Quantitative and Qualitative Analysis and Comparison of Current Transport Options for Live HTTP Streaming

Name: Martin Emanuel Koch

Student ID: 11105043

Program: Media Technology

1st examiner: Prof. Dr.-Ing Luigi Lo Iacono (TH Köln)

2nd examiner: Prof. Dr.-Ing Ulrich Reiter (TH Köln)

Deadline: 08 August 2019

Declaration:
I hereby confirm on my honour that this thesis is the result of my independent work. All sources and auxiliary material used in this thesis are cited completely.

Place, date, signature: _____

Fakultät für
Informations-, Medien-
und Elektrotechnik

Technology
Arts Sciences
TH Köln

Bachelor’s Thesis

Title: Quantitative and Qualitative Analysis and Comparison of Current Transport Options for Live HTTP Streaming

Reviewers:

- Prof. Dr.-Ing Luigi Lo Iacono (TH Köln)
- Prof. Dr.-Ing Ulrich Reiter (TH Köln)

Abstract: This thesis investigates current transport options of different HTTP versions and their impact on live HTTP streaming, particularly DASH. First, a theoretical basis is established, explaining concepts, workflow and technologies of HTTP-based live streaming. Context-relevant components of the network stack are discussed with a focus on transport protocols TCP and QUIC, leading to a comparison of HTTP/1.1, HTTP/2 and HTTP-over-QUIC (predecessor to HTTP/3). Finally, a testbed is designed, implemented and discussed to quantitatively evaluate the protocols’ overhead, impact on quality of experience in varying network conditions and their adaptation performance during changing bandwidths.

Keywords: HTTP Streaming, Live, DASH, QUIC

Date: 08 August 2019

Bachelorarbeit

Titel: Quantitative und Qualitative Analyse und Vergleich Aktueller Übertragungstechnologien von Live HTTP Streaming

Gutachter:

- Prof. Dr.-Ing Luigi Lo Iacono (TH Köln)
- Prof. Dr.-Ing Ulrich Reiter (TH Köln)

Zusammenfassung: Die vorliegende Arbeit untersucht aktuelle Transportoptionen verschiedener HTTP Versionen und deren Auswirkungen auf live HTTP Streaming, insbesondere DASH. Dabei werden zunächst Grundlagen des Live-Streamings über HTTP anhand von zentralen Konzepten, Arbeitsschritten und Technologien erläutert. Die in diesem Kontext relevanten Komponenten des Protokollstapels werden veranschaulicht, mit Fokus auf die Transportprotokolle TCP und QUIC, im Hinblick auf den folgenden Vergleich von HTTP/1.1, HTTP/2 und HTTP-over-QUIC (Vorgänger zu HTTP/3). Im Anschluss dazu wird ein Testbett erstellt, welches quantitative Unterschiede der Netzwerkprotokolle bezogen auf Verwaltungsdaten, Auswirkungen auf “Quality of Experience” in beeinträchtigten Netzwerken und die Anpassung an schwankende Internetgeschwindigkeiten, untersucht.

Stichwörter: HTTP Streaming, Live, DASH, QUIC

Datum: 7. August 2019

Table of contents

1. Introduction **1**
2. HTTP and Streaming Media **2**
2.1. Progressive Download 3
2.2. Single Bitrate Streaming 3
2.3. Adaptive Bitrate Streaming 3
3. OSI Model **5**
3.1. Transport-oriented Layers 6
3.2. Application-oriented Layers 7
4. Transport Options **9**
4.1. Transmission Control Transport - TCP 9
4.1.1. Connection Management 9
4.1.2. Reliable Data Transfer Mechanisms 10
4.1.3. TCP Head-of-Line Blocking 12
4.1.4. Header Format 12
4.2. QUIC 13
4.2.1. Connection Establishment 14
4.2.2. Connection Migration 15
4.2.3. Multiplexing Without Head-of-Line Blocking 15
4.2.4. Congestion Control 16
4.2.5. Header Format 16
5. HTTP **17**
5.1. HTTP/1.1 17
5.2. HTTP/2 18
5.3. HTTP/2-over-gQUIC (Predecessor to HTTP/3) 18
5.4. Summary 19
6. HTTP Adaptive Streaming: Workflow **20**
6.1. Encoder 20
6.1.1. Transcoding 20
6.1.2. Segmentation and GOP 22
6.1.3. Container 23
6.2. Deployment 25
6.2.1. Advantages of Streaming via HTTP 25
6.2.2. Comparison to Non-HTTP Live Streaming 25
6.3. Playback 26

6.3.1. HTML5 26
6.3.2. Playout Buffer 28
6.3.3. Rate Adaption 29
6.4. Live vs. VoD 29
7. Dynamic Adaptive Streaming over HTTP - DASH 31
7.1.1. Background and Scope 31
7.1.2. Media Presentation Description 31
7.1.3. Other HAS Protocols 32
8. Quality Monitoring of HTTP Streaming 34
8.1. Key Performance Indicators of QoS 34
8.2. Key Performance Indicators of QoE 34
9. Testbed 36
9.1. Related Work 36
9.2. Test Setup 36
9.2.1. Preparation of the Test-data 38
9.2.2. Server Setup 39
9.2.3. Network Setup 40
9.2.4. Client Setup 43
9.3. Test Cases 44
9.3.1. Network Overhead of a Streaming Session 45
9.3.2. QoE in Networks of Varying Bandwidth, Delay and Loss 46
9.3.3. Adaptation Performance 47
9.4. Evaluation 48
9.4.1. Network Overhead of a Streaming Session 48
9.4.2. QoE in Networks of Varying Bandwidth, Delay and Loss 49
9.4.3. Adaptation Performance 55
Conclusion and Outlook 56
Appendix 57
Sources 58
List of Figures 62
List of Tables, Codes, Formulas 63
Eidesstattliche Erklärung 64

1. Introduction

Video streaming is everywhere and yet its use is still increasing: In late 2018 Sandvine's global internet report [SGIR] estimated video streaming to account for 58% of all internet traffic and is expected to rise up to 82% by 2022. While over-the-top[1] (OTT) services like Netflix[2] or Hulu[3] mostly produce on-demand content, social media platforms like Facebook[4] have begun offering live video streaming services via the Hypertext Transport Protocol (HTTP), with the most popular streams reaching viewerships similar to major sporting broadcasts.

Meanwhile, the internet and its underlying protocols evolve quicker than ever. A fully standardised version of the internet protocol HTTP (version 1.1) was introduced in 1997 and remained almost unchanged until recently. When a minor revision of the standard in 2014 did not manage to alter basic design problems of the protocol, Google had already developed and was one year from deploying their own better network protocol SPDY (pronounced "speedy"). Widely observed performance improvements have led the Internet Engineering Task Force (IETF) to standardise Google's protocol into HTTP/2 (only one year after republishing HTTP/1.1), which today powers more than a third of the world wide web. Yet, Google remained unsatisfied with the state of internet performance and decided to replace the hitherto underlying transport technology, Transmission Control Protocol (TCP), with their own protocol Quick UDP Internet Connection (QUIC). HTTP-over-QUIC, as SPDY earlier, is now being standardised and shall be released as HTTP/3, in late 2019. Nevertheless, Google's version is already deployed and powers most traffic between Chrome browser and services of Google, e.g., Youtube.

Today many HTTP versions coexist, leaving streaming providers to decide upon their means of deployment. Unfortunately, the performance implications are not always obvious. For HTTP-based streaming protocols can only reflect the technology they are built upon, this work will attempt to explain the fundamentals of live HTTP streaming and its relation to underlying network protocols. This knowledge is used in a testbed, quantitatively measuring performance differences in a simplified live-streaming environment. The goal of this thesis is to explain and demonstrate current possibilities of network protocol optimisation in the context of live HTTP streaming.

Cologne, August 2019

---

[1] i.e. video delivered via normal internet infrastructure

[2] https://media.netflix.com/en/about-netflix

[3] https://www.hulu.com/press/about/

[4] https://www.facebook.com/facebookmedia/solutions/facebook-live

2. HTTP and Streaming Media

Originally, developed by Tim Berner Lee at CERN in 1989, to exchange research papers in laboratories, HTTP has grown to become the main protocol for communication on the internet. It was designed to be connectionless, media independent, stateless and extensible and has evolved greatly since it was first seen. Despite the fact that other protocols, to some extent, provide better properties for the delivery of media—unsurprisingly since HTTP focuses on hypertext documents—the worldwide infrastructure and support of HTTP were too powerful for media distribution companies not to take advantage. How streaming outside of HTTP compares will not be the focus of this work but rather how differently nuanced HTTP versions impact streaming over HTTP.

Streaming media describes the distribution and playback of media[5] in which the end-user constantly receives and plays back content from a streaming provider. If the last mile[6] of the stream delivery is accomplished with standard HTTP servers instead of dedicated streaming servers, it is commonly referred to as HTTP streaming. In the client-server computing model, HTTP implements a request-response protocol. A simple example: A computer hosts a website, this is the server. Another computer uses a web browser to access this website, this is the client. To access the website, the client sends a HTTP GET request to the server in which the HTTP resource is wanted by using Uniform Resource Locators URLs[7] [WIK2]. The server looks for the requested resource and (in this case) responds with an HTML[8] file which contains the requested website.

Even though the protocol has originally not been intended for media delivery, innate extensibility and media independency have allowed three conceptual models of streaming over HTTP to emerge which will be discussed in the following Sections 2.1-2.3 [THb]:

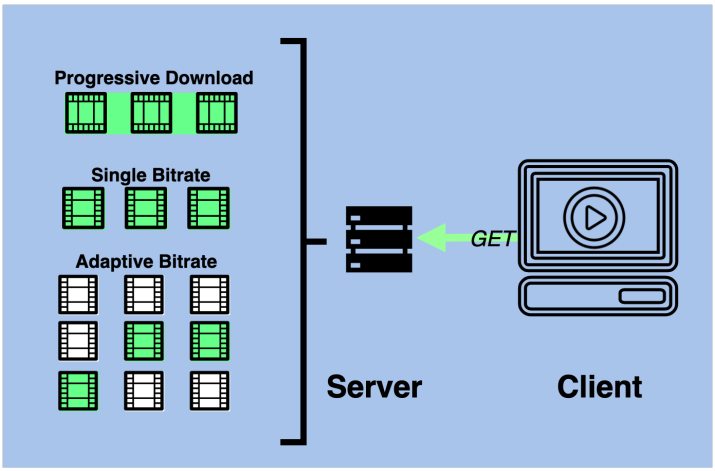

Figure 1: HTTP streaming technologies, the green underlay implicates separately downloadable files

[5] streaming media will be used synonymously with streaming video in this thesis

[6] last in the series of processing steps in video streaming

[7] In practice, the Uniform Resource Identifiers schemes http or https need to be pre-pended.

[8] HTML HyperText Markup language is descriptive language to structure of webpages. https://developer.mozilla.org/en-US/docs/Glossary/HTML

2.1. Progressive Download

To an end-user, progressive download, also referred to as “pseudo” streaming, is only distinguishable from “true” streaming by the inability to seek beyond the already downloaded content and by saving the video to storage instead of temporary memory [WkP]. It is illustrated in Figure 1, that in progressive download, data is stored in a single file and can therefore only be downloaded as a whole.

2.2. Single Bitrate Streaming

In single bitrate streaming, the client might start playing at an arbitrary point without having downloaded earlier parts of the video. This becomes possible as the media was previously split into chunks/segments, out of which each represents a point-of-entry for download and in consequence playback. Conceptually, single bitrate streaming works as follows:

A client opens a website with an embedded video-player. Once loaded, the video player retrieves meta information on the video, such as the preview-image, the quality level, the length of the clip and where to find the segments for a given time interval. The client could arbitrarily select a point of time for which the player starts retrieving the closest segments of the clip via HTTP GET requests[9]. As the player receives the responses, segments are processed and assembled to fill the playback buffer[10] and finally played back.

To playback without any stalling[11], the buffer must never run empty, i.e., the average buffer drainage[12] needs to be smaller than the buffer fill[13]. This strongly depends on many factors like the clip’s bitrate, the available network throughput, the length of a media segment, etc., and needs to be finely tuned to provide a stall-free playback. Tuning single-bitrate-streaming is problematic for it cannot provide an optimal streaming experience to clients with varying internet speed levels. If the stream were to be provided in a low quality, all clients would be able to stream but clients with high bandwidth internet connections would be unsatisfied. They would demand high quality streams which in turn would exclude low bandwidth clients. This left stream providers in compromise between video quality and stalls resulting in a suboptimal experience for a majority of their clients.

2.3. Adaptive Bitrate Streaming

Adaptive Bitrate Streaming is an approach to hitherto optimisation dilemmas extending on idea of single bitrate streaming: Before distribution, media is temporarily divided into same-length segments which are then encoded at different video quality levels (VQL). Similarly to single bitrate streaming, clients reproduce the media content by downloading segments,

---

[9] The way of retrieving data via HTTP.

[10] The media buffer stores already downloaded but not yet played media

[11] i.e. unwanted stop or pause of playback

[12] The buffer is emptied when the media is played out.

[13]The buffer is filled when new segments are downloaded successfully.

however now segments of the same time interval exist on multiple quality levels. The player approximates the current internet speed and CPU usage[14] and frequently requests segments of an adequate quality level[15] to continue playback without re-buffering[16]. Segmented media could be made available as individual files or in one file where segmentation is at ranges of bytes[17]. Standard HTTP GET request will be used to retrieve segments stored as individual files (Figure 1) and a HTTP range header is used for byte wise serving[18].

