTP/2 and HTTP/3 were both bound and listening on port 443 (using the NGINX *reuseport* option [59]), which is the semantic port for HTTPS [60]. HTTP/1.1 was set up to listen on port 8443. An *add_header* [61] directive was used to add an *Alt-Svc* header to HTTP responses, advertising the availability of HTTP/3 on port 443.

The student also configured all HTTP versions to serve the duplicate static files, which was used to test the web server's performance.

4.4 Clients

Now that a web server supporting multiple HTTP versions was set up, the web server's performance was tested. To test the web server, the student needed to use HTTP clients that supported all three versions.

4.4.1 File Download Tests

While the time it takes to download files is not necessarily a good metric to determine video streaming performance on its own, it can still provide insight into the overall performance of the protocol under differing network conditions.

Requirements for a test setup to test the file download performance of the web server were as follows:

1. be able to download files from the web server.
2. be able to files over different HTTP protocols
3. be able to record the time it takes to download a file
4. be able to alter the network shaping on the server and client

The student used *Bash* to create an appropriate script to realize each requirement.

4.4.1.1 Requirement: Download files

The student used the *curl* command line tool to download files from the web server. *curl*, as was detailed in Chapter 3, is a command line tool that allows for the transfer of data using a variety of protocols. *curl* is available in the Ubuntu package repository and can be installed using the package manager *apt*.

The application data was then redirected to */dev/null* to avoid writing the data to disk, which would have been unnecessary and would add to the test time to write the data to disk.

```bash
#!/bin/bash
curl -s -o /dev/null $URL
```

Listing 4.5: Downloading files with curl, and redirecting to */dev/null*

4.4.1.2 Requirement: Use different HTTP protocols

As was described in Chapter 3, the student compiled *curl* against the Cloudflare *quiche* library. This custom-built version of *curl* supports HTTP/3, HTTP/2, and HTTP/1.1. The student could then use *curl* to download files over the three protocols. The *Bash* script used this version of *curl* to request files over the different HTTP protocols. To differentiate between downloading over HTTP/3 and HTTP/2 (since they share the same port number of 443), a special command-line flag was used with *curl* to specify the protocol. To differentiate between HTTP/1.1 and the other protocols, the port number was changed to 8443 to download the file over HTTP/1.1.

```bash
#!/bin/bash
curl --http3 -s -o /dev/null $HOSTNAME
```

Listing 4.6: Downloading files over different HTTP protocols

4.4.1.3 Requirement: Record the time it takes to download a file

The student initially attempted to use the Linux *time* command line tool to record the time it takes to download a file. *time* is a command line tool that runs a command and records the time it takes to run the command. The student used the *-f* flag to format the output of the *time* command to be the time in milliseconds that *curl* takes to complete, which was then used by the *Bash* script to record the time it took to download a file.

This, however, proved to be imprecise, as the *time* command would record the time taken for the command to start up and subsequently finish instead of the time for the file to be downloaded.

To avoid this, the student used *curl*'s write-out flag, which allows for outputting the time to download or some other metrics. The student used the *time_total* variable to output the time from request to the entire data transfer completion [62]. This was then used by the *Bash* script to record the time it took to download a file.

```bash
#!/bin/bash
curl --http3 -s -o /dev/null -w "%{time_total}\n" $HOSTNAME
```

Listing 4.7: Downloading files with curl, and recording the time it takes to download the file

4.4.1.4 Requirement: Alter the network shaping on the server and client

To apply network shaping with *tc* as detailed in Chapter 3, the script would need to run commands on the server to apply network shaping remotely. To do this, the student used Secure Shell (SSH) to run commands remotely on the server. SSH is a protocol that allows for the execution of commands on a remote machine [63]. SSH Keys were used to authenticate the client to the server, allowing the client to run commands on the server without waiting on user input for a password.

To run different tests under different network conditions, the *Bash* script would read in a text file which defined a label to name the results file for each test, the file to download, and the *tc* command to run on the server.

```
2mb_loss_5 2MB.txt tc qdisc add dev eth0 root netem loss 5%
```

Listing 4.8: Example line of a testfile used by the *Bash* script

4.4.2 Video streaming test

To address the aims of this report, the student tested the video streaming performance of the web server. To do this, the student used the Google Chrome browser to stream a video from the web server. As Google Chrome has the majority of the market share for web browsers [54], it was decided that it would be the best choice for testing the video streaming performance of the web server. Google Chrome is based on the Chromium open-source project, which, as was discussed in Chapter 3, has support for HTTP/3 & QUIC via Google *quiche*.

A website that used the DASH video streaming protocol was used to test video streaming potential on a browser. Requirements for a test setup to test the video streaming performance of the web server were:

1. be able to stream a video from the web server over different HTTP versions
2. be able to keep track of bitrate switches and the minimum, mean, and maximum bitrate of streamed video
3. be able to keep track of buffer length throughout playback of the video stream
4. be able to store the results of each test

5. be able to define a test scenario for each test

The student used *ffmpeg* and *GPAC* [64, 65] to transcode Blender's Big Buck Bunny video [66] into a series of DASH video streams at multiple bitrates with three distinct segment sizes. The student then used *react* [67] to build a website that could stream the video over HTTP using the *dash.js* library. *React* was used due the simplicity of including the *dash.js* [68] library in the web app. *NextJS* [69] was the framework used to build the website, as it is a framework that allows for server-side rendering of React components and includes a simple backend server that was used for storing the video stream metadata.

4.4.2.1 Requirement: Be able to stream a video from the web server over different HTTP versions

A website was built making use of the *dash.js* [68] library to stream a video over HTTP using TypeScript (TS) [70], a superset of JavaScript (JS) that adds a sophisticated type system to JS and gets trans-piled to JS. This allows for *dash.js* to feed video segments to the HTML5 video player and use the browser to fetch the video segments. The browser will abstract away the details of the HTTP protocol from the underlying JavaScript code in the library. The library can then stream video without depending on a specific version of HTTP.

4.4.2.2 Requirement: Be able to keep track of bitrate switches and the minimum, mean, and a maximum bitrate of streamed video

The *dash.js* [68] library allows video stream metadata capture at runtime. One such metric that can be read is the current bitrate of the video stream. TypeScripts global *setInterval* [71] function was used to poll the metrics at an interval of one second. The metrics were then stored in *React* state variables [67], which tracked the complete history of the video stream's bitrate throughout the entire length of the playback. The minimum, mean, and maximum bitrates were calculated when the video finished.

4.4.2.3 Requirement: Be able to keep track of buffer length throughout playback of the video stream

The buffer length at any given time is another metric capable of being queried at runtime. This was also polled with the same *setInterval* function and stored in *React* state variables.

4.4.2.4 Requirement: Be able to store the results of each test

The results of each test were serialized into a JavaScript Object Notation (JSON) [72] variable before being sent to the backend Application Programming Interface (API). The API would then unmarshal this JSON object into a Structured Query Language (SQL) [73] *INSERT* query before using a *MySQL* client to execute the query on the Planetscale database [74]. The results of the test were then stored in the database. This allowed for the results to be queried and granularly analyzed later.

4.4.2.5 Requirement: Be able to define a test scenario for each test

To define a test scenario for each test, the student used URL query parameters, which permitted the definition of a test scenario to be passed to the website. The website then uses the query parameters to define and run the test scenario. This allowed the test scenarios to be defined in a URL, which could be shared with others to run the same test scenario.

Variables that could be defined in the test scenario were:

- the name of the video to stream
- the size of video segments to request
- the network constraint applied
- the number of times to repeat the test

These variables were then used to define the video stream and different labels used in the serialized JSON object to define metadata about the test (including the network constraint applied).

4.5 Data Analysis

Initially, *Go-Echarts* [75] was planned to be utilized for automation of graphs using recorded metrics from experiments using *quic-go*. However, *quic-go* presented performance problems outlined in 3, which went unused in the final results and report.

After this, a custom *curl* script was used to automate data collection before feeding the results into a Python *matplotlib*-based script to generate graphs [76]. This was a much more time-consuming process than initially planned, as the student had to manually define graphs and their properties in the Python script for each test scenario. Due to this, the student opted to use Google Sheets [77] to generate graphs for most of the experiments, as it was much quicker to generate graphs in Google Sheets than in Python.

Google Sheets was a valuable tool for generating graphs. Data could be chosen granularly to include different formulae to calculate valuable metrics, such as the interquartile range of buffer length. However, as the graphs became more complicated, Google Sheets proved unsuitable, as graphs with multiple axes and legends were challenging to create. Due to this, the student then moved to Microsoft Excel [78] to generate more complex graphs.

Microsoft Excel proved invaluable for generating graphs, as it allowed for creating complex graphs with multiple axes and legends. However, the student had to manually define the data to include in the graphs, which was time-consuming. The student also had to manually define the axes and legends for each graph, which was also time-consuming. That being said, it was the best tool used for generating graphs. Microsoft Excel generated all graphs shown in 5.