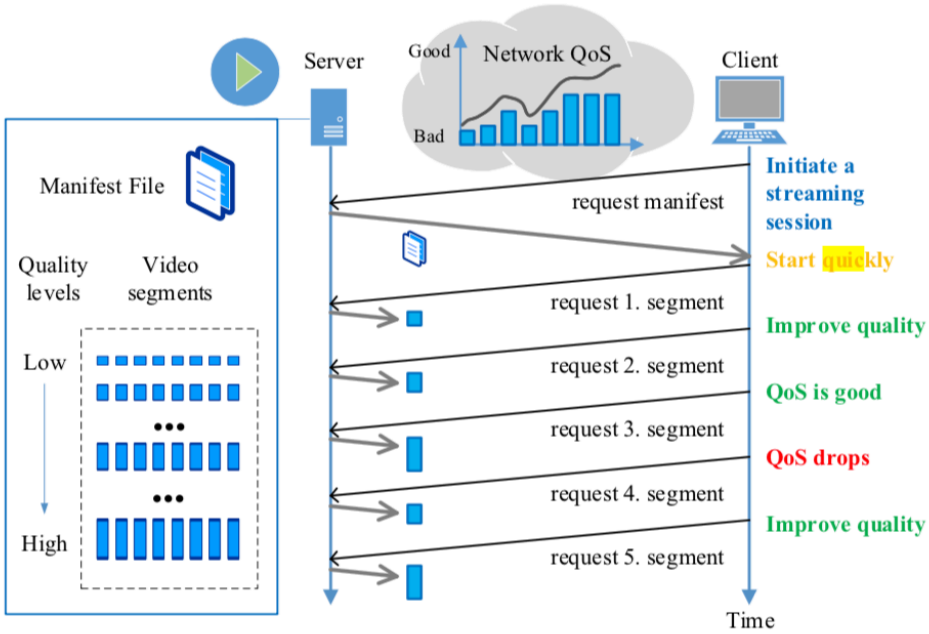

Figure 2: HTTP Adaptive Streaming concept by [BUF]

Independent from the individual network resources, clients are provided with their best individual experience, i.e., clients with high bandwidth internet access request high quality segments, and vice versa. This significantly increased the quality of streaming services at the cost of resources and complexity. Nowadays the concept of adaptive streaming is almost exclusively implemented over HTTP, namely HTTP Adaptive Streaming (HAS). In order to understand how different HTTP versions could take effect on HAS, the underlying protocols will be examined with the help of the OSI model in the following section.

---

[14] If the CPU cannot process the file fast enough frames might be dropped. This is considered as well as the internet speed for adaptation.

[15] file size

[16] the process of waiting for media to be retrieved until playback can resume

[17] E.g. segment 1 could be from Byte 0-500, segment 2 from Byte 500-1000. A request asking for “Range: bytes=500-1000” would request the second segment.

[18] This is only used in on Demand scenarios and shall not further be discussed.

3. OSI Model

The Open Systems Interconnection (OSI) model is a reference model which describes the communication, i.e., data exchange of applications over a network. Clear definitions of structure and processes enable interoperability of protocols as well as modelling and tracing of data flow between sender and receiver. In the OSI model network communication services are abstracted into 7 layers, out of which layers 1-4 are transport-oriented and layers 5-7 application-oriented. The core concept of abstraction allows the encapsulation into layered services. As a result, higher layer protocols use services of the next lower protocol while providing services to the next layer up. A practical example of encapsulation is the separation of transport and (logical) addressing.

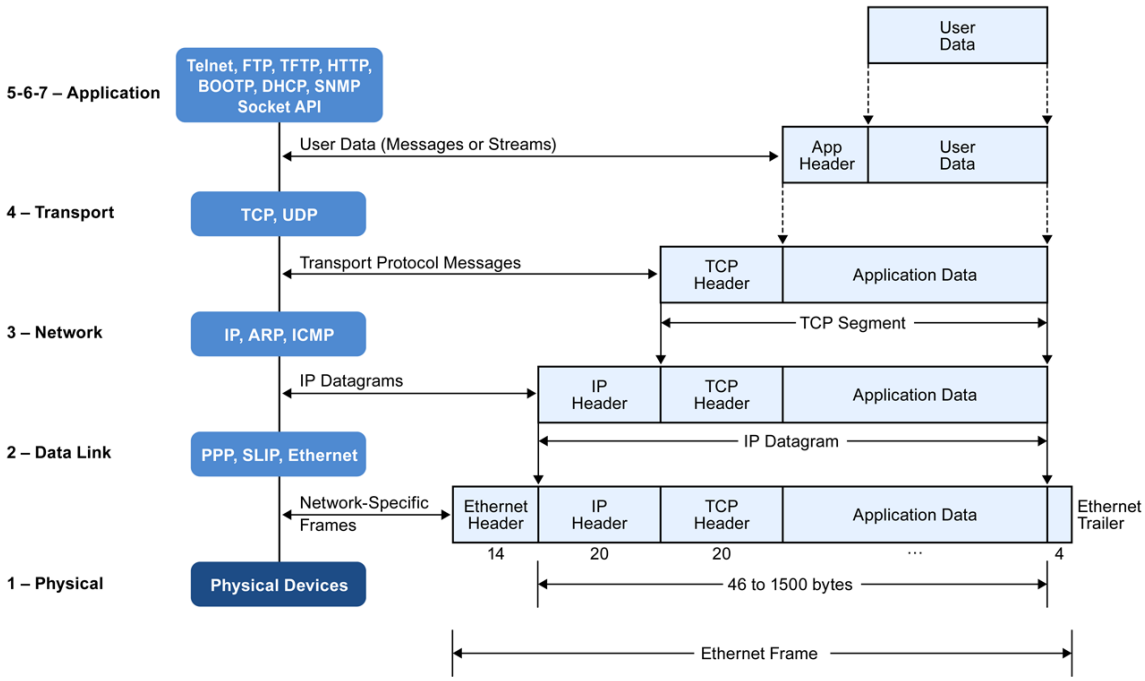

Figure 3: Network overhead, encapsulation and the OSI model, Zhang[19]

In order to achieve encapsulation, the actual data is enriched with meta data to prepare for transport. Starting from the highest layer on the sender-side, header- and trailer information are added to the original message (user data) and passed down to the next layer. This process is continued until the lowest layer in which bit-transfer between sender and receiver occurs. The overhead added by each layer is also referred to as protocol data units (PDUs). The process of encapsulation implies that lower layers cannot distinguish previously added PDUs from original data. After the transmission between the hosts, the receiving side ingests information at the lowest level. PDU information is processed and removed from the data of the current layer and then passed upwards until same layer where the original message was sent from is reached. [NET1] [Osi2]

[19] image source: http://www.cs.wayne.edu/fengwei/16sp-csc5991/slides/lab1-slides.pdf

3.1. Transport-oriented Layers

The lowest three layers are known as media layers and are concerned with actually moving the data between hosts. Together with the fourth layer they make the 4 transport-oriented layers:

Layer 1: The Physical layer sends and receives bits via different types of communication media. The PDU (the unit of data it processes) is a bit or signal.

Layer 2: The Data Link layer translates messages from the network layer into bitstreams which are then transmitted by the physical layer. This does not only involve hardware addressing, via the MAC address (Media Access Control), to send the messages to the correct devices on the local network, but also includes Logical Link Control (LLC) handling error notifications, network topology, and flow control. The PDU is a frame and usually has a maximum amount of data it can transfer at once. Maximum Transmission Unit (MTU) defines the maximum size of a message that can be carried by one data link transaction and is related with protocol performance. Theoretically, a higher MTU leads to greater throughput efficiency but causes higher network delays. Practically the MTU is limited by the smallest MTU in the path of delivery as overly sized frames will be discarded by the receiving device.

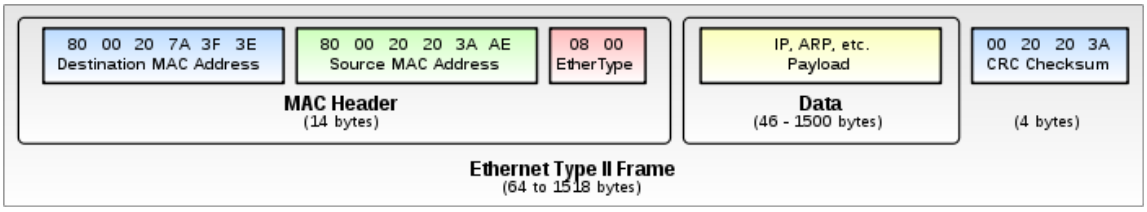

Figure 4: Ethernet header, wikipedia[20]

E.g. the most common ethernet frame Type II has a maximum frame size of 1518 bytes. The header consists of the hardware address of both destination and source which occupy 6 bytes each, plus 2 bytes for the frame type in use. Including the trailer with a CRC checksum of 4 bytes creates a total overhead of 18 bytes. This leaves a maximum packet size of 1500 bytes and defines the MTU for the network layer.

Layer 3: The network layer logically connects clients (routing) and is mainly responsible for packet forwarding. The most prominent network protocol is the internet protocol IP which defines a packaging and addressing system. The PDU is a IP datagram or packet.

[20] image source: https://en.wikipedia.org/wiki/Ethernet_frame

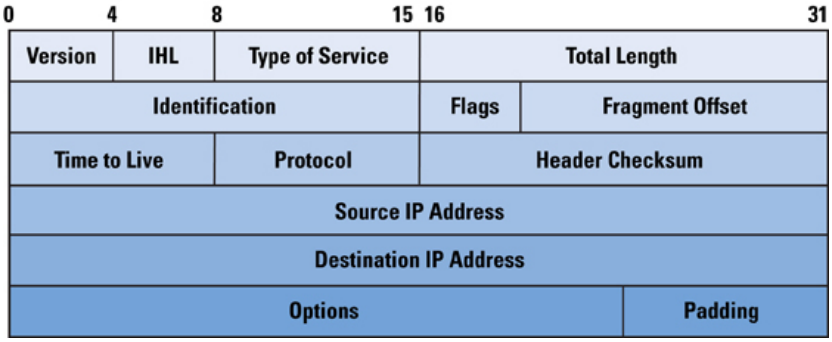

Figure 5: IPv4 header, Cisco[21]

Today two versions of IP coexist. The original IPv4 (version 4) is slowly being replaced by the IPv6 (version 6) which wants to improve the address shortage of its predecessor. IPv4 adds a fixed overhead of 20 Bytes which can exceed up to 60 bytes with extensions and IPv6 only provides a lower bound of 40 bytes [NET2].

Layer 4: The transport Layer manages packetization of data and the delivery of packets. Transport Control Protocol TCP and User Datagram Protocol UDP are the most prominent protocols today but many others exist.

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Length																Checksum															

Figure 6: UDP header, wikipedia[22]

UDP is a connectionless protocol, which is often used by time sensitive protocols. This is because it does not implement any mechanisms of reliability, hence it is fast but does not guarantee delivery. Fig. 15 shows the header format which has a total length of 8 bytes, consisting of source and destination port, the header length and a checksum for error correction at 2 bytes each. TCP and QUIC implement reliable communication over unreliable channels which will be explained in Sections 4.1-4.2 [NET3].

3.2. Application-oriented Layers

The upper three layers of the OSI model make up the application-oriented layers, the PDU is the data itself.

Layer 5: The session layer establishes, manages and ends sessions between two hosts. E.g. the most prominent Transport Layer Security TLS (former Secure Sockets Layer SSL) allows hosts to exchange cryptographic keys to secure the communication channel. HTTP can use TLS to securely communicate on the internet and the keys for encryption are usually exchanged after a connection to another host has been established.

---

[21] image source: https://www.cisco.com/c/en/us/about/press/internet-protocol-journal/back-issues/table-contents-38/104-ip-spoofing.html

[22] image source: https://en.wikipedia.org/wiki/User_Datagram_Protocol

Layer 6: The presentation layer translates data between the application and the networking service. This involves character encoding, data compression and encryption.

Layer 7: The application layer is the top-most layer and a high level API with which the end-user interacts to access network resources. This layer allows resource sharing or remote access over network.

HTTP is the most prominent application layer protocol due to its usage in the world wide web. HTTP messages consist of requests or responses, each containing information for transport, the header, and the data to transmit, the body. There are different ways of packaging, organising and sending these messages. As an application protocol, HTTP uses lower layer protocols for example for transport or encryption. A widespread misconception is the requirement of HTTP messages to be delivered via TCP. In fact HTTP only requires a reliable transfer protocol. Section 4 will explain transfer protocols TCP and QUIC and Section 5 how different HTTP version, process HTTP messages to send them over the network.

4. Transport Options

In September 1981 the IETF published the first version of TCP in RFC 793[23]. In contrast with UDP, standardised in the previous year, it implements mechanisms to assure the delivery of packets (at the cost of speed). Until now most of the internet is powered by the so-called TCP/IP stack which refers to using the combination of the TCP and the IP protocol for network communication. TCP is a long lived and fully adopted protocol. Relying on this persistence, companies installed middle-boxes which forwarded traffic to analyse issues, improve quality of service or to enforce policies. Consequently, updates or changes to the way internet protocols work must not only consider protocol interactions but also all the middle-boxes interfering with network traffic. The rigidity caused by middle-boxes is often referred to as ossification and caused optimisation attempts like SCTP[24] to fail. In 2012 Google released Quick UDP Internet Connections (QUIC), a protocol built on top of UDP to replace TCP as means of transport for HTTP/2.

This section will give an overview of TCP and QUIC with HAS in mind. Both transport protocols abstract a reliable network over unreliable channels. Network transmission almost exclusively occurs via best-efforts networks, i.e., packets can be lost, corrupted, out of order, etc. In order to fulfil previously mentioned guarantees, they establish a connection and implement mechanisms of flow control, error correction, loss recovery and how to deal with congestion.

4.1. Transmission Control Transport - TCP

TCP is a connection-oriented end-to-end protocol which provides reliable, in-order delivery of data. It abstracts the network connection behind a network socket interface which is accessible by the application layer to send and receive data, e.g., World Wide Web, email, etc.

4.1.1. Connection Management

To exchange data a connection needs to be established. In TCP, this connection is established via the 3-way-handshake[25]:

1. HOST A sends a **SYN** packet to a remote host, telling HOST B that it wants to open a connection. This packet contains a randomly picked *sequence number x* and some TCP options.
2. To accept, HOST B replies with a **SYN-ACK** packet that confirms the receipt of the SYN packet by replying with *x+1* and also sends a randomly picked sequence number *y*.

---

[23] The IETF publishes “Request For Comment” or RFC, documents which contain formal internet standards. This one concerns TCP. https://www.ietf.org/rfc/rfc793.txt

[24] Stream Control Transmission Protocol is a protocol for transmitting multiple streams of data at the same time between two end points that have established a connection in a network https://en.wikipedia.org/wiki/Stream_Control_Transmission_Protocol

[25] It is assumed that HOST B is listening on a port accessible by HOST A

3. Once HOST A received this, a final **ACK** is sent**,** containing both incremented sequence numbers *x+1* and *y+1.* Now a TCP socket connection is open and data exchange can begin.

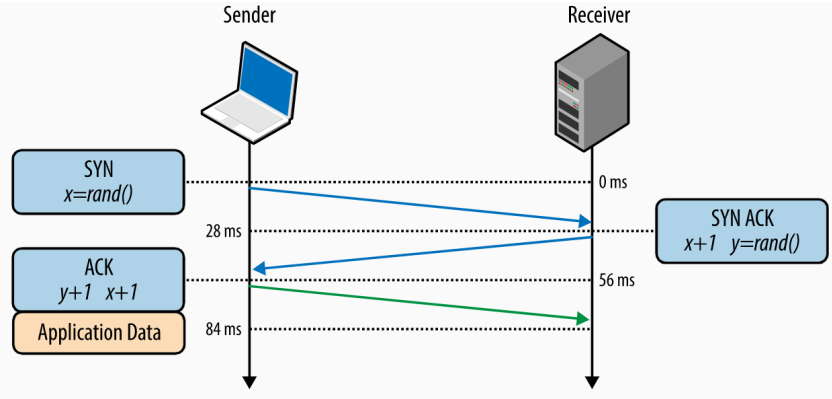

Figure 7: 3-way-handshake in TCP [hpbn1]

As a result, before any application data is transmitted, the 3-way-handshake causes a delay of 1 Round Trip Time RTT (to the receiver and back). Therefore TCP connection become very expensive in networks with high delays (the time interval for propagating a packet from a client to a server). TCP *Fast Open* is a new mechanism which allows sending data in the SYN packet for repeat connections but only slowly finds adoption. Nevertheless, the optimised handling of TCP connections remains crucial to maintain the performance of web applications in all network conditions.

4.1.2. Reliable Data Transfer Mechanisms

Like UDP, TCP incorporates similar mechanisms to deal with data corruption caused by imperfect transmission. It performs an additive check for both header and payload. The algorithm works in the same simple principle (*Ones-Complement-Arithmetic*) for UDP and TCP. Checksums partially correct data corruption within a packet but neither recover nor detect packet loss. To make sure that packets arrive, they require identification. To achieve this the initially agreed-upon sequence number *x/y* (Figure 7) is monotonically increased and placed in the TCP header. Receipt of a packet requires the receiver's confirmation by sending an acknowledgement with the sequence number. When a segment of data is send, a retransmission timer of variable length is started, the *retransmission timeout (RTO)*. If no acknowledgment is received within this timeframe, a retransmission is triggered. If the receiver notices loss, or out of order receipt, in the simple way, a *duplicate acknowledgement DUP ACK* tells the sender up to which sequence number, packets had been received, implicitly requesting the retransmission of higher number segments. With *TCP Fast Retransmissions* segments which arrive out-of-order (#4 - #2 - #3) are better dealt with, using selective acknowledgements SACKs. SACKs not only inform the sender about the missing segments but also about the ones which have been received which increases efficiency.

These mechanisms ensure reliability but would make a congested network collapse without any rate control. This is because a busy (best-effort) network would simply discard of packets if it becomes overwhelmed. Once time passes the RTO, retransmission is invoked

which would result in the TCP senders flooding the network with retransmissions. The following mechanisms avoid this and other issues related to the sending rate [hpbn1] [GreT]:

Flow Control

Flow control manages the rate at which data is transmitted. Sending all data at once could overwhelm the receiver processing speed, data would be unnecessarily lost and would require retransmission. TCP uses a *sliding window* algorithm to limit the amount of data send on the network without respective acknowledgements by the receiver. Apart from acknowledging in the ACK message, the receiver additionally signals how much data it can currently process with the *receiver window rwnd in the TCP header*. Once the sender adjusted the sliding window accordingly, the window is slid forward to transmit the next TCP segments.

Congestion Control

While flow control makes sure that the receiver can handle the amount of data transmitted, congestion control (CC) prevents senders from overwhelming the network. CC manages the size of the *congestion window*[26], i.e., limiting how much data is sent over the network without acknowledgement. There is a plethora of algorithms for CC, almost exclusively with the similar principle of using observed loss (and delay) as device to regulating the sending rate.

Additive Increase Multiplicative Decrease AIMD is a congestion control algorithm, which converges to providing network resource fairness[27] at optimal network utilisation. The idea is to additively increase the sending rate for every acknowledgement received and to multiplicatively decrease at the detection of loss. This assumes that loss is exclusively due to congestion.

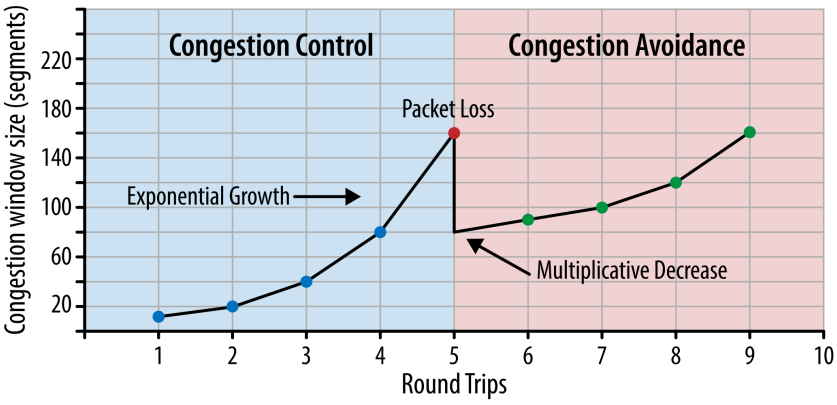

Figure 8: TCP Congestion Control and Congestion Avoidance, [hpbn1]

The initial phase of TCP congestion control is called *Slow Start and* describes the technique of exponentially increasing the sending rate until network saturation is reached(the naming refers to the prior approach of dump-sending a fixed amount of data after start-up). After the

[26] In practise, optimal congestion window and receiver window depend on the network delay and target bitrate and are calculated with the Bandwidth-Delay Product.

[27] In a network without priorities, fairness means everyone has the same bandwidth. Early CC-free versions of transport protocols did not have problems with congestion because the network was undersaturated. Providing bandwidths which would not saturate at the internet scale today would be lavish, hence the need for CC.

congestion window grows beyond the threshold *ssthresh*, TCP enters a *congestion avoidance phase,* using the earlier described AIMD.

Some algorithms, like *TCP Tahoe*, set the congestion window to 0 when any loss is detected during slow start. This can be inefficient because the loss of one packet does not necessarily imply network saturation. The most commonly used algorithm, *New Reno*, uses fast retransmissions and fast recovery, which perform better by managing retransmissions within a single sending window [hpbn1]. Other approaches like *TCP Vegas* or *BBR* improve CC-efficiency in high bandwidth networks, by using network delay and bandwidth estimations rather than loss as a measure of congestion still lack adoption. A strong limitation to quickly testing new CC algorithms is that TCP is implemented in the kernel. This implies that the operating system needs to be updated to enable new features. Furthermore, this makes it difficult to apply different congestion control algorithms for different applications. [DASN] [NET4]

4.1.3. TCP Head-of-Line Blocking

TCP segments are identified by a sequence number and if one packet is lost during transmission, the promise of in-order-delivery halts processing of all arrived packets with higher sequence numbers, until retransmission succeeded. This is called TCP head-of-line blocking (HOL). I.e. in-order-delivery causes unpredictable arrival times of packets called jitter. Some applications like video streaming do not really need in-order delivery but are negatively impacted by jitter. [hpbn1]

4.1.4. Header Format

TCP connections are identified in the header by a source and destination port, summing up to 4 bytes, followed by the sequence number and the acknowledgment number, each adding another 4 bytes. 4 bits describe the header length, 3 bits are reserved for alignment, 9 bits of control flags to manage data flow and together with a window size described in 2 bytes it adds another 4 bytes. Finally the core header contains a checksum and urgent pointers of two bytes each which makes a *minimum TCP overhead of 20 bytes* plus optional 40 bytes to transmit options.

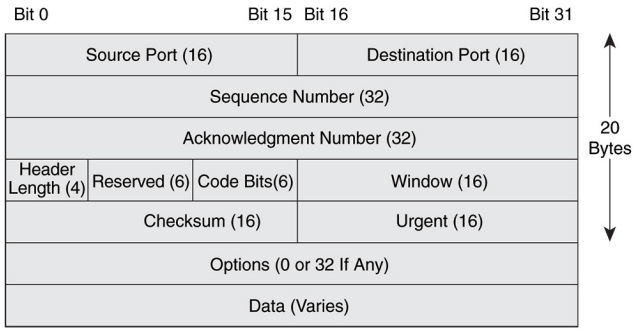

Figure 9: TCP header, Cisco[28]

[28] image source: https://ciscoskills.net/2011/02/25/understanding-tcp/tcpheader/

4.2. QUIC

QUIC[29] is a new transport protocol which was initially developed by Google in 2012. After revolutionising HTTP/1.1 with their proprietary protocol SPDY (later HTTP/2) they realised that further optimisation could only occur at a lower level in the network protocol stack. Other protocols like SCTP have tried to deploy “improved versions” of TCP but generally found little adoption due to ossification (middle-boxes). This is why Google decided to build QUIC, a new transport protocol on top of UDP which combines the functionalities of HTTP/2, TLS and TCP. This shall be possible to deploy because most of the protocols header information as well as the payload are encrypted, i.e., middle-boxes cannot see what they deliver and so they generally deliver. After large scale experiments, Google reported significant improvements of page-loading latency and handed it to the IETF. In 2016 the IETF established a working group to standardise QUIC as a general-purpose transport protocol, rather than only a new transport option for HTTP/2. Google’s original QUIC version, now gQUIC, and the IETF’s version QUIC (no longer an acronym) have grown incompatible during standardisation. Like with HTTP/2 and Google’s SPDY, it is to be expected that once standardisation completes, gQUIC will become deprecated and IETF’s version will be the only QUIC in use. [FECQ]

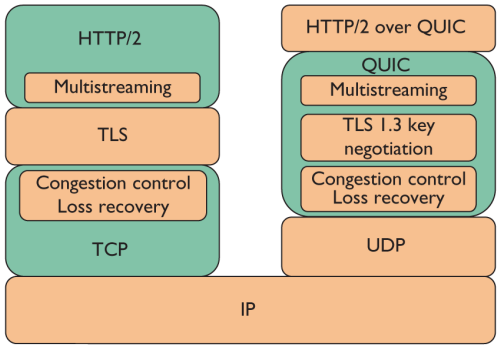

Figure 10: HTTP/2 over QUIC compares to HTTP/2+TLS over TCP, IETF[30]

Implemented in user-space[31], QUIC is a fast-changing and extensible transport protocol. Since the first deployment in 2012, more than 40 different gQUIC versions have powered communication between Google’s browser *Chrome* and their web services. While features like Forward Erasure Correction have been experimented with and taken out due to malperformance [FECQ], others have remained. According to Google, QUIC currently has the following features[32] [ChrQ]:

- Dramatically reduced connection establishment time
- Connection migration

---

[29] Originally at Google an acronym for Quick UDP Internet Connections but the standardised IETF QUIC claims to be a term for itself.