Chapter 5

Experiments

Experiments were run against an OVH Virtual Private Server (VPS) with the following specifications:

- CPU - Unspecified Intel Xeon
  - 1 vCore
  - 1 Thread
  - 2.4 GHz Clock Speed
  - x86-64 Architecture
  - Intel
  - Meltdown mitigation: PTI (Page Table Isolation)
- 2 GB RAM
- 20 GB SSD
- 100 Mbps network
- Ubuntu 20.0.4 LTS, Kernel 5.15.0-69-generic

Experiments were recorded from a local machine (Framework Laptop [79]) with the following specifications:

- CPU - Intel Core i7-1260p
  - 12 Cores
  - 16 Threads

- 4.7 GHz Boost Clock Speed
  - 3.4 GHz Base Clock Speed [1]
  - x86-64 Architecture
  - Intel
  - Meltdown mitigation: Not affected

- 32 GB RAM
- 1 TB SSD
- 1 Gbps network
- Ubuntu 22.0.4 LTS, Kernel 5.19.0-40-generic

Sustained download throughput was measured using the *iperf3* tool, a command-line tool for testing network bandwidth [80]. The tool was run in server mode on the VPS and client mode on a local machine. The client machine was connected to the VPS via a 100 Mbps network link.

Sustained TCP download throughput was measured at 92Mbps with 0% packet loss. Sustained UDP download throughput was measured at 86Mbps with 0% packet loss. Tests were run multiple times to ensure consistency.

File download tests are an average of 30 iterations of tests run with *curl*, as described in Chapter 4. Video streaming tests are an average of 3 iterations of tests run on the remote machine streaming Big Buck Bunny [66] for nine minutes and fifty-six seconds each using *dash.js*, as described in Chapter 4. Initial load time tests were the mean average of the three tests' initial load times.

5.1 Results

5.1.1 Video Streaming Results Under Optimal Conditions

Under optimal network conditions, of maximum download speed of 50 Mbps, 23ms round trip time, and 0% packet loss, HTTP/3 performed similarly to the other versions of HTTP, except for minor regressions in video buffer length.

[1] This CPU has efficiency and performance cores, more details can be found at `https://ark.intel.com/content/www/us/en/ark/products/226254/intel-core-i71260p-processor-18m-cache-up-to-4-70-ghz.html`

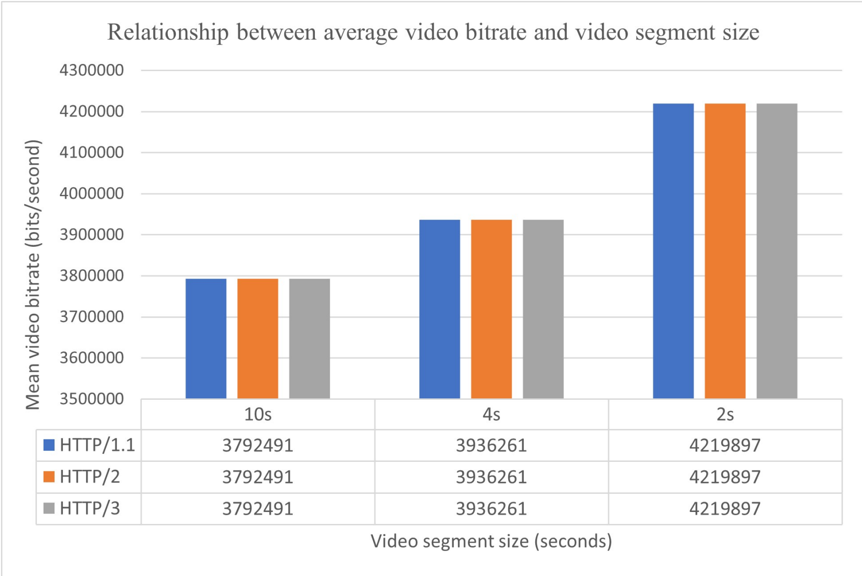


Figure 5.1: Mean bitrate of HTTP/3, HTTP/2, and HTTP/1.1 under optimal network conditions

As shown in Figure 5.1, HTTP/3 yielded the same mean streamed video bitrate as the other two protocols. This is expected from any competent protocol for streaming video, as the maximum video bitrate shown is 4.2 Mbps when the link bandwidth is 50 Mbps.

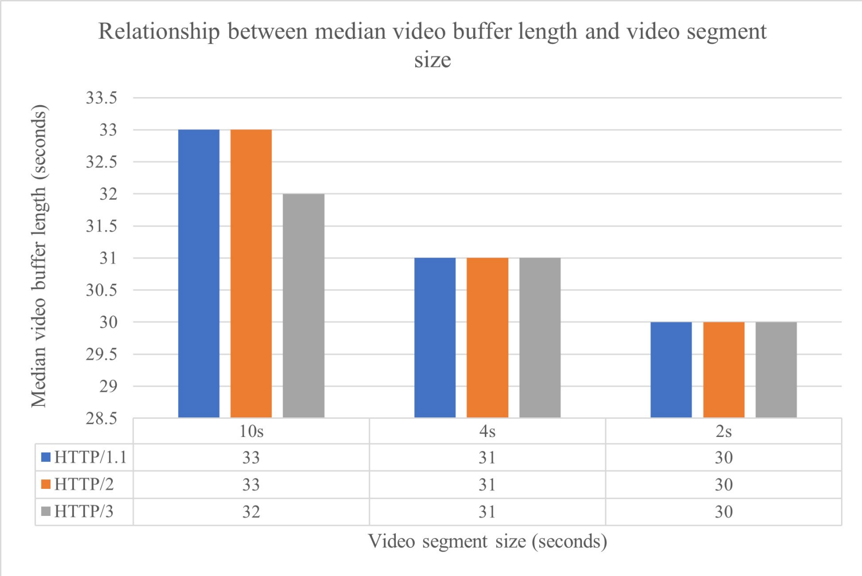


Figure 5.2: Median buffer length of HTTP/3, HTTP/2 and HTTP/1.1 under optimal network conditions

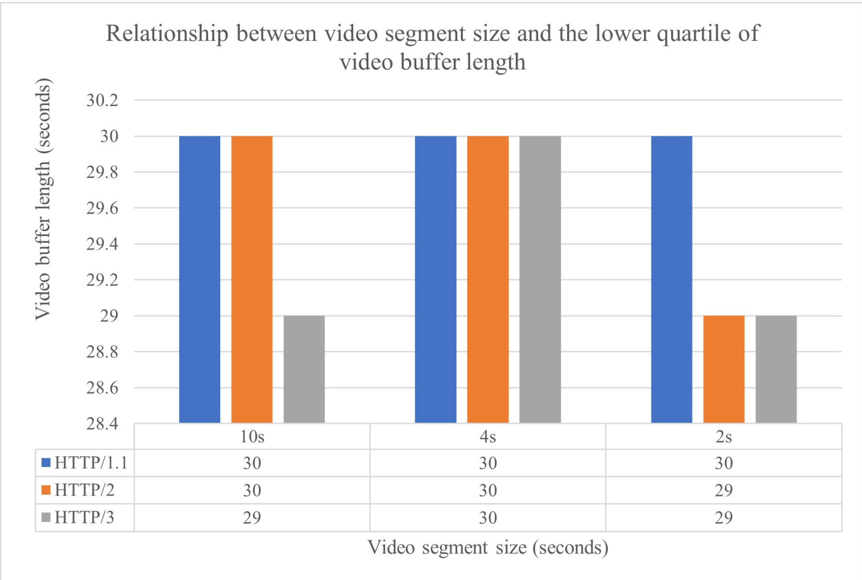


Figure 5.3: Lower quartile of buffer length of HTTP/3, HTTP/2 and HTTP/1.1 under optimal network conditions

When comparing the median and lower-quartile buffer lengths between the three protocols, HTTP/3 performed slightly worse than HTTP/2 and HTTP/1.1, as seen in Figure 5.2. This 1% decrease in the median buffer length and 3% decrease in the lower-quartile of buffer length for HTTP/3 is only present for ten-second segments. However, for HTTP/3, the other segment sizes yield equal results with the other protocols.

This is a minor regression in performance and can likely be explained by HTTP/3's slight decrease in throughput performance at higher bandwidths, as was seen in Figure 5.5. While the ten-second segments comprised significantly lower bitrate data, the overall segment was more prominent, and thus, the overall segment took longer to download. The other segment sizes, which are considerably smaller, are likely to have taken advantage of HTTP/3's stream multiplexing capabilities. This would likely have resulted in a better overall throughput when streaming using lower segment sizes, which would explain the similar performance with the other protocols for the smaller segment sizes (four seconds and two seconds).