[30] image source: https://datatracker.ietf.org/meeting/98/materials/slides-98-httpbis-quic-00

[31] opposing to TCP which is implemented in the kernel

[32] Open extensions: Forward Erasure Correction [FECQ], Multi-path Delivery, Unreliable Transfer

- Improved congestion control
- Stream multiplexing without head-of-line blocking

4.2.1. Connection Establishment

Traditional TCP requires a full RTT to establish a connection, a seemingly inevitable delay for data-transfer to begin. According to [w3t], more than 50% of internet traffic is secured by Transport Layer Security TLS. This requires an extra RTT for the exchange of cryptographic keys, i.e., the initiation of a secure TCP connection can be very expensive at 2 RTTs. For instance, to connect from a mobile carrier with 3G internet in Europe, to a server in the United States frequently involves network delays larger than 100 ms. 2 RTT would cause 400 ms for connection establishment only, before the actual transmission starts.

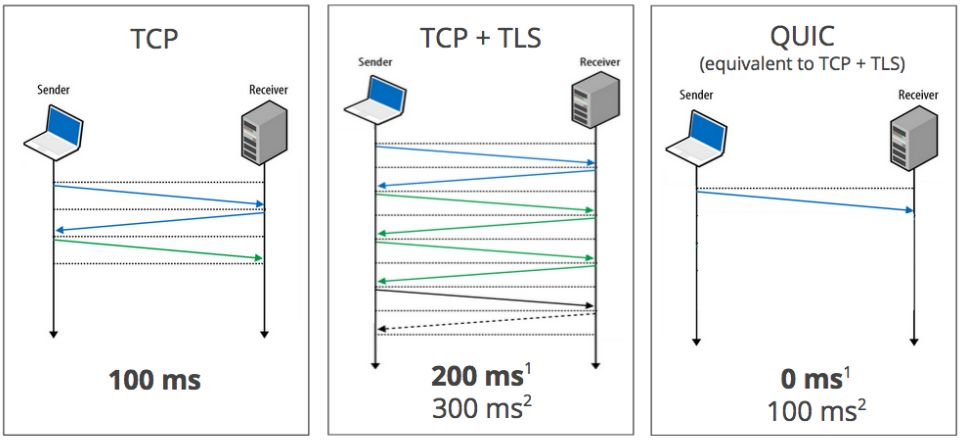

Figure 11: Connection Establishment TCP, TCP+TLS and QUIC (1.Repeat connection, 2.First connection), chromium[33]

QUIC reduces the number of RTTs by combining cryptographic (TLS 1.3) and transport handshakes. Additionally, a client caches a token and tries to reuse it for 0 RTT connection established in a later connection, i.e., immediate data transfer. The resulting time for connection establishment exists in three scenarios:

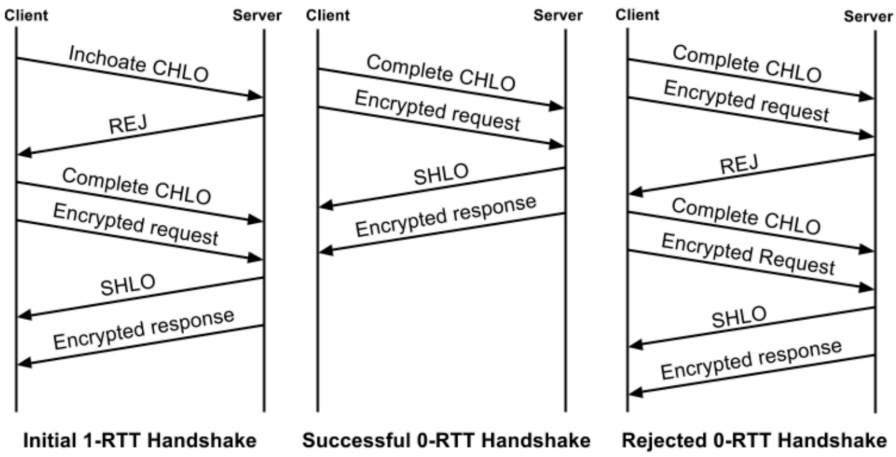

Figure 12: Connection Establishment QUIC [GQ]

[33] image source:
https://blog.chromium.org/2015/04/a-quic-update-on-googles-experimental.html

1. Initial Connection

If the client has never spoken to the server, an inchoate c*lient hello CHLO* message is sent to the server. The server responds with a *reject message REJ,* containing transport and cryptographic information to establish the current session and a source-address token to encrypt the next message (and possibly future connections). The client completes the handshake, sending a *Complete CHLO* along with an encrypted request. The server responds with a *server hello message SHLO* along with the encrypted response. All further message exchange is encrypted.

2. Repeat Connection

In case of a repeated connection, the client attempts to reuse previously cached cryptographic keys and immediately sends a Complete CHLO along with an encrypted request. If the server also remembers the connection, it responds with a SHLO message as well as the encrypted response. The handshake was successful(-ly “restored”) and happened in 0-RTT, i.e., data was transmitted along with the first message.

3. Failed Repeat Connection

If the client tries to initiate a repeat connection but the server does not remember or refuses the connection, 0-RTT fails. The server then regards the client's message as an inchoate CHLO and will need 1-RTT until data exchange begins (as if it was an initial connection).

[QWG] [CIQ] [GQ]

4.2.2. Connection Migration

To identify a connection, QUIC uses a Connection ID of 8 bytes. In contrast, TCP uses a fast-lived tuple of source and destination IP address and port number, which can change due to client migration or other IP address changes. Therefore QUIC Connection IDs can be reused. As a consequence, QUIC clients can maintain a connection despite the change of IP address which improves the internet performance of migrating clients[34] [SRI].

4.2.3. Multiplexing Without Head-of-Line Blocking

The IETF describes QUIC streams as an abstraction of a unidirectional or bidirectional, ordered byte-stream. Streams are implicitly created by sending data, stream management imposes low overhead and the number of streams is unlimited. Technically, data is packaged into frames and distinguished by a unique stream ID. One or more frames compose a QUIC packet which is then sent over the network. Since no ordering of bytes between different streams is necessary, loss of a frame only affects the stream in which loss occurred. Therefore QUIC avoids head-of-line blocking on a transport level [QC22].

[34] Migration could for example be due to changing networks. This is particularly interesting for mobile with the transition from Wi-Fi to Cellular networks.

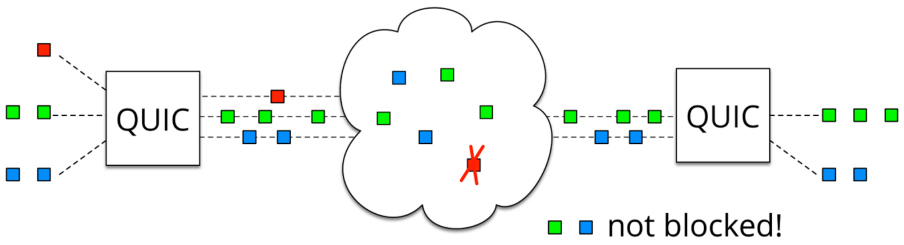

Figure 13: QUIC: stream multiplexing without head-of-line blocking [GQ]

4.2.4. Congestion Control

As for now, QUIC versions mostly use TCP-similar congestion control but with a new sequence numbering scheme. Every QUIC packet is assigned a new sequence number. In contrast with TCP, this includes retransmissions which improves the estimation of network delay. This is possibly an applicable strategy to improve congestion control, as new CC algorithms approximate congestion by observing RTT and bandwidth, rather than loss. Chair of Internet Congestion Control at the IETF, Jana Iyengar, explains “that the advantage in terms of CC is that QUIC is implemented in user-space” [FSTL]. Allowing fast-paced testing of network configurations in real environment makes further optimisation attainable. [CDVL]

QUIC’s mechanisms of flow control are split into connection level flow control and stream level flow control. Stream level flow control divides available network resources between the streams, with optional prioritisation. Connection level flow control manages the total byte-flow of all streams and ascertains not to overwhelm the receiving side. [QC22]

4.2.5. Header Format

QUIC wire-format is in development, thus experiences change both at gQUIC and at the IETF. Therefore, the description of all current QUIC wireframes is regarded out of scope and will instead be practically evaluated for the protocol versions in use. A major difference to TCP is that QUIC encrypts most of the header information as depicted in Figure 15. While encumbering internet service providers less transparency of network traffic, it also increases protocol security significantly. [Marx]

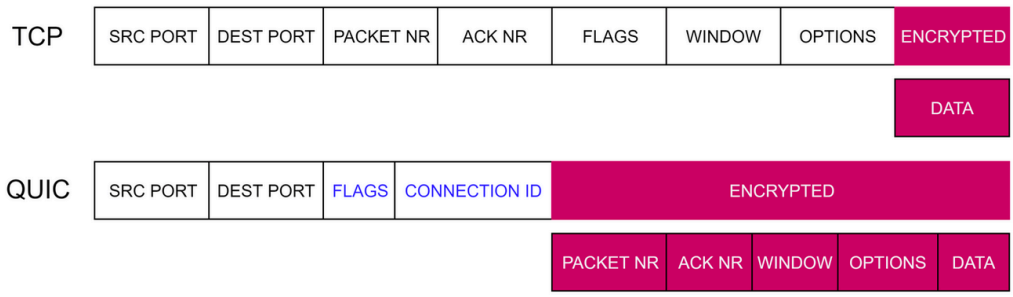

Figure 14: QUIC’s header is more secure than that of TCP [Marx]

5. HTTP

Since the beginning of the internet, the IETF and the World Wide Web Consortium (W3C) have been the main drivers of HTTP standardisation but in recent years Google strongly engaged in the development of internet protocols, primarily motivated to improve the latency of their own web services. Figure 15 highlights excepts of HTTP protocol related developments and Google's involvement.

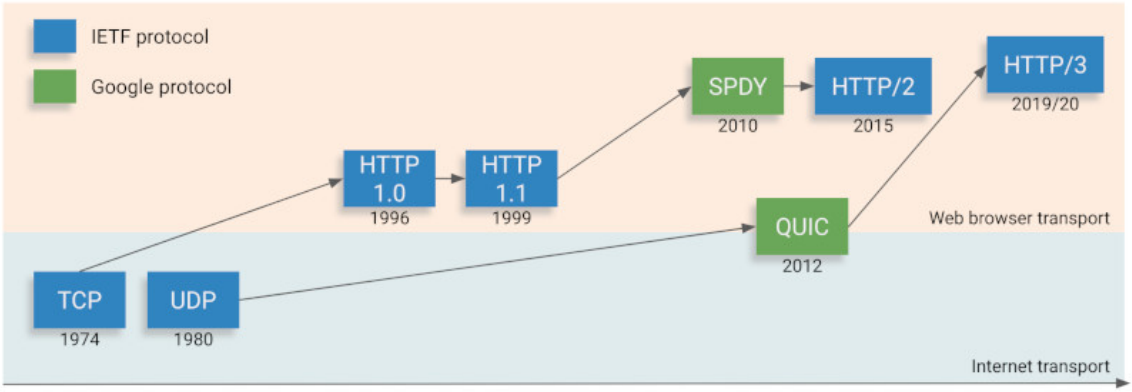

Figure 15: In blue the standardised HTTP versions of the IETF and in green google's proprietary protocols as predecessors [35]

The first standard HTTP/1.0 is commonly not regarded as a full protocol standard as it was only an informal RFC. While most servers still support it, the protocol finds rare use today. Current HTTP versions in use are HTTP/1, HTTP/2. In late 2018 the IETF decided to define the mapping of HTTP semantics on top of QUIC in a new version 3 of the protocol which is therefore called HTTP/3. Since it is still in active development, this work considers Google's HTTP/2-over-gQUIC vicariously, as it is already deployed and provides comparable features. The following section will summarise the protocol properties of the different HTTP versions: [MDN]

5.1. HTTP/1.1

As previously mentioned, HTTP messages consist of requests or responses, each containing (simplified) meta-data for transport, the header, and the data to transmit, the body. The body of these messages would be binary, but in HTTP/1.1 the header needs to be in white-space-delimited text fields which can lead to various processing difficulties.

Other than that HTTP/1.1 only requires a reliable transfer protocol and does not force the use of TCP. Since mappings other than HTTP over TCP are very rare, common HTTP implementations inherit all transport properties of TCP. This involves the 3-way-handshake, but also the lack of stream multiplexing. Each TCP connection can only transfer a single request at a time which makes requesting many HTTP resources at once[36] perform poorly. A common "hack" is to open multiple TCP connections in parallel which in turn caused new problems involving congestion control.

---

[35] image source: https://bloggeek.me/who-needs-quic-in-webrtc/amp/

[36] Sending multiple HTTP requests without waiting for responses is so called "pipelining".

5.2. HTTP/2

The goal of HTTP/2 was to optimise performance issues of HTTP/1.1 without requiring any alterations to the API, the “HTTP semantics”. Problems of the predecessor were mainly solved by introducing a binary framing and multiplexing layer, to stop web developers from opening multiple connections for concurrency. Practically this adds complexity between the socket interface and the higher HTTP API as seen in Figure 16: Instead of only delivering the body in binary while the header is in textual format like HTTP/1.1 does, HTTP/2 encodes the whole message into binary frames. The textual HTTP header gets encoded into a binary HEADERS frame and the body is parsed into a DATA frame. A TCP segment then consisted of one or more frames of type HEADERS, DATA or one of the many stream management frames of HTTP/2. While HTTP multiplexing improved the perceived latency it did not change TCP innate head-of-line blocking, i.e., the loss of a packet causes all “multiplexed” streams to wait and not only the transaction affected by it.

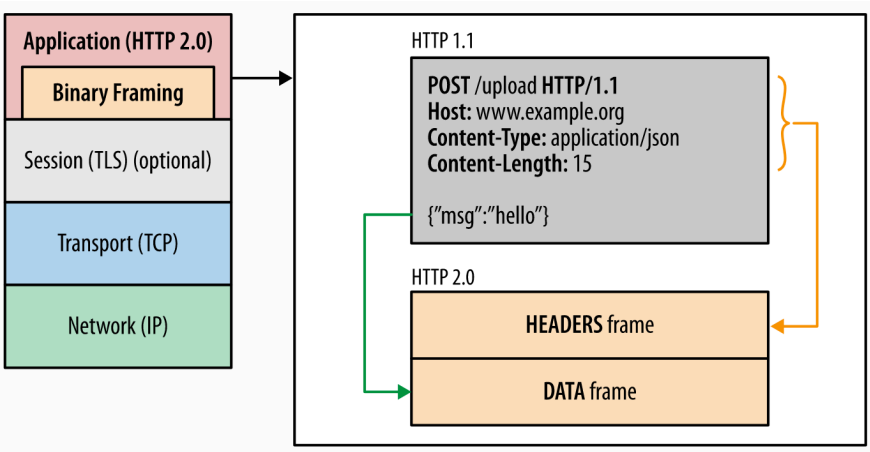

Figure 16: The binary framing layer of HTTP/2. The textual HTTP header gets encoded into a HEADERS frame and the body is parsed in a DATA frame [hpbn1]

Another problem of HTTP/1.1 was the inefficiency of the headers. As they were in textual format, they were large and repetitive and wasted valuable network resources. With separation of header and data in the framing layer, header compression became possible and was implement in HPACK. This technology establishes a shared compression context between client and server and makes the transmission of headers smaller, i.e., more efficient.

A last new feature introduced by HTTP/2 is HTTP push, which allows the server to push additional resources along with the requested resource. Breaking away from the pure request-response model allows the server to, for example, send all necessary resources(linked in a website) along with the actually requested HTML file.

5.3. HTTP/2-over-gQUIC (Predecessor to HTTP/3)

While HTTP/2 improved the poor design of HTTP/1.1, it could not change the problems related with TCP. Google therefore continued to innovate and created the QUIC transport protocol as an alternative transport for HTTP semantics. The protocol, firstly deployed in 2012,

powers 3.1% of all websites [w3t] and is currently under standardisation at the IETF. In late 2018 the IETF announced that the mapping of HTTP-over-QUIC will be called HTTP/3.

HTTP/2-over-QUIC implements a framing layer and other HTTP/2-features like server push and header compression[37]. By using QUIC as transport it is encrypted by default and also profits from the advantages which QUIC has over TCP. This includes reduced latency during connection establishment, pluggable congestion control, extensibility and stream multiplexing without head-of-line blocking. [h3]

5.4. Summary

The following table summarises the previous chapters concerning the properties of the different HTTP mappings. The differences concern connection establishment and management, multiplexing within a connection, protocol efficiency and extra features like HTTP push. To what extent HTTP Adaptive Streaming is impacted by the use of different HTTP versions (and their mappings) shall be practically evaluated in a testbed in the last section.

HTTP Version	1.1	2.0	HTTP-over-QUIC / 3.0
First Connection	1 RTT	1 RTT	1 RTT (secure!)
Repeat Connection	1 RTT	1 RTT	0 RTT
Secure TLS 1.3	optional +1 RTT	optional +1 RTT	included +0 RTT
Persistent Connection	yes	yes	yes
Connection Migration	no	no	yes
Transport Protocol	mostly TCP	mostly TCP	QUIC
Multiplexing	no	TCP HOL	yes
Streaming	chunked-encoding	binary framing	QUIC
Alt-Svc	yes	yes	yes
Header Format	text	binary	binary
Header compression	no	HPACK	QPACK
Server push	no	yes	yes

Table 1: Features of the standardised HTTP/1.1 and 2.0 and the new HTTP/2-over-QUIC which will soon become HTTP/3

[37] QUIC uses QPACK header compression which does not introduce head-of-line blocking

6. HTTP Adaptive Streaming: Workflow

Delivery chains of live streams can be manifold and require optimisation at every point of the chain. In its simplest but complete form this involves media capture, forwarding, processing, encoding, publishing and last mile delivery [THb]. Live streaming could for example begin in a TV production studio or from remote. HAS only comes into play after the media has been ingested at the encoder. The encoder ingests audio and video media, and outputs adaptive streaming formats ready for distribution. The following section, will analyse the workflow between the encoder and the media player, explaining principles, protocols and their interaction with the underlying network. This part of the streaming chain is nearly the same for Live and Video on Demand (VoD) situations and will therefore be explained as one. Afterwards a distinction between Live and VoD is made in Section 6.4.

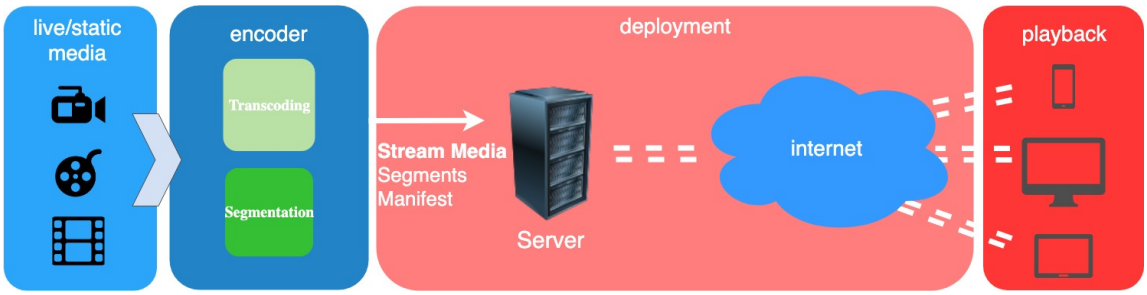

Figure 17: HTTP Adaptive Streaming workflow

6.1. Encoder

Figure 17 illustrates the steps which take place in HTTP stream encoders. At first, video and audio data is transcoded[38] and reduced in file size. Afterwards, media is segmented and put into file containers and a descriptive file—often called manifest—is created which describes the format and access of streaming media. Now streaming media is ready for delivery via a standard web-server. The following section will explain relevant parameters and concepts of transcoding, segmentation and file containers. The manifest will be discussed in Section 7 along with the HAS-protocol DASH.

6.1.1. Transcoding

To understand why transcoding is important, even though it is a time and resource consuming task in the delivery chain, a comparison of available network speed and needed bitrate is of help: 24 Mbit/s was the average download speed in Germany in late 2018 according to [BMVI]. Most of the size in multi-media is caused by video data[39]. Video itself is nothing but a series of sequentially displayed images. A raw, RGB image with Full HD resolution and 8

---

[38] transcode = digital to digital; encode = analog to digital

During this work encode, transcode or encoder, transcoder will be used synonymously always implying digital to digital conversion

[39] Audio data is relatively small compared to video and can be compressed better.

bit depth per colour channel requires at least 49 Mbit[40] of storage[41]. Using compression algorithm YUV422p, which converts three colour channels into one luminance and two downsampled chrominance components, a single frame still requires at least 33 Mbit. Therefore, a YUV422p encoded video with a frame-rate of 24 fps[42] would result in a bit rate above 790 Mbit/s. This is 30 times higher than the previously mentioned download speed average and proves the necessity of compression.

Figure 18: What happens in the DASH encoder - Timmerer at Bitmovin[43]

A codec (COder-DECoder) describes the set of rules to convert from one format to another (encoding) and back (decoding). This process of compression can be lossy or lossless. Lossless media compression preserves all the information, but compression ratios benchmark at 2:1. Therefore it is inevitable to use lossy compression for video streaming. [Corb]

H.264[44] as an example of lossy compression

In video-compression, it is common to distinguish between intra-frame compression (within a picture) and inter-frame compression (between consecutive pictures). Intra-frame coding uses redundancy[45] and irrelevance reduction[46] within a frame. Inter-frame coding uses motion prediction between temporarily consecutive blocks of pixels, i.e., it describes the repositioning of a part of an image instead of a whole new image. This motion prediction can be mono-directional, i.e., using only the following pictures for prediction but could also take the previous pictures into account which is called bi-directional. Motion prediction inevitably produces an ordered dependency between pictures which is commonly referred to as

---

[40] Usually file size is in Byte but for the sake of comparison it is represented in Bit.

[41] Full HD Image = 1080×1920×3×8 bit (height*width*RGB*depth)

[42] frames per seconds

[43] image source:
https://www.slideshare.net/christian.timmerer/mpegdash-dataset-mmsys-2012

[44] H.264 is a block-oriented motion-compensation-based video compression standard.
https://en.wikipedia.org/wiki/H.264/MPEG-4_AVC#Features

[45] https://en.wikipedia.org/wiki/Huffman_coding#Basic_technique

[46] https://en.wikipedia.org/wiki/JPEG

Group of Pictures (GOP). The longer a GOP, the greater the compression ratio but also the dependency between images, hence it requires more time to encode and decode. Inter-frame images can therefore only be decoded after the referenced images have become available.

Apart from the keyframe spacing (the length of the GOP), there are many other codec parameters. *Resolution* is straightforward, the larger it is, the larger the video size and generally the better the quality. *Bitrate control* manages the compression level (the greater the compression, the worse the image quality) and is divided into constant, average and variable bitrate control. *Constant bitrate control* (CBR) maintains a constant amount of transferred data per unit of time. *Average bitrate control* (ABR) takes into account the video's complexity over time and provides greater detail to more complex parts while reducing the quality of less complex parts. *Variable bitrate* (VBR) combines a target bitrate but allows some variance. This provides significantly better visual quality than CBR but poses an extra challenge for the media player as the actual media-bitrate deviates from the reported media-bitrate [BUF].

Codecs differ in speed, compression ratio, computational efficiency, quality, platform support and cost. Since the choice of codec does not immediately affect transport, we shall assume H.264 as codec throughout this thesis[47].

6.1.2. Segmentation and GOP

In H.264 a video segment consists of different types of frames. I-frames use intra-frame compression only and are the largest of all frame types. P-frames reference previous I-frames only, whereas B-frames reference both previous and subsequent I- and P-frames. The set of pictures between two I-frames is commonly referred to as a Group of Pictures (GOP). Encoder can place an extra I-frame inside a GOP to improve quality when a change of scene occurs.

As explained in 2.3 adaptive streaming clients automatically switch between segments, containing the same content but at different qualities, to cope with bad internet connections. This requires that segments can be decoded independently from one another, i.e., they must begin with an Instantaneous Decoder Refresh (IDR) frame[48]. IDR frames are I-frames which specify that no subsequent frame can reference a frame before it. GOPs which are not referencing frames from another group are called closed GOPs and always start with an IDR frame.

---

[47] because of its market dominance at 82% in 2018. Other relevant codecs would be HEVC, VP9 or WebM according to the Global Media Report of encoding.org.

[48] which resembles a possible stream access point in DASH

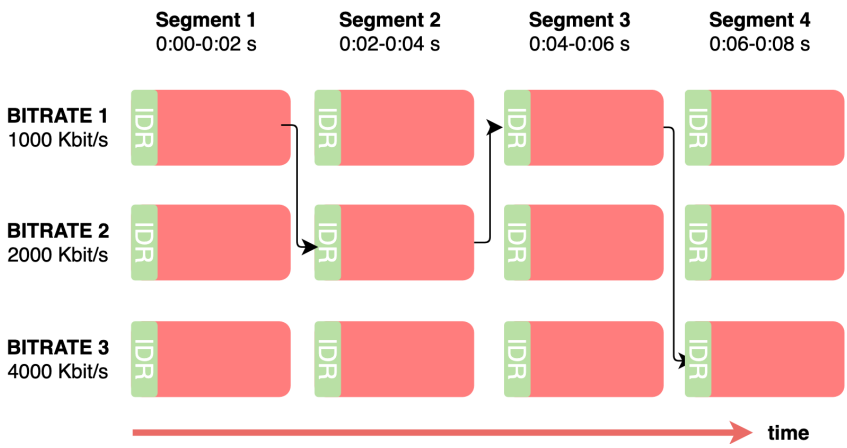

Figure 19: Media segments need to start with IDR Frames to allow switching between representations of different bitrates.

The Keyframe-Interval[49] can be fixed or variable. In both scenarios the length of the video segment[50] needs to be the same or a multiple of the closed GOP-length because otherwise the player could not decode it independently. The longer the Keyframe-Interval, the more efficient is the compression. It also defines a lower bound for the encoding-delay because a set of pictures needs to be buffered before encoding/decoding can take place. Furthermore, because video players try to minimise playback stalls, they tend to gather 2-3 segments before starting playback. Long GOPs imply longer segments, which further increases the startup delay. This also negatively impacts the adaptability to network changes and seeking, because with less IDR-frames, less stream access points are available. This encoding parameter has a great effect on distribution. Segment duration is commonly between 2-10s which has proven to perform best in most network scenarios. Bitmovin conducted a study in which they discovered DASH to perform best, with a persistent[51] connection using a segment length of around 2 seconds [BIT1].