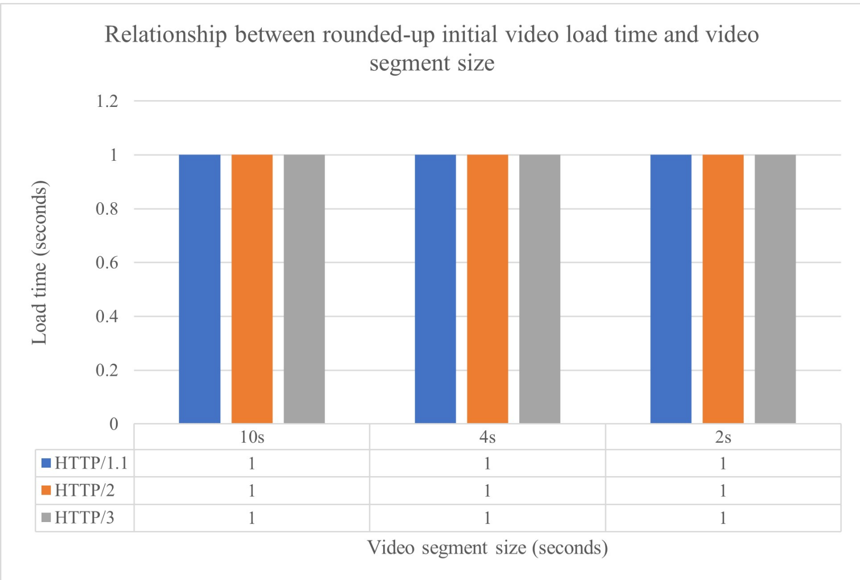
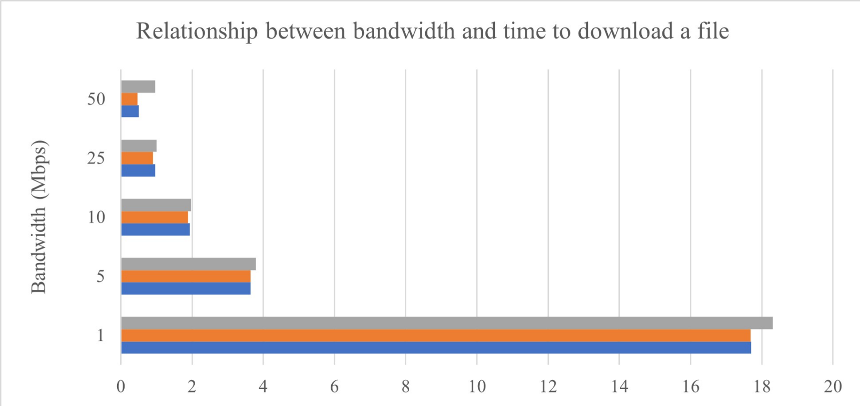


Figure 5.4: Rounded-up initial load time of HTTP/3, HTTP/2 and HTTP/1.1 under optimal network conditions

The initial load time was also measured; the results can be seen in Figure 5.4. All protocols yielded a sub-second initial load time of one second or less. This, however, is to be expected as the size of a single segment of any of the segment sizes should be

5.1.2 Performance under bandwidth constraints

5.1.2.1 File download performance



	1	5	10	25	50
HTTP/3	18.31354437	3.795143767	1.9792732	0.997971	0.969709567
HTTP/2	17.6869623	3.6445929	1.892591833	0.899321567	0.462405467
HTTP/1.1	17.7012109	3.641161667	1.9321617	0.960597933	0.499910867

Figure 5.5: Mean time to download a 2 MB file using HTTP/3, HTTP/2, and HTTP/1.1 under different bandwidth constraints

Under all bandwidth constraints, HTTP/3 had less throughput than the other versions of HTTP. This can likely be explained by three causes: congestion control ramp-up, packet length, and Operating System (OS) scheduling.

QUIC's congestion control algorithm, as detailed in Chapter 2, uses a slow-start technique to ramp up the amount of data that can be sent without causing congestion. This may cause performance degradation because, from Figure 5.5 results, HTTP/3's throughput is significantly lower than the other protocols at higher bandwidths. This is likely due to the slow-start ramp-up of the congestion control algorithm, which would cause HTTP/3 to have lower throughput than HTTP/2 and HTTP/1.1 earlier during the download.

The packet length observed for QUIC downloads was 1294 bytes (including UDP and IP headers) over IPv4. This is significantly smaller than the corresponding packet lengths for TCP of up to 1506 bytes. This likely gives QUIC a disadvantage compared to TCP, as under these conditions, TCP can send more data in fewer packets than QUIC. The packet length and UDP Kernel implementation overheads would contribute to QUIC performing worse than TCP under any bandwidth.

The last proposed cause of HTTP/3's performance degradation is page-table isolation for user space programs [81]. HTTP/2 and HTTP/1.1 are both TCP-based protocols and, thus, are implemented in the Linux kernel [82]. Programs that run in kernel space are not subject to the same scheduling environment as user space programs [81]. User space programs are subject to context switches, which is the context of switching between kernel and user space, which causes significant overhead, because since Meltdown [83], for affected Intel processors, the Linux kernel isolates page tables from other user space programs to protect the data, and then reload them into cache when switching back to kernel space. This means that any program running in kernel space will likely be more performant than the same program running in user space. This low-level performance advantage of kernel space programs is likely the cause of HTTP/2 and HTTP/1.1 performing better than HTTP/3 in terms of throughput.

5.1.2.2 Video streaming performance

These tests show some metrics of video streaming across the three protocols and how well they each perform under constrained bandwidth. The metrics measured were the median and lower quartile of buffer lengths, the initial load time, and the mean used video bitrate. The results of these tests can be seen in Figures 5.7, 5.8, 5.11, and 5.5.

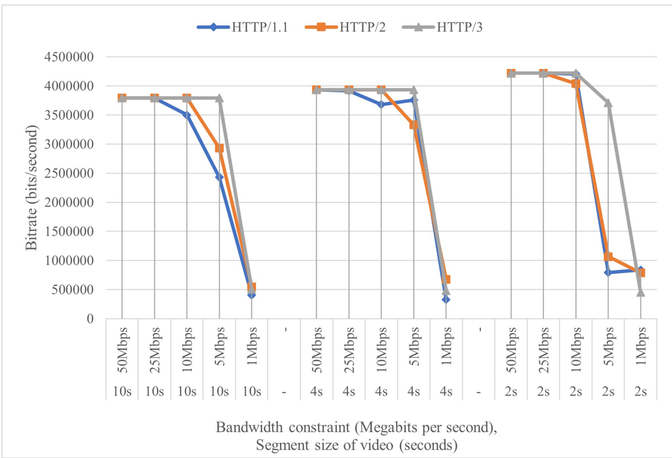


Figure 5.6: Relationship between mean video bitrate and bandwidth

The mean bitrate used by each protocol is shown in Figure 5.6, and the results show that HTTP/3 performed similarly with the other protocols for bandwidths of 50 Mbps, 25 Mbps, and 10 Mbps. However, under a bandwidth constraint of 5 Mbps, HTTP/3 performs better than the other protocols for any segment size. This may be explained by QUIC's stream prioritization implementation being different from HTTP/2's.

At the lowest-tested bandwidth constraint of 1 Mbps, HTTP/3 performed equal to or worse than the other protocols. This can likely be explained by QUIC's inefficiencies on Linux (see Section 5.1.2.1).

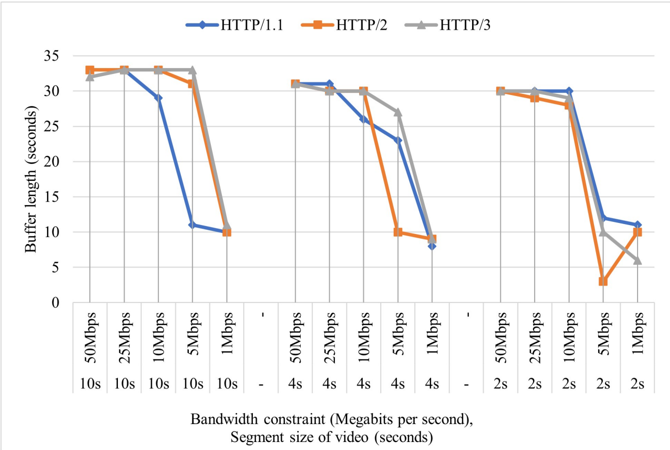


Figure 5.7: Relationship between median video buffer length and bandwidth

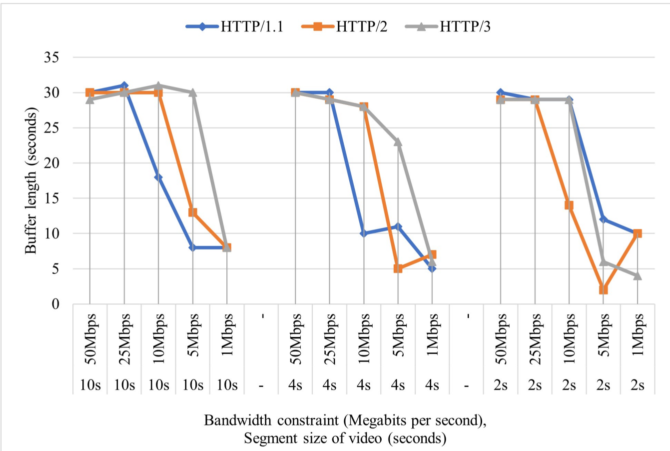


Figure 5.8: Relationship between the lower quartile of video buffer length and bandwidth

Regarding the median buffer length in Figure 5.7, HTTP/3 performs equal to or better than the other protocols until a bandwidth constraint of 1 Mbps. At this bandwidth constraint, HTTP/3 performs worse than the other protocols.