6.1.3. Container

While codecs deal with how data is encoded and decoded, containers organise how data is stored. It is a meta format which bundles multiple files into one container. This is relevant for packaging, file-transport and presentation. Usually containers, also referred to as wrappers or file format and can distinguished by their file extensions. The main differences are the support of different audio/video codecs, meta-data but also license and support for the container in different streaming protocols and environments. Some formats also incorporate Digital Rights Management (DRM) to prevent unauthorised distribution.

---

[49] the time interval at which IDR frames appear

[50] in most streaming protocols the same across all segments

[51] When after sending a first HTTP request, a network communication channel remains open for further message exchange, rather than closing after the first.

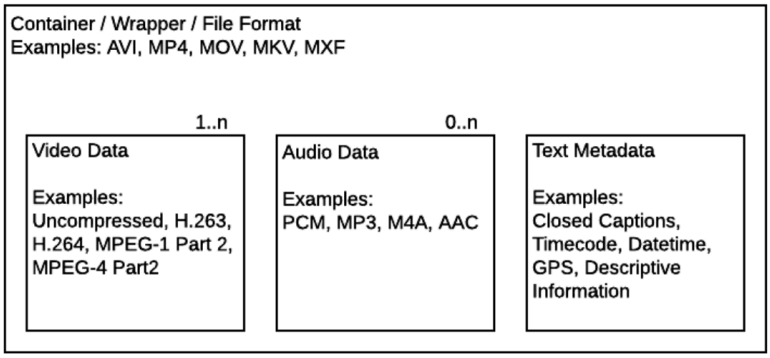

Figure 20: An illustration of how digital containers store video, audio and meta data [SWGDE]

Currently competing formats are the older MPEG-2-TS, mainly supported by Apple's HLS protocol and the ISO-Base Media File Format ISOBMFF[52] which is mainly used by DASH. Streaming providers who want to serve both protocols have double the cost in encoding and storage. After Apple announced support for fragmented MP4 in 2016, Common Media Application Format CMAF[53], a mezzanine container format, has been developed to serve both protocols. As seen in the Figure 21, CMAF specification excludes the manifest generation. Streaming providers store media in CMAF and generate DASH manifest and a HLS playlist which reduces costs immensely[54] [WZA1]. Apart from the advantages of storage, CMAF provides a feature which will improve HTTP streaming live latency drastically. By allowing addressing of media below the segment level, the distribution becomes independent from the segment length. Combined with smart deployment live delays of large scale HTTP streaming can be reduced below 5 seconds and is increasingly becoming a topic of interest throughout the streaming industry. [MUX]

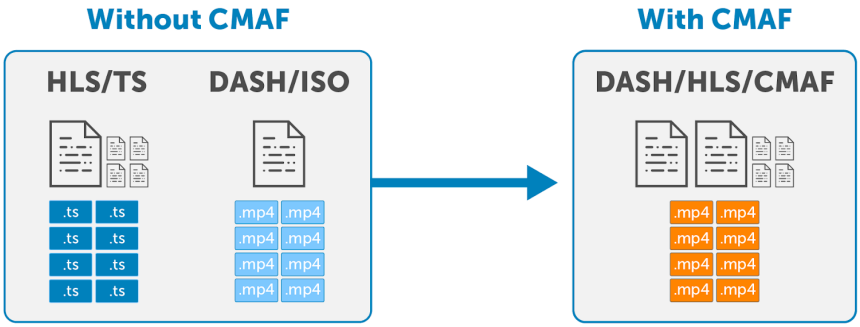

Figure 21: Advantages of storing media in CMAF[55]

[52] MP4 is based on ISOBMFF

[53] CMAF https://mpeg.chiariglione.org/standards/mpeg-a/common-media-application-format

[54] Unfortunately digital-rights-management is still in the process of harmonisation as HLS (CBS) and DASH (CENC) currently still use different DRM technologies. This makes parallel hosting of HLS and DASH still difficult.

[55] image source: https://developer.apple.com/documentation/http_live_streaming/about_the_common_media_application_format_with_http_live_streaming

6.2. Deployment

For HTTP streaming technologies have already been explained in Section 2, Section 6.2.1 will focus on the advantages of deploying streaming via HTTP and Section 6.2.3 will contrast HAS with non-HTTP streaming solutions.

6.2.1. Advantages of Streaming via HTTP

There are multiple HAS protocols which partly differ in their support of container formats, DRMs and codecs but generally share the following advantages. [DASN]

- **Standard HTTP-servers** can be used as media is made available as downloadable files.
- **Statelessness** meaning that the server does not have to manage each client, leading to lower resource demand and allowing scalability
- **Reuse Cache and CDN infrastructure** meaning that the client can retrieve media from geographically closer network edge caches. Furthermore during live streaming HTTP proxy caching significantly reduces the amount of requests on the server.
- **Firewall and NAT friendliness** - NAT is a solution to handle the problems regarding the relatively small number of possible devices due to the address shortage of the 32bit IPv4 address space. The division of network addresses into local and remote often breaks how streaming protocols like RTSP initiate their connections unless the NAT service supports their initiation protocol. HTTP rarely encounters problems with NAT since it works on a request-response-model. Apart from NAT, firewalls could block ports other than the ones relevant for their services. The original RTMP protocol runs on TCP port 1995 and often encounters firewall problems.

6.2.2. Comparison to Non-HTTP Live Streaming

HAS protocols share many advantages by running on HTTP, but also common disadvantages concerning live latency. Live (end-to-end) latency describes the time from the recording of the media until playback at the end user. Traditional broadcasting[56] is able to provide reliable live streams at scale with end-to-end latencies below 5 seconds. This has become a benchmark for internet streaming and requires complex setups for systemic reasons.

HTTP was not designed for streaming media but rather to reliably deliver HTTP resources such as HTML pages. This means the recognition and recovery of loss is intrinsic to HTTP and comes at the expanse of increased latency. Additionally HAS is stateless like HTTP, i.e., the server does not manage streaming sessions. While this scales well it introduces further latency caused by the pull-based nature of HAS. Due to the statelessness, the server cannot actively inform the client on the availability of segments. Instead a new segment needs to be completely processed before it gets published and signalled in the manifest. This explains HAS' higher latencies compared to other technologies.

---

[56] Broadcasting refers to a communication type in which one point sends information to all other points, e.g., satellite broadcast in television.

RTMP[57] is the oldest scalable low latency streaming technology which is still in use. It requires a browser plugin and only some versions of it allow encrypted data transfer. Furthermore it lacks DRM support and does not provide adaptive streaming. WebRTC[58] is a UDP-based[59] peer-to-peer technology which enables sub-second latency but does not provide DRM and is expensive at scale. WebRTC realises adaptive streaming with a selective forwarding unit which decides which quality clients receive. This performs really well but makes WebRTC not only expensive at scale but also limited in scale [Mil].

	HTTP (DASH, HLS, CMAF)	RTMP (Flash)	WebRTC
Function	last mile	ingest, last mile	last mile
Browser Support	all*	yes but decreasing	partly
Plugin required	no	yes	no
Latency	standard: 30s-45s tuned: 2-10s	< 2 s	< 1 s
Scalability	> 100k	>100k	<10k
ABR	yes	no	partially
Secure	yes	only RTMPE/RTMPS	yes
DRM	yes	no	no

Table 2: Last-mile streaming protocols [exM]

6.3. Playback

6.3.1. HTML5

In the early days of HTTP streaming, browsers relied on plugins to support the streaming formats like the RTMP-based *Adobe Flash*. These had to be updated frequently and sometimes even posed a security threat to the user [FLS]. Nowadays most browsers do still not support popular streaming protocols like DASH or HLS natively[60] but instead HTML5 APIs[61] which enables Javascript[62] applications to handle the playback of video streams, without the

[57] The Real-Time Messaging Protocol needs a plugin like Adobe Flash to work, is semi proprietary (Adobe) and while it is almost deprecated in browsers, it is still relevant as ingest for live streaming as it can achieve low latencies. https://www.adobe.com/devnet/rtmp.html

[58] https://webrtc.org/

[59] User Datagram Protocol is a connectionless unreliable transport protocol.

[60] it can be used without extra component or programs

[61] programming interfaces provided by browsers in compliance with the official standards defined by the W3C

[62] A scripting language for the browser and for non-browser environments https://developer.mozilla.org/en-US/docs/Web/JavaScript

need for plugins. To do this, javascript enabler applications rely on the following three HTML5 APIs [EYE]:

- **Media Source Extensions MSE** provide access to the media pipeline within a browser as well as an interface to the internally supported codecs for decoding. It does not define which codecs are supported (this is browser-dependent) but rather enables client-side “prefetching and buffering”[63].
- **Encrypted Media Extensions EME** provide access to Digital Rights Management DRM and the Content Decryption Module CDM and enable the playback of encrypted media. It does not define which DRM is supported as this is, equal to codec-support, browser-dependent.
- **VTTCue** is an interface for exposing WebVTT cues to the DOM. This is needed to process subtitles.

	DASH	HLS	MSE
Chrome	no	no	yes
Firefox	no	no	yes
Internet Explorer	no	no	mostly
Edge	yes	yes	yes
Opera	no	no	yes
Safari	no	yes	yes

Table 3: Streaming protocol and MSE support in modern browsers according to caniuse.com

The browser takes care of handling encoding media like this: When encoded, possibly encrypted streaming media arrives at the client, it is pushed to the MSE source buffer. If encryption is detected, the EME initiates a connection to a DRM server and relevant information is retrieved which allows the CDM to decrypt media. The browser then forwards unencrypted media to the HTML5 video element for playback.

While the browser takes care of handling encoded media, it is unaware of streaming protocol formats. This is where javascript enabler applications continue, handling how media bytes are requested and buffered. A javascript streaming application like DASH.js[64] or HLS.js[65] typically consists of the following parts (see Figure 22) [EYE2]:

- The **Manifest Parser** downloads and parses the manifest and informs the Player Engine about the available quality levels, the required bandwidth and the address (URL) of media segments.
- The **Player Engine** is the core of the player. It orchestrates the segment downloader, manages timing, pushes the data to the MSE source buffer or the subtitle parser and decides when to change the quality level.

---

[63] This means the process of retrieving segments and reassembling them into media.

[64] DASH.js https://dashif.org/dash.js/

[65] HLS.js https://github.com/video-dev/hls.js/

- **Segment Downloader** executes the downloads as instructed by the Player Engine, can implement retry logic and reports the time it took to download a segment to the ABR Manager.
- **ABR Manager** implements heuristics (explained in Section 6.3.3) to give recommendation to the Player Engine on when to change the quality level. This is the most critical part of a player.
- The **License Request Wrapper** and **Remuxer** are optional components for the encryption and extended support of streaming formats.

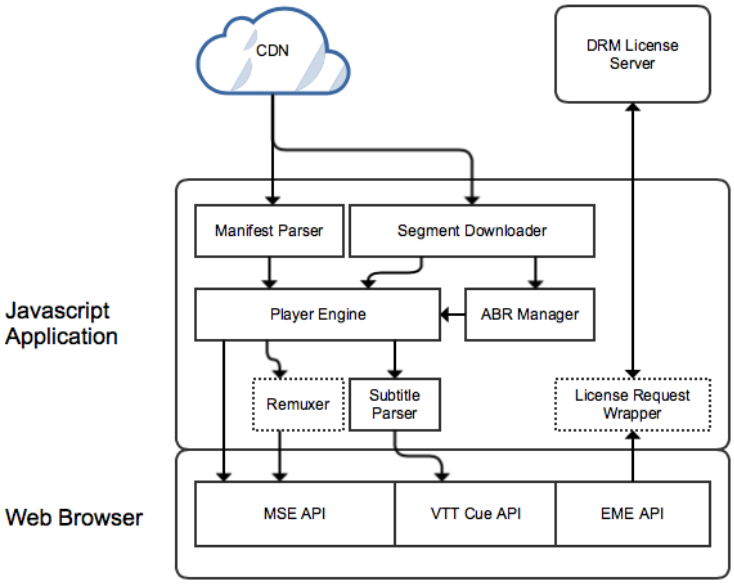

Figure 22: Javascript + MSE for streaming protocol support [EYE2]

6.3.2. Playout Buffer

The playout buffer[66] is where segments are stored before playout. Efficient buffer management is crucial for providing good, stall-free streaming experiences. When a stream is started, the player tries to fill the buffer as quickly as possible. To build a stable buffer which can handle network quality fluctuations, the players commonly only start playback after the buffer contains 2-3 segments.

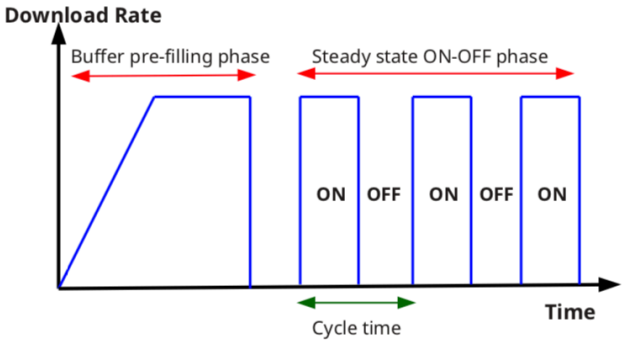

Figure 23: Bursty behaviour of HAS players [BUF]

[66] An abstraction of the MSE source buffer.

The buffer size is limited and once a threshold is reached the player enters a steady state. As the player tries to maintain a maximum threshold, i.e., it always downloads the next segment once the playout of another is completed, network traffic will show an ON-OFF pattern. At ON, segments are downloaded and OFF is the extra time $\Delta T=T-t$ seconds when the player manages to download a T-second segment in t seconds.