The lower quartile of buffer lengths in Figure 5.8 shows that HTTP/3 performs similarly to the other protocols until a bandwidth constraint of 1 Mbps. At this bandwidth constraint, HTTP/3 performs worse than the other protocols. An exception is seen for two-second segments at 5Mbps, where HTTP/3 performs worse than HTTP/1.1 but better than HTTP/2.

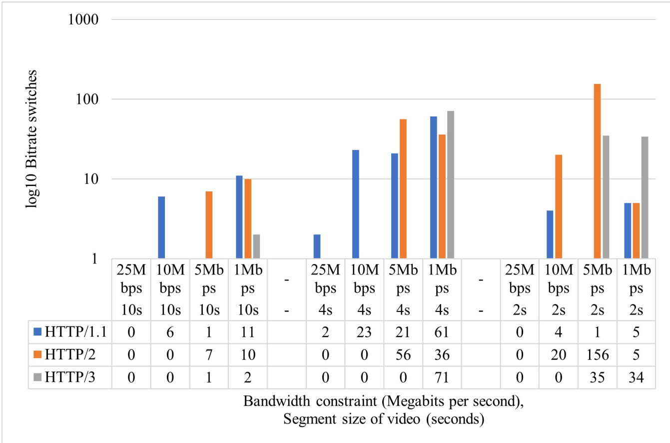


Figure 5.9: Relationship between the number of bitrate switches and bandwidth

Under almost all conditions in Figure 5.9, HTTP/3 undergoes fewer bitrate switches than the other two protocols. This is likely due to QUIC using stream multiplexing [33], which is an improvement over HTTP/1.1, and QUIC's stream prioritization appears to work better than HTTP/2 in these scenarios.

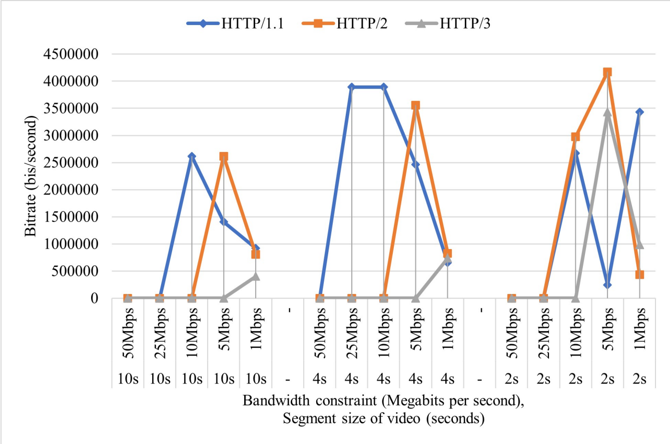
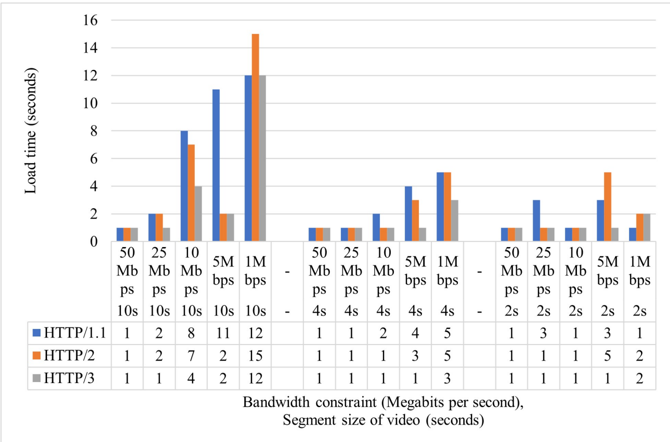


Figure 5.10: Relatinship between the video bitrate distribution streamed and bandwidth

The distribution of bitrates streamed shows that HTTP/3 undergoes significantly fewer bitrate switches than the other two protocols, with an exception for 5 Mbps bandwidth for two-second segments. This may be a circumstance where throughput for the protocol was lower than the desired amount for a higher quality but higher than necessary for a lower quality. This would then cause the adaptive-bitrate algorithm to switch between the two.



	50 Mbps 10s	25 Mbps 10s	10 Mbps 10s	5Mbps 10s	1Mbps 10s	- -	50 Mbps 4s	25 Mbps 4s	10 Mbps 4s	5Mbps 4s	1Mbps 4s	- -	50 Mbps 2s	25 Mbps 2s	10 Mbps 2s	5Mbps 2s	1Mbps 2s
HTTP/1.1	1	2	8	11	12		1	1	2	4	5		1	3	1	3	1
HTTP/2	1	2	7	2	15		1	1	1	3	5		1	1	1	5	2
HTTP/3	1	1	4	2	12		1	1	1	1	3		1	1	1	1	2

Figure 5.11: Relationship between rounded-up initial load time of video and bandwidth

The initial load time of video streams in Figure 5.11 shows that HTTP/3 performs significantly better than the other two protocols, providing a better quality of experience overall, as there are only five instances where initial load time was $>= 2$ seconds. As was discussed in 2, this is a significant indicator of good UX, as a user's likelihood of video abandonment increases rapidly from two seconds onwards.

5.1.3 Performance under packet loss conditions

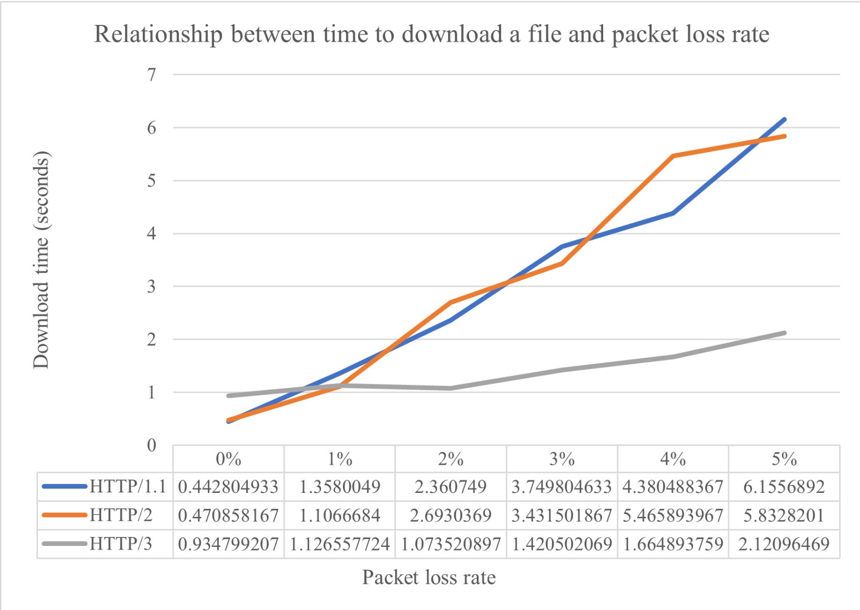


Figure 5.12: Mean time to download a 2 MB file using HTTP/3, HTTP/2, and HTTP/1.1 under different packet loss rates

Under packet loss conditions, HTTP/3 performs significantly better than the other protocols. This is due to how QUIC handles packet loss and its distinction from TCP [29, 36], which is discussed in Chapter 2. The results of these tests can be seen in Figures 5.14, 5.15, 5.18, and 5.12. These results show that at packet loss rates from 1% to 3%, packet loss has minimal effect on HTTP/3, whereas it causes severe degradation in HTTP/2 and HTTP/1.1.

5.1.3.1 Video Streaming Performance

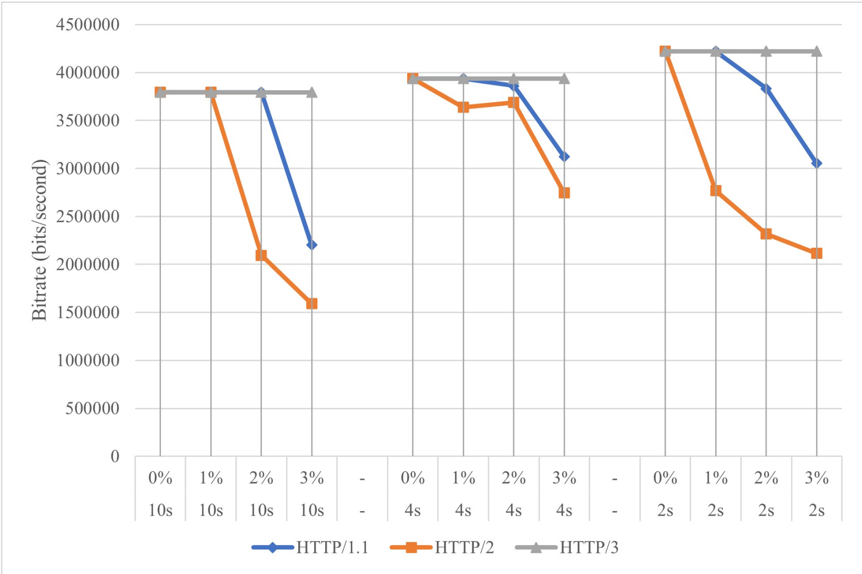


Figure 5.13: Relationship between mean video bitrate and packet loss rate

HTTP/2's performance in this category is noticeably poor compared to the other protocols. This is likely due to head-of-line blocking, which causes multiple streams to be interrupted concurrently, as discussed in Chapter 2. This problem is not present in HTTP/3, as QUIC uses per-path congestion control.

HTTP/1.1's performance is still poor, but not as bad as HTTP/2's performance. This can be explained both by HTTP/2's design of using a single TCP stream which may get blocked, and the fundamental design of TCP congestion recovery. QUIC's design states that QUIC implementations can reduce the congestion window immediately after a packet loss, as classic TCP NewReno does [29], or they can implement a less aggressive approach by using Proportional Rate Reduction [84]. This allows QUIC's congestion to reduce the congestion window less than TCP would in the same situation, which allows for higher throughput for QUIC than TCP.

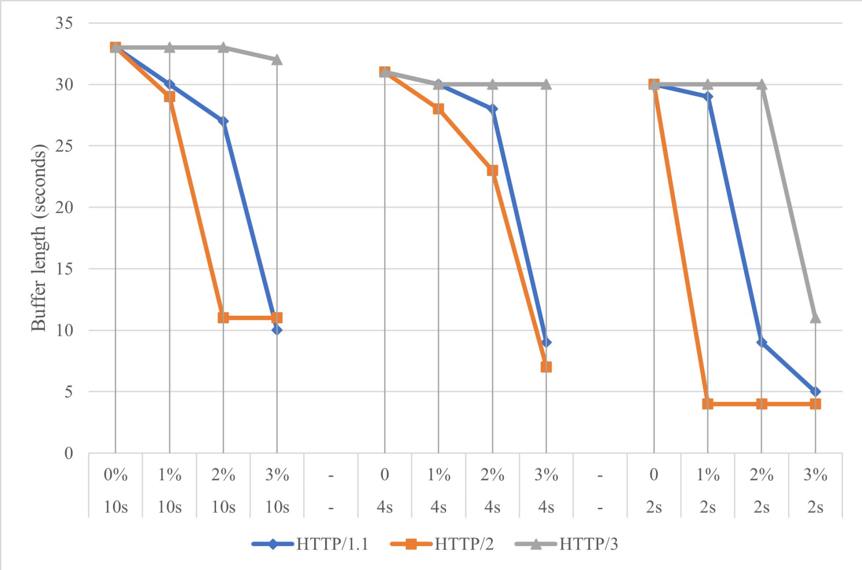


Figure 5.14: Relationship between median video buffer length and packet loss rate

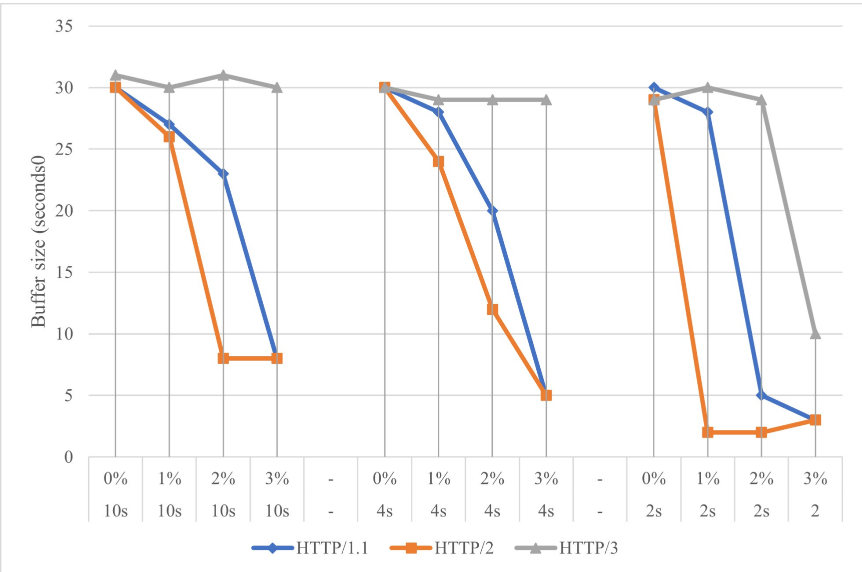


Figure 5.15: Relationship between the lower quartile of video buffer length and packet loss rate

The median and lower quartile of video bitrate in Figures 5.14 and 5.15, it can be seen that for HTTP/1.1 and HTTP/2, the buffer length drops dramatically as packet loss increases. HTTP/3 does not suffer from this problem, as *nginx*'s implementation of QUIC's congestion control is less drastic in responding to packet loss than standard TCP NewReno. This is discussed in Chapter 2.

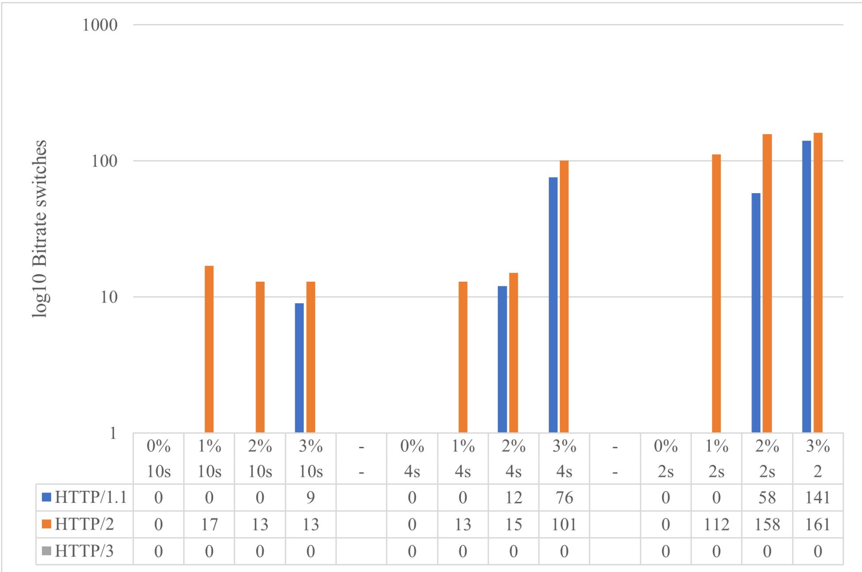


Figure 5.16: Relationship between the number of bitrate switches and packet loss rate

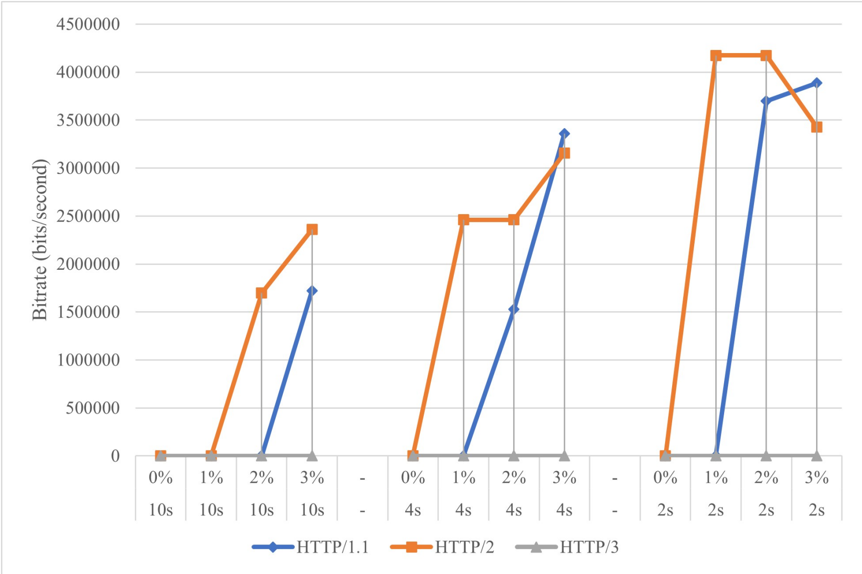
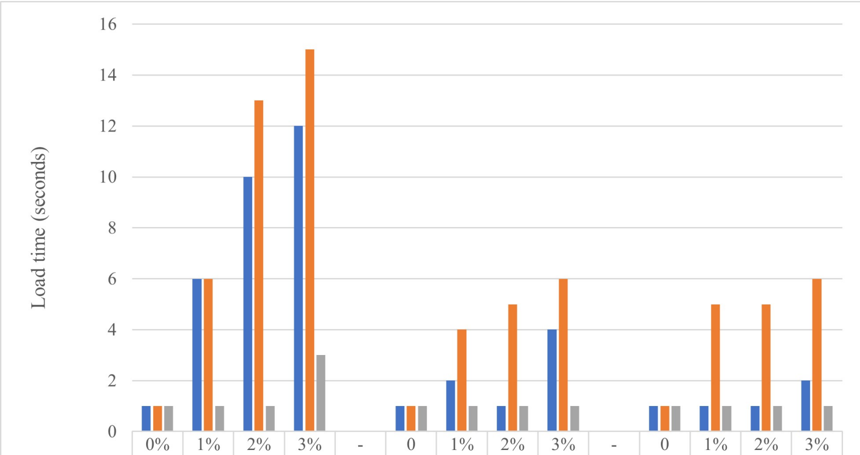


Figure 5.17: Relationship between video bitrate distribution and packet loss rate

The performance difference for HTTP/3 in the context of packet loss can be very clearly seen in Figures 5.16 and 5.17. HTTP/3 does not undergo any bitrate switches, and as such, the bitrate distribution is zero for all tested packet loss rates. HTTP/2 and HTTP/1.1 both undergo many bitrate switches, and the bitrate distribution is significant for both.