Video on Demand (VoD) and live streaming differ because in live streaming the maximum buffer is limited by the availability of already created segments. While it appears to be simple, this component is difficult to optimise and bad management can cause degraded streaming experiences full of stalls [BUF].

6.3.3. Rate Adaption

Bitrate adaptation can be regarded as an optimisation problem[67]: media bitrate is to be maximised at a minimal number of stalls and the least number of quality changes. There are three main types[68] of adaptive bitrate algorithms which leverage information provided by the media player.

Bitrate adaptation algorithms [DIF]:

- **Throughput-based** uses the moving average of the segment downloaded rate to determine the maximum media bitrate which can be supported.
- **Buffer-based** monitors the buffer occupancy and maps a maximum bitrate accordingly. It can be prone to oscillations, which is usually stabilised by taking into account throughput-history.
- **Mixed/Dynamic** dynamically switches between the previous two algorithms and performs best.

6.4. Live vs. VoD

Streaming without regard to the technological realisation can be divided into Video-on-Demand VoD and Live. If HAS streaming is on demand all segments already exist at the time of requesting the stream. Therefore the manifest file is complete at the time of the first request. Contrarily in Live HAS: At the time of the initial request, the manifest is yet to be completed. Content is still being produced and after a while the manifest, which the client currently has, will not contain any new segments. HAS streaming mostly follows a pull-based approach, meaning that the server only sends files explicitly requested by the client and cannot inform the client about new segments.

---

[67] Even though this seems straight-forward, it still is a highly researched topic in the streaming industry. One aspect to consider is the interaction of the ABR Controller, which represents an application layer congestion control, and the transport layer congestion control (e.g. TCP cubic). This will be briefly discussed in a later section.

[68] Secondary rules involve rate adaptions due to CPU limitations or prevent quality oscillations

Common solutions to the problem of signalling segments' availability in live HAS are [UnS]:

- **Dynamic Manifest**, meaning that the manifest is updated, re-requested at a fixed interval.
- **Static Manifest**, uses a naming template and estimates availability time.

7. Dynamic Adaptive Streaming over HTTP - DASH

7.1.1. Background and Scope

In the early days of HAS multiple proprietary solutions coexisted which were incompatible despite strong similarities . This caused extra expenses for streaming providers as it required separate content preparation and storage for wide support of clients. In 2010 the MPEG Working group started the HAS standardisation process until the publication of the international standard MPEG DASH in 2014 [DASN].

The HAS standard DASH only defines how segments are formatted, indexed and addressed. Not specified is the delivery of the DASH manifest, the codec to be used as well as the client behaviour for fetching, adaptation heuristics and playback. MPEG has patented DASH and charges a fee to implement a DASH-client. The DASH industry forum[69] open-sourced DASH.js, a javascript-based media player which will be used in later experiments. This streaming player enables DASH support using HTML5 APIs as described in Section 6.3.1.

7.1.2. Media Presentation Description

At the core of MPEG-DASH is the manifest which is called Media Presentation Description (MPD). It is a hierarchical[70] data model in XML-format[71] which contains temporal and structural information on the streaming content. Other HTTP streaming protocols have similar indexing files like for example the *.m3u8* playlist file in HLS.

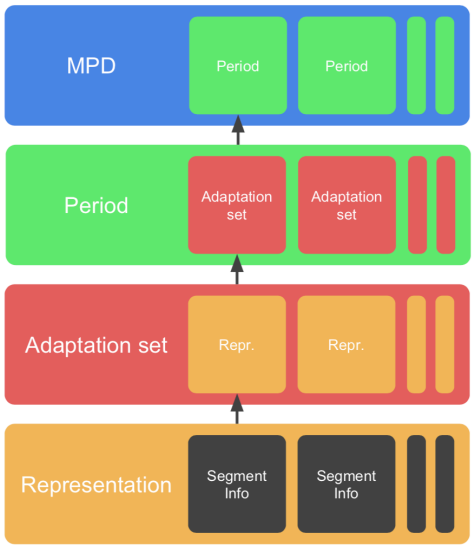

Figure 24: Structure of the MPD [DASN]

[69] DASH-IF is a group of leading streaming companies that drives research and development to achieve greater protocol adoption.

[70] i.e. data is organised in a tree-like structure. One MPD contains N Periods, which each contain N adaptation sets…

[71] eXtensible Markup Language is a descriptive language for storing and transporting data.

DASH's manifest MPD is a hierarchical model which contains multiple nested layers with the following functionalities:

MPD

The outer layer of the MPD defines metadata about the stream like the total runtime or the DASH profile.

Period

The period defines a period of media content. This could represent a scene or be used for ad insertion. The number of periods and the length are flexible and a period consists of one or more adaptation sets.

Adaptation Set

Arrangements of different media components like video, audio and subtitles can be achieved with adaptation sets. It is common to group representations containing the same codecs, language, resolution or audio channel formats. A period could also contain a subset instead of adaptations sets to restrict the combination of adaptation sets - For example to only allow HD content with 5.1 audio channel format [BIT2] [DASN].

Representation

Several representations of one adaption enable the client to change between low and high quality representations. Apart from the media, a representation contains rich meta information such as the used codec, media type, media bitrate or segment size, to help the player choose. Most importantly, the provided URL which points to the representation's location, i.e., the URL from which to retrieve the segment. It is also possible to use multiple URLs to increase reliability and possibly the speed of access if the resource is located closer to the client.

7.1.3. Other HAS Protocols

The previously explained standard by MPEG is yet to be fully adopted, i.e., other HAS protocols are in use. According to the global media report 2019 [GIMF] Apple's HLS still dominates the market at 67%(-7% since 2018) but DASH continues to find adoption at 29%, 7% more than in 2018. While DASH continues dominating OTT devices and desktop browsers, Microsoft's Smooth Streaming (MSS) continues to move towards deprecation, which is confirmed by Microsoft's engagement in the new CMAF standard.

According to a protocol comparison by Bitmovin, MPEG-DASH is the most efficient and flexible protocol involving features such as codec agnosticism (independence), flexibility with container formats, UHD resolution support, etc. Most core features are also provided by Apple's HLS such as support of both fragmented MP4 (fmp4) and Transport Stream (TS), the deployment on standard HTTP-servers or DRM flexibility. Microsoft Smooth Streaming on the other hand lacks many features and cannot compete with the other two protocols, hence why it's starting to become deprecated.

Feature	Apple HLS	MS Smooth Streaming	MPEG–DASH
Deployment on Ordinary HTTP Servers			
Official International Standard (e.g., ISO/IEC MPEG)			
Flexible Content Protection with Common Encryption (DRM)			
Agnostic to Video Codecs			
Agnostic to Audio Codecs			
ISO Base Media File Format Segments			
MPEG-2 TS Segments			
HEVC Ready (UHD/4K)			
Efficent Ad Insertion			

Table 4: HAS-protocol feature comparison: green = yes, red = no by Bitmovin[72]

[72] HAS Protocol comparison by Bitmovin, image source: https://bitmovin.com/mpeg-dash-vs-apple-hls-vs-microsoft-smooth-streaming-vs-adobe-hds/

8. Quality Monitoring of HTTP Streaming

Conventionally deployed HAS runs over best-effort networks, which strictly do not guarantee any Quality of Service (QoS). QoS in computer networking refers to traffic prioritisation and resource reservation [CisQ]. Quantitative measures for this include packet loss, bit rate, throughput, transmission delay, jitter, etc. In this sense best-effort implies variable bit rate, loss or latency depending on the momentary traffic load [WIK3].

In video streaming the term QoS was newly interpreted as a metric for users' satisfaction with the provided streaming service. Even though HTTP streaming metrics lack standardisation, Quality of Experience QoE metrics have increasingly found adoption. According to streamingmedia.com [StrM] these metrics estimate the perceived improvement or degradation of the service, i.e., the viewer's satisfaction with the streaming experience.

Therefore, to measure HTTP streaming protocol quality, distinction is made between: QoS, referring to network related issues, such as fairly sharing network resources, and QoE representing the viewer's satisfaction.

8.1. Key Performance Indicators of QoS

While QoE metrics are visible to all end-users, metrics of QoS could, for example, only be a starting point to reduce the cost of deploying a service. There is a plethora of QoS metrics which always involve the statistical analysis of network parameters. Here are the most important ones [Tri]:

- *Overhead analysis*: Even though DASH runs exclusively on top of HTTP, overhead added during transmission can vary between versions and configuration.
- *Congestion Control & Fairness*: When providing a service for many users, it is important to share available resources fairly. In DASH, congestion control exists mainly on the transport level but strongly interacts with ABR techniques implemented in the application layer.
- *Computational Complexity*: As computational complexity increases, the CPU consumption is higher leading to higher energy costs.
- *Memory Usage:* Higher RAM usage is immediately linked to higher cost in deployment.

8.2. Key Performance Indicators of QoE

Correctly representing user experience is traditionally achieved in subjective quality evaluation tests resulting in a Mean Opinion Score MOS. In video streaming using unreliable transfer, algorithmically predicting video quality can be a very complex task - Packet loss would need to be mapped to video degradation. Fortunately, DASH runs on top of a reliable transport protocol which means that video quality immediately relates to media throughput. Nevertheless, there are a number of parameters, provided by most DASH players, which can be related to the viewers' experiences. Following various research papers in the field of QoE measurements (e.g., [Tri], [StrM]), these metrics are commonly used:

- **Rebuffering ratio** describes the ratio of the total stall duration and the total time streamed.
- **Start-up time** describes the time from the first request to a streaming webpage until playback starts.
- **Average bitrate** describes the average of the selected bitrates during a streaming session. In HAS higher bitrates immediately link to a better video quality level.
- **Number of quality switches**, describes the stability of a stream, i.e., less changes in quality improve QoE.
- **End-to-end latency** describes the time from media capture until final playback at the client during live streaming.

9. Testbed

9.1. Related Work

In 2016 Bitmovin[73] and the Information Institute of Technology (ITEC) conducted a study in which they analysed performance differences of DASH stream delivery via HTTP/1.1 and HTTP/2 over a standard TLS/TCP connection, HTTP/1.1 over plain TCP (without TLS), as well as a SPDY (HTTP/2 predecessor) mapping over an early version of Googles transport protocol QUIC (gQUIC 15-19). To provide a baseline for comparison, they firstly measured the protocol overhead. Furthermore they tested network link utilisation and analysed how well the protocols adapted to varying network bandwidths. They concluded: While HTTP/2 is slighltly less efficient due to the framing mechanism, it provides improvements in high delay networks compared to the earlier HTTP-version HTTP/1.1. SPDY-over-QUIC performs worse than HTTP/1.1 with a slightly higher overhead and lower media throughputs. [BitQ]

In summer 2018 Arisu,Yildiz and Begen published “Is QUIC ready for prime time streaming“, analysing how QUIC’s features could improve HTTP-based streaming. They tested both in a controlled and an uncontrolled environment with a python-based player as a client and a slightly adapted version of a QUIC Server provided by Google vs. an Apache server providing HTTP/1.1 and HTTP/2.0. Test cases evaluate different multiplexing techniques, congestion control algorithms and how fair protocols behaved. Also QoE-oriented metrics like the average playback bitrate or re-buffer rate were analysed in varying network conditions and during switched networks. They found that QUIC provides significantly better QoE with lower wait times and shorter re-buffer rates but is not *fair* to competing TCP streams. [AYB]

Both studies attempted to benchmark QUIC as an alternative transport protocol for DASH related streaming services. [BitQ] focused on a baseline performance assessment, combining and comparing hitherto transport and application layer protocols in a controlled streaming environment. [AYB] conducted a study focusing on QUIC’s practical QoE implications compared to TCP in combination with multiple application layer mappings. Tests were carried out in a closed-of as well as an open network environment (internet). This testbed will complement elements of the baseline performance assessment of [BitQ] with QoE measurements similar to [AYB]. Even though the differences between live and on-demand streaming appear to be negligible, limitations such as generally lower buffer levels enfeeble live streaming stability and are more affected by the network protocols. Therefore, this testbed will focus on HAS in live streaming contexts.

9.2. Test Setup

With IETF QUIC and HTTP/3 still being in active development, it is yet to be supported by any browser. Even though the original gQUIC and the standardised IETF QUIC have become incompatible it is possible to use gQUIC as reference in terms of performance. This is very convenient since Google’s *Chrome* Browser has QUIC enabled by default. In fact, most communication between Google services and *Chrome* is powered by gQUIC, leading

[73] https://bitmovin.com/

gQUIC to contribute 3.2% of global internet traffic in July 2019. HTTP/2 has a share of 30.4% for the delivery all websites and the rest is almost exclusively powered by HTTP/1.1. Furthermore HTTPS, encrypted traffic, is now used for the majority of all websites at 53.8%.

For comparability this testbed seeks to provide all HTTP versions on a single technology-stack, as well as only using open source software. The following software is made use of:

Server-sided:

Caddy Server[74] is open-source, uses TLS(1.3) by default, supports HTTP/1.1 and HTTP/2 over TCP and has experimental QUIC support, complying with Google QUIC and is thereby supported by Google Chrome. This testbed will only use caddy server but could be extended to use other servers instead.

Ubuntu[75] is an operating system running on the Linux Kernel and powers the server in the testbed. The version which is used in the testbed is 18.04.