	0% 10s	1% 10s	2% 10s	3% 10s	- -	0 4s	1% 4s	2% 4s	3% 4s	- -	0 2s	1% 2s	2% 2s	3% 2s
HTTP/1.1	1	6	10	12		1	2	1	4		1	1	1	2
HTTP/2	1	6	13	15		1	4	5	6		1	5	5	6
HTTP/3	1	1	1	3		1	1	1	1		1	1	1	1

Figure 5.18: Relationship between the rounded-up initial load time of video and packet loss rate

As shown in Figure 5.18, HTTP/3's initial load time is significantly lower than HTTP/2 and HTTP/1.1 for all packet loss rates. This gives HTTP/3 a significant advantage regarding UX, as the initial load time is significantly lower as packet loss rates take effect. This is discussed in Chapter 1.

5.1.4 Performance under latency conditions

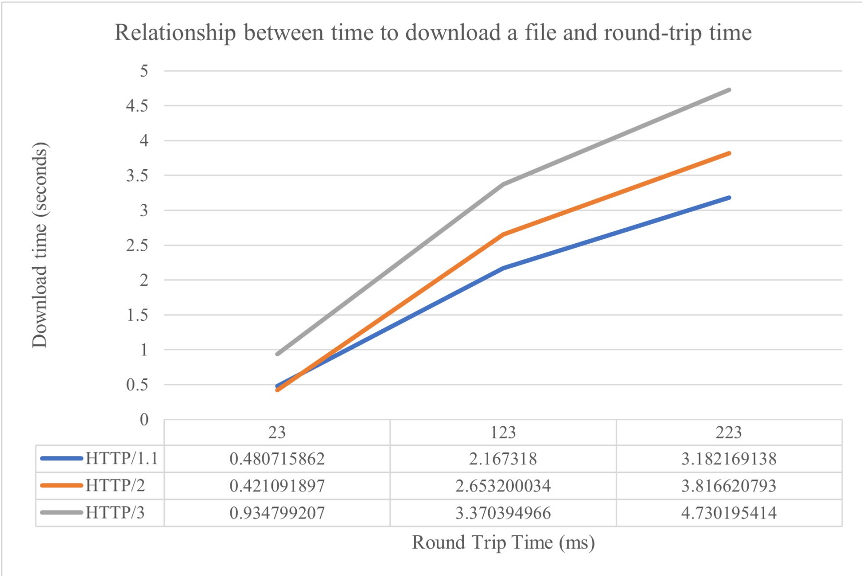


Figure 5.19: Mean time to download a 2 MB file using HTTP/3, HTTP/2, and HTTP/1.1 under different latency

When round-trip time is increased, HTTP/3 performs poorly compared with the other protocols. This can likely be explained by QUIC's design especially considering the congestion control method, which incorporates the average round trip time into the congestion window calculation [36]. This is discussed in Chapter 2.

5.1.4.1 Video streaming performance

Under all metrics and increased round-trip time, HTTP/3 performs poorly compared to the other protocols. The mean video bitrate, median buffer level, lower quartile of buffer level, and initial load time are all significantly worse for HTTP/3. The throughput in Figure 5.19 shows that HTTP/3 is the only protocol to significantly decrease throughput as latency increases.

This is likely due to implementing the congestion control algorithm in QUIC, specifically, the average round trip time used in calculating the congestion window [36]. As well as this, as congestion control is implemented on a per-path basis, a slow-start CWND is used for each path, which would cause the throughput to be lower than TCP, which uses a single CWND for downloads over the same connection. This is discussed in Chapter 2.

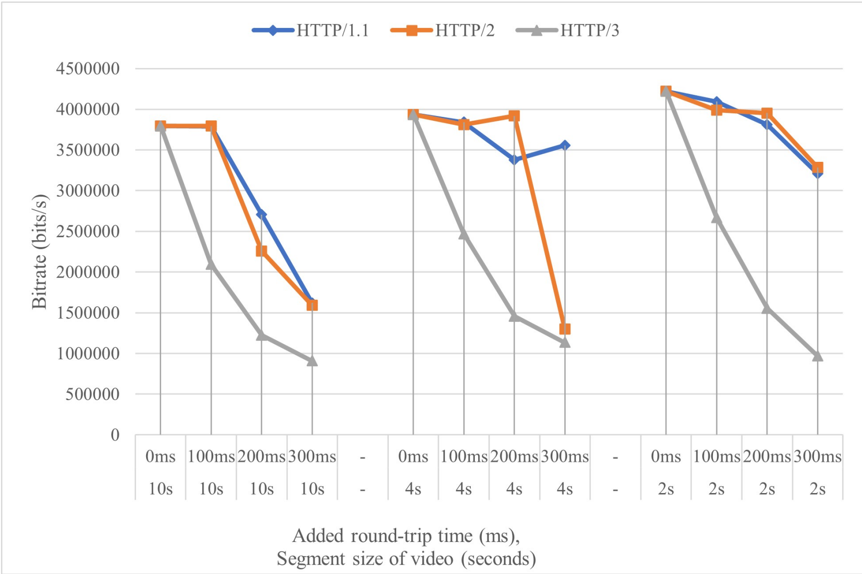


Figure 5.20: Relationship between mean video bitrate and latency

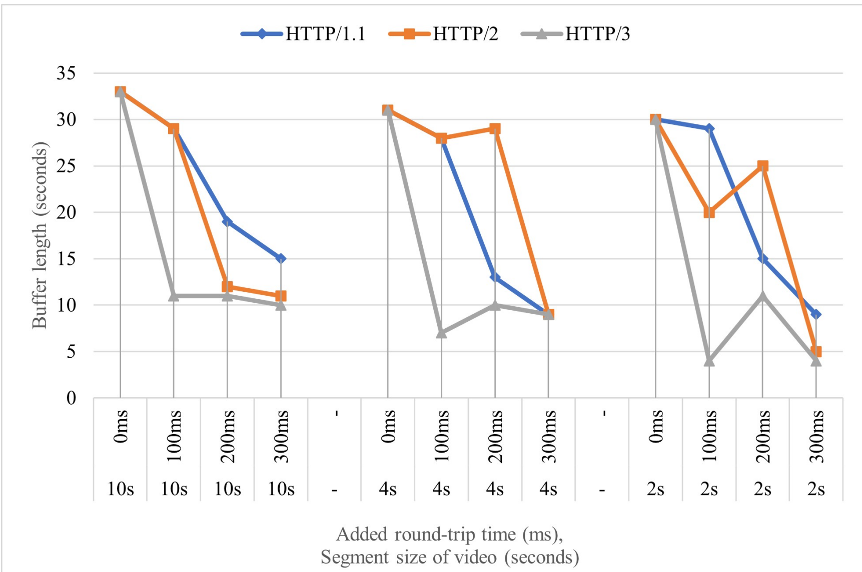


Figure 5.21: Relationship between median video buffer length and latency

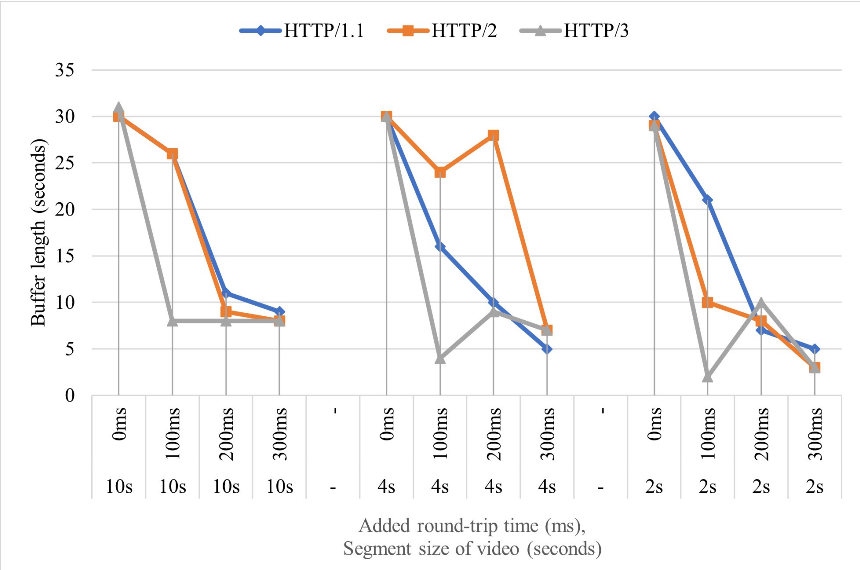
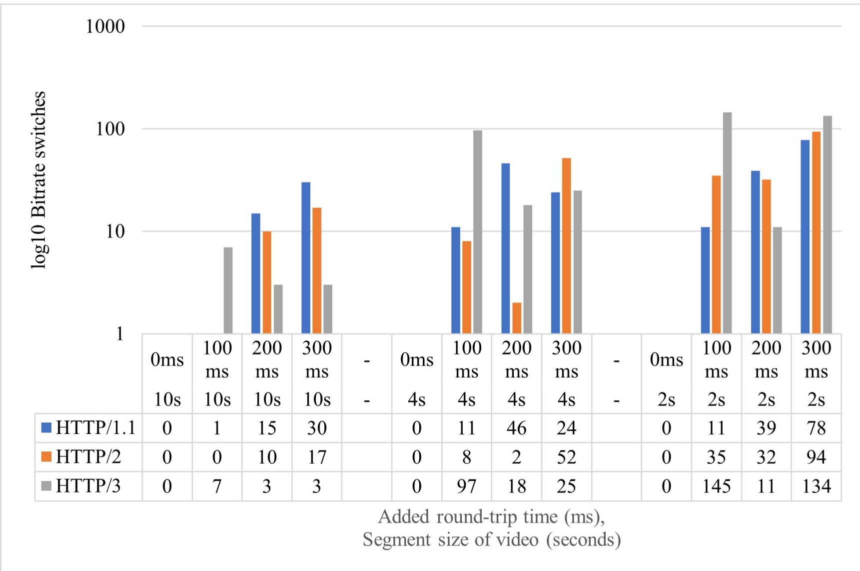


Figure 5.22: Relationship between the lower quartile of video buffer length and latency



	0ms 10s	100 ms 10s	200 ms 10s	300 ms 10s	- -	0ms 4s	100 ms 4s	200 ms 4s	300 ms 4s	- -	0ms 2s	100 ms 2s	200 ms 2s	300 ms 2s
HTTP/1.1	0	1	15	30		0	11	46	24		0	11	39	78
HTTP/2	0	0	10	17		0	8	2	52		0	35	32	94
HTTP/3	0	7	3	3		0	97	18	25		0	145	11	134

Figure 5.23: Relationship between the number of bitrate switches and latency

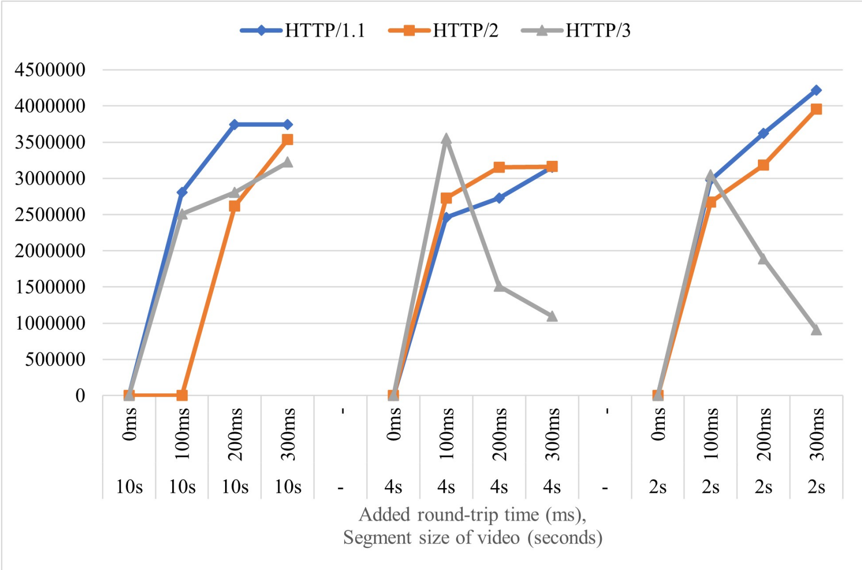


Figure 5.24: Relationship between video bitrate distribution and latency

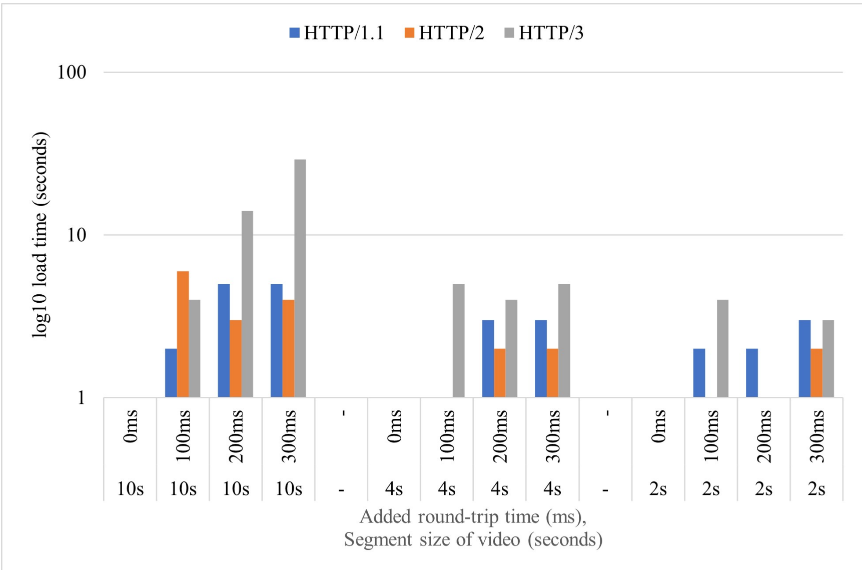


Figure 5.25: Relationship between the rounded-up initial load time of video and latency

Each graph (from Figures 5.20 to 5.25) shows that HTTP/3 performs worse than HTTP/2 and HTTP/1.1 under increased latency. The drop in video bitrate as latency increases is more significant for HTTP/3 than the others. The initial load time in Figure 5.25 is also significantly higher for HTTP/3 than the other protocols. This gives significantly worse QoE for HTTP/3 than the other protocols under increased latency.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

From the results discussed in Section 5.1, it can be concluded that HTTP/3 is a viable alternative to HTTP/2 and HTTP/1.1 for video streaming. This can be said as HTTP/3's performance was similar to the performance of the other protocols under differing bandwidth conditions.

Results also dictate that in most scenarios, HTTP/3 provides equal or better video streaming performance than other protocols, except where latency significantly impacts the network conditions of the client-server communication.

Video streams over HTTP/3 tend to have lower initial load times than HTTP/2 and HTTP/1.1. According to Akamai, this is an important metric to assess QoE, especially concerning video abandonment rates [3]. This indicates that HTTP/3 is an adequate protocol for video streaming compared to the previous HTTP versions.

6.2 Future Work

The work carried out during this project aims to investigate the performance of HTTP/3 for video streaming by comparing it with previous versions of HTTP. To adequately investigate this topic, more specific tests will need to be carried out to better outline where HTTP/3 lies compared to the other two protocols.

One such test would be to run multiple video streams concurrently over different protocols and investigate how fair the protocols are in sharing the available bandwidth. This is an important experiment to measure QUIC and TCP bandwidth contention.

Another test that could be carried out is to test the performance of QUIC when utilizing user space networking such as Data Plane Development Kit (DPDK) [85]. DPDK is a set of libraries and drivers for fast packet processing [85] done in user space. This would be an insightful test to see if the performance of QUIC can be improved by allowing direct access to the NIC instead of requiring syscalls to the kernel to process packets.

6.3 Personal Reflection on the Project

This project was the first academic writing project I have undertaken. Working on this project was challenging yet highly rewarding. As this project's subject requires a lot of background knowledge of the underlying protocols and other technologies, a large portion of time was taken to read through dozens of standards and research papers to understand the topics better.

The Internet Engineering Task Force (IETF) drafts and RFCs were especially useful for understanding the QUIC protocol and HTTP/3. The IETF drafts and RFCs were written in a very technical manner and were challenging to understand at times. However, after reading through a few drafts, I understood the structure of the documents and how to read them better. This was important, as each of the protocols, and many of the specifics surrounding different congestion control algorithms, etc., were defined in these documents.

One of the most challenging parts of this project was designing and conducting experiments to test the performance of the different protocols and ensuring that experiments were reproducible and that the results consistently proved challenging. As well as conducting the experiments, analyzing the results was also a difficult task, as there were many different metrics to consider. Another aspect of conducting experiments and analyzing data was that I had to go through multiple tools and methodologies before finding a tool that worked for me. Trying the tools explained in Chapter 4 took a significant amount of time to find a toolset that satisfied my use case.

I'm glad I had prior experience writing documentation in LaTeX, which allowed me to add any necessary markup for the report quickly. If I had not had prior experience with LaTeX, I would have had to spend significant time learning how to use it, which would have taken away from the time I had to work on the project.

This project was initially intended to be a software development project, where I would implement a video streaming service utilizing HTTP/3. A lot of time, work, and planning went into the system design requirements of a video streaming service, down to setting up a mesh of video encoding servers and a CDN. However, as time passed, the project scope changed dramatically and became an empirical research project comparing HTTP versions' performance. This was a difficult change, as I had to re-evaluate the project's goals and requirements and re-plan the project's timeline. As well as this, my personal experience and preference for software development and system design were not utilized in this project, which was a shame. However, it was worth the change, as I am glad that I was able to learn more about the underlying protocols and technologies that make up the Internet.

Bibliography

[1] N. Anderson, "Netflix offers streaming movies to subscribers."

[2] Cisco, "Cisco visual networking index: Global mobile data traffic forecast update, 2017-2022."

[3] . S. S. Krishnan and R. K. Sitaraman, "Video stream quality impacts viewer behavior: Inferring causality using quasi-experimental designs," in *Proceedings of the 2012 Internet Measurement Conference*, IMC 12, (New York, NY, USA), p. 211–224, Association for Computing Machinery, 2012.

[4] M. Tingley, "Streaming video experimentation at netflix:visualizing practical and statistical significance," *Netflix Technology Blog*, sep 2018.

[5] D. Raca, J. J. Quinlan, A. H. Zahran, and C. J. Sreenan, "Beyond throughput: A 4g lte dataset with channel and context metrics," in *Proceedings of the 9th ACM Multimedia Systems Conference*, MMSys '18, (New York, NY, USA), p. 460–465, Association for Computing Machinery, 2018.

[6] SimilarWeb, "Top websites ranking."

[7] I. J. S. 29, "Dynamic adaptive streaming over http (dash) — part 1: Media presentation description and segment formats," standard, International Organization for Standardization, 2022.

[8] D. I. Forum, "Github.com - dash-industry-forum/dash.js stargazers."

[9] D. I. Forum, "Dash.js adaptive bitrate algorithm directory." Available at: `https://github.com/Dash-Industry-Forum/dash.js/tree/43ce52870cbda1686ce81e9cf2b32c47d9b0e4ee/src/streaming/rules/abr`.

[10] R. Pantos and W. May, “HTTP Live Streaming.” RFC 8216, Aug. 2017.

[11] I. J. S. 29, “Generic coding of moving pictures and associated audio information — part 1: Systems,” standard, International Organization for Standardization, 2022.

[12] Y. S. at Twitch, “Live video transmuxing/transcoding: Ffmpeg vs twitchtranscoder, part i.”

[13] YouTube, “Choose live encoder settings, bitrates, and resolutions.”

[14] Apple, “Http live streaming.”

[15] Mozilla, “Evolution of http: Mdn.”

[16] R. T. Fielding, M. Nottingham, and J. Reschke, “HTTP Semantics.” RFC 9110, June 2022.

[17] T. Berners-Lee, “The http protocol as implemented in w3.”

[18] H. Nielsen, R. T. Fielding, and T. Berners-Lee, “Hypertext Transfer Protocol – HTTP/1.0.” RFC 1945, May 1996.

[19] H. Nielsen, J. Mogul, L. M. Masinter, R. T. Fielding, J. Gettys, P. J. Leach, and T. Berners-Lee, “Hypertext Transfer Protocol – HTTP/1.1.” RFC 2616, June 1999.

[20] Google, “Spdy: An experimental protocol for a faster web.”

[21] Google, “Spdy protocol.”

[22] R. Peon and H. Ruellan, “HPACK: Header Compression for HTTP/2.” RFC 7541, May 2015.

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

[24] M. Bishop, “Http/3,” 2022.

[25] D. Data, “Browser connection limitations.”

[26] W. Eddy, “Transmission Control Protocol (TCP).” RFC 9293, Aug. 2022.

[27] E. Rescorla and T. Dierks, "The Transport Layer Security (TLS) Protocol Version 1.2." RFC 5246, Aug. 2008.

[28] E. Blanton, D. V. Paxson, and M. Allman, "TCP Congestion Control." RFC 5681, Sept. 2009.

[29] A. Gurtov, T. Henderson, S. Floyd, and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm." RFC 6582, Apr. 2012.

[30] G. Fairhurst and T. Jones, "Transport features of the user datagram protocol (udp) and lightweight udp (udp-lite)." RFC 8304, Feb. 2018.

[31] R. Woundy and K. Kinnear, "Dynamic Host Configuration Protocol (DHCP) Leasequery." RFC 4388, Feb. 2006.

[32] G. Chromium, "Quic, a multiplexed transport over udp."

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

[34] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," Internet-Draft draft-ietf-quic-transport-00, Internet Engineering Task Force, 11 2016. Work in Progress.

[35] M. Nottingham, "Identifying our deliverables," oct 2018.

[36] J. Iyengar and I. Swett, "QUIC Loss Detection and Congestion Control." RFC 9002, May 2021.

[37] D. Gallatin, "The "other" freebsd optimizations used by netflix to serve video at 800gb/s from a single server."

[38] nginx, "Github.com - nginx/nginx."

[39] Netcraft, "Web server survey," Feb 2023.

[40] L. Crilly, "Introducing a technology preview of nginx support for quic and http/3."

[41] NGINX, "Milestone nginx-1.25."

[42] Google, "Github.com - google/boringssl."

[43] OpenSSL, "Openssl blog | quic and openssl."

[44] R. L. Alessandro Ghedini, “Http/3: the past, the present, and the future.”

[45] Cloudflare, “Github.com - cloudflare/quiche.”

[46] BiagioFesta, “Connections stop working with aes-ciphers because of missing key update.”

[47] heinrich5991, “Clienthello not identical when retry is received.”

[48] Google, “Github.com - google/quiche.”

[49] Envoy, “Github.com - envoyproxy/envoy.”

[50] QUIC-Go, “Github.com - quic-go/quic-go.”

[51] M. Seemann, “Throughput of quic-go.”

[52] Caddy, “Github.com - caddyserver/caddy.”

[53] Google, “Google chrome and chromeos additional terms of service,” 2021.

[54] SimilarWeb, “Similarweb - browsers,” Mar 2023.

[55] G. Chromium, “The chromium projects.”

[56] curl, “Github.com - curl/curl.”

[57] f5, “What is traffic shaping?.”

[58] Linux, “tc - manual - show / manipulate traffic control settings.”

[59] NGINX, “Nginx quic,” 2023.

[60] E. Rescorla, “HTTP Over TLS.” RFC 2818, May 2000.

[61] NGINX, “Nginx add header,” 2023.

[62] curl, “curl - libcurl,” 2023.

[63] C. M. Lonvick and T. Ylonen, “The Secure Shell (SSH) Connection Protocol.” RFC 4254, Jan. 2006.

[64] FFmpeg, “Ffmpeg.”

[65] GPAC, “Gpac.”

[66] Blender, "Big buck bunny."

[67] Facebook, "React."

[68] D. I. Forum, "Dash.js | github.com." Available at: `https://github.com/Dash-Industry-Forum/dash.js/tree/43ce52870cbda1686ce81e9cf2b32c47d9b0e4ee/src/streaming/rules/abr`.

[69] Vercel, "Next.js."

[70] Microsoft, "Typescript."

[71] Mozilla, "setinterval."

[72] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format." RFC 8259, Dec. 2017.

[73] MySql, "Mysql documentation."

[74] PlanetScale, "Planetscale documentation."

[75] go echarts, "Github.com | go-echarts/go-echarts."

[76] Matplotlib, "Matplotlib."

[77] Google, "Google sheets."

[78] Microsoft, "Microsoft excel."

[79] Framework, "Framework laptop framework laptop 13 (12th gen intel® core™)."

[80] iPerf, "iperf3 manual."

[81] T. kernel development community, "Page table isolation."

[82] torvalds/Linux, "Github | linux/net/ipv4/tcp.c."

[83] G. P. Zero, "Meltdown: Reading kernel memory from user space."

[84] M. Mathis, N. Dukkipati, and Y. Cheng, "Proportional Rate Reduction for TCP." RFC 6937, May 2013.

[85] Intel, "Data plane devleopment kit (dpdk) user guide."

Acronyms

CDN Content Delivery Network. 1, 53

DASH Dynamic Adaptive Streaming over HTTP. 1, 3, 4, 25, 26

HLS HTTP Live Streaming. 4

HTTP Hypertext Transfer Protocol. 1–3, 5–8, 12, 13, 22–26, 30, 33–35, 37, 40–42, 44, 45, 50, 51, 53

HTTP/1.0 Hypertext Transfer Protocol Version 1.1. 6

HTTP/1.1 Hypertext Transfer Protocol Version 1.1. 6, 8, 11, 18, 22, 24, 51

HTTP/2 Hypertext Transfer Protocol Version 2. 5–7, 11, 12, 14, 17, 22, 24, 33–37, 40–42, 44, 45, 50, 51

HTTP/3 Hypertext Transfer Protocol Version 3. 1, 2, 5, 7, 13, 16–19, 22–25, 30, 31, 33–38, 40, 41, 43–46, 50–53

QoE Quality of Experience. 1, 2, 50, 51

TCP Transmission Control Protocol. 5–12, 14, 15, 30, 34, 35, 40, 41, 43, 46, 52

TLS Transport Layer Security. 9, 14, 15, 18

TLSv1.3 Transport Layer Security Version 1.3. 9, 16, 22

UDP User Datagram Protocol. 11–14, 30, 34

UX User Experience. 1, 2, 39