Docker[76] is a containerisation software, which is used to isolate as much of the testbed as possible to keep it portable and thereby simplify the reproduction of results.

FFMPEG[77] is command line program which has most encoding, decoding and transcoding capabilities. FFMPEG is used to transcode the video and to generate a DASH livestream which conforms with the DASH standard.

Tc[78] is a command line tool to conFigure traffic control in the linux kernel. With this the network emulation will be achieved.

Client-sided:

Chrome[79] is a popular browser by Google which will be used on our client. It supports all versions of our testbed.

DASH.js[80] is the reference player of the DASH Industry Forum. It is a production quality MPEG-DASH media player fully built in Javascript.

Puppeteer.js[81] is a Node library which provides a high-level API to control *Chrome* or Chromium. With *puppeteer* streaming sessions are conducted and meta data is collected.

TShark[82] is a command line version of the tool Wireshark to capture network traffic. In the testbed it is used to measure the network overhead during a streaming session.

---

[74] Caddy: https://caddyserver.com/

[75] Ubuntu: http://releases.ubuntu.com/18.04/

[76] Docker: https://www.docker.com/

[77] FFMPEG: https://ffmpeg.org/

[78] Tc: http://man7.org/linux/man-pages/man8/tc.8.html

[79] Chrome: https://www.google.com/chrome/

[80] DASH.js: https://dashif.org/dash.js/

[81] Puppeteer.js: https://github.com/GoogleChrome/puppeteer

[82] TShark: https://www.wireshark.org/

9.2.1. Preparation of the Test-data

The testbed will use a short film “Big Buck Bunny” by *blender foundation* as streaming content. The original video is in 1080p resolution at 30 fps and encoded in H.264. The next steps will explain how FFMPEG is used to create an Adaptive DASH stream.

Live streaming is a resource consuming task, which, if operated on the same machine on which the server is run, could distort test results. Since the transcoding does not affect the transport, media data is prepared for DASH, so that only packaging and manifest generation, both computationally simple processes, need to be done live in the transmuxer:

The idea is to provide each live streaming session of 100 seconds with 14 possible quality levels, i.e., video-bitrates ranging from 100kBit/s to 4500kBit/s. This task is split into transcoding and live-transmuxing.

Transcoding

Like Timmerer pointed out in [BitQ], the actually perceived quality is not of immediate interest (HTTP-innate reliability causes VQLs to be equal to the measured media throughput) and to prevent the player logic from limiting bitrate adaption to the reported screen size, the video resolution is fixed at 640x360 pixels. The audio does not have a great impact on the final streaming bitrate and will be removed to simplify the setup. In FFMPEG, the video is transcoded to codec H.264 with a keyframe interval of 2 seconds (same as the later segment duration). The video will then be compressed at 14 constant bitrates[83] as shown in the table below. MP4 containers allow multiple video quality levels to be stored into a single file which is done accordingly.

Quality	Resolution	Max Rate (bps)
0	640x360	100k
1	640x360	200k
2	640x360	350k
3	640x360	500k
4	640x360	700k
5	640x360	900k
6	640x360	1110k
7	640x360	1300k
8	640x360	1600k
9	640x360	1900k
10	640x360	2300k
11	640x360	2800k
12	640x360	3400k
13	640x360	4500k

Table 5: Pre-transcoded video and the resulting bitrates

Transmuxing

Now in the live transmuxer, FFMPEG can simply select the wanted bitrate, segment along the keyframes and generate the MPD. This is very performant, will not overly occupy the CPU and shall be done live.The result is an endless stream looping 100s of pre-encoded but live transmuxed DASH media. The manifest uses segment template, i.e., it only needs

[83] Momentarily observed bitrate can deviate up to 10% from the maximum bitrate.

to be retrieved once. The segment names are then generated from the template until the stream ends.

9.2.2. Server Setup

On the server machine, the creation and the hosting of the stream is split into 4 processes. To make it easier to reproduce the testbed these processes are virtualised into Docker containers and perform the following tasks, illustrated in Figure 25.

The first container runs FFMPEG, creating a DASH live stream from the prepared media. It loops over a pre-transcoded video and writes resulting streaming media (representations and MPD) into a shared volume.

Apart from the streaming media, a HTML site is added to the volume, containing the DASH-IF Reference player DASH.js, and a script to expose player statistics to the DOM. The content of the shared volume is hosted in three containers each speaking different HTTP versions:

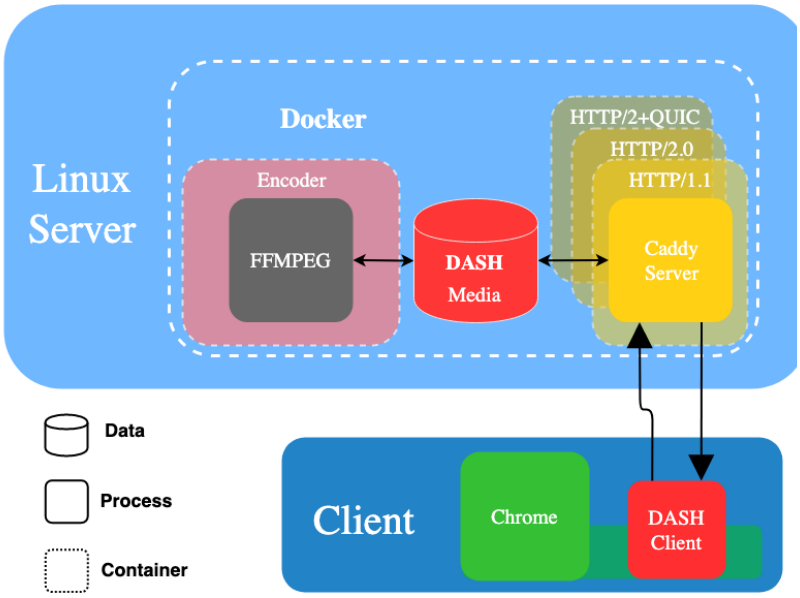

Figure 25: Testbed streaming server setup: one encoder container, three server containers

One container hosts HTTP/1.1 without TLS on TCP port 8080, as well as a secure version using TLS 1.2 on TCP port 445. Another container hosts HTTP/2 over TLS+TCP on port 444. The last container hosts the HTTP/2 mapping over QUIC/43 on UDP port 443.

Server	Caddy:Latest
HTTP/1.1 NO TLS Port	TCP Port: 8080
HTTP/1.1 Port	TCP Port: 445
HTTP/2	TCP Port: 444
HTTP/2+QUIC/43	TCP/UDP Port: 443

Table 6: Server Configurations

9.2.3. Network Setup

The testbed depends on a realistic simulation of network traffic. An initial approach to emulating network conditions was client-sided and simply re-used *chrome*'s built-in network throttling tool. First results were inconclusive: e.g. connection times were always zero, random performance differences independent from the protocol, etc. Further research showed why: In an article by Google [thrG], the following approaches to emulating network traffic are mentioned:

1. *Request level throttling* works on the application layer and is a simple and inaccurate emulation because it only effects the final download speed of a page. This is the standard in Google *Chrome* developer tools' throttling.
2. *Proxy-level throttling* is a proxy server through which all traffic is filtered. While the quality of emulation is often good most proxy-level throttling tools do not support UDP nor QUIC and can only throttle TCP traffic.
3. *Packet-level throttling* realises network emulation on the IP layer and provides the most realistic network emulation but introduces more variance than other technologies. This is because the emulation is additional to the real network quality.

Request level throttling can be used to evaluate application layer performance.It should not be used to evaluate the performance network protocols. Since proxy-level throttling commonly does not support UDP or QUIC this testbed will emulate traffic on packet-level. The following section will explain how this is achieved.

Packet-level throttling with linux traffic control Tc

Both TCP and QUIC/UDP run over IP. This means data is broken done into small units, packets, and sent from source to destination node. The linux command line tool traffic control Tc provides an interface to the linux kernel to control the ingress and egress traffic on a network interface.

Traffic Control distinguishes between rules of shaping, scheduling, policing and dropping. Shaping controls the rate of transmission and only occurs on egress. Scheduling adds priority mechanisms to the transmission of outgoing packets. Policing is the equivalent of scheduling for ingress traffic but much simpler in its capabilities. Dropping limits how many packets a network interface can carry at a time.

All packets which enter or leave a network interface need to have rules on how this happens. This can be a simple queuing logic, meaning the **f**irst **i**n, is the **f**irst **out** (FIFO). Meanwhile packets are stored in a buffer of limited size. Consequently, if the buffer of the queue is full, packets get dropped. This is the most primitive way of dealing with packet transmission but queueing can become more complex.

Queuing disciplines (qdiscs), which are applied to a network interface, describe how data is sent and always implement some sort of queueing with a FIFO logic. They can be classful or classless. While classless *qdiscs* treat all traffic equally, classful queuing disciplines allow to create classes which can then be filtered independently.

Token bucket filter is a classless *qdisc* and the main algorithm for rate control. The idea is to have a token bucket which is filled at a steady rate but only until it is full. Every packet needs to use a token to be processed, hence the transmission rate is eventually limited by the token rate. TC also implements a classful version of this algorithm, the *Hierarchical Token Bucket (HTB) filter*, a *classful qdisc*. A second *classless qdisc* is *netem* which provides algorithms for the simulation of delay, latency, reordering, etc. [Sys]

The testbed requires the adjustment of bandwidth, delay and loss of both ingress and egress. Egress traffic can be controlled with the *qdiscs netem* and HTB, i.e., determining the sending rate of packets, delaying them or discarding at random to simulate loss.

Ingress *qdiscs* naturally have fewer capabilities than egress *qdiscs*. E.g. it would not make sense to try limit the incoming bandwidth or rate because packets have already arrived. To solve this an intermediate functional block IFB is used. This is a pseudo-device on the same computer to which incoming packets all incoming packets are forwarded. From the IFB packets will be resent to the original interface, hereby allowing to treat incoming traffic like outgoing traffic. The following Figure illustrate the complete setup on the setup which allows the emulation of varying network conditions. [LnX]

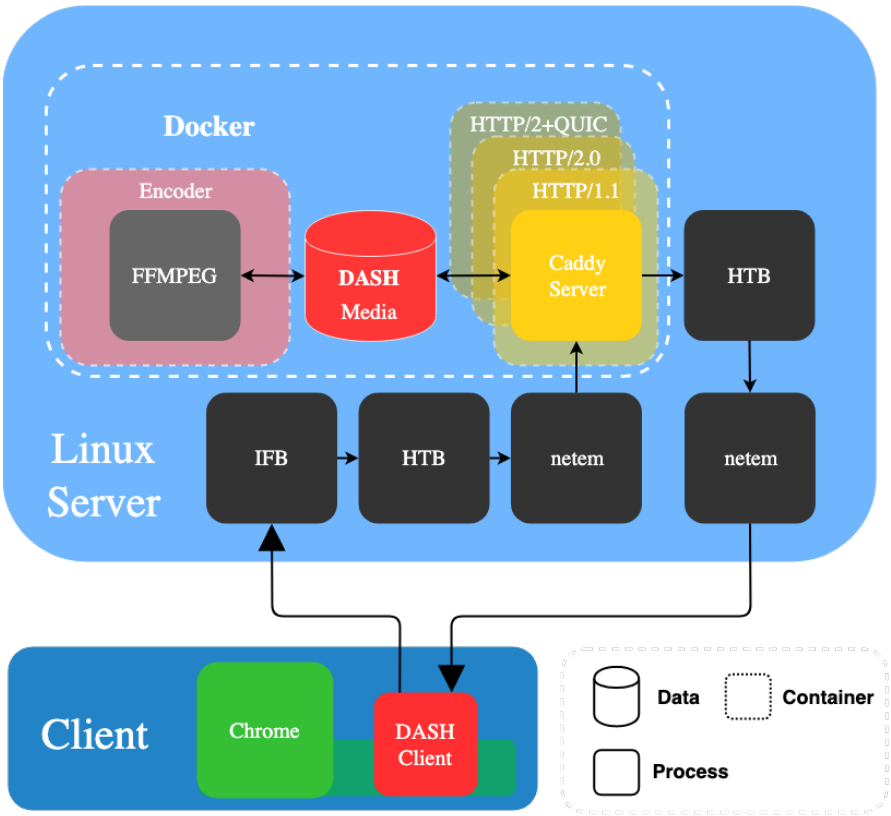

Figure 26: Complete testbed setup to emulate network conditions

Figure 26 clearly shows that all of the network emulation takes place on the server machine. In order to automate streaming sessions on the client side, the client needs to be able to run traffic control commands on the server side. For this remote command line over a *secure shell (SSH)* connection will be used. The server machine is configured to allow root login without password. The client's public key is copied to the server's authorised hosts and can now connect to the server, using private key as identification. In the following script the cli-

ent establishes a connection to the server and passes a list of variables to the host machine. Once connected all old *qdiscs* are removed. Then egress traffic is rate-limited with HTB and delay and loss are added using *netem*. For limiting ingress traffic an IFB is created to which all traffic is redirected. The IFB is now sending packets (egress!)to the local network interface and HTB and *netem* are used in the same way as for real egress traffic. Loss will not be applied again, because it has already been applied to egress traffic. [IFB]

The following script shows the code which will be used in the testbed:

```
1. #ssh into server
2.  ssh -i ~/.ssh/foo […] 'bash -s' <<'ENDSSH'
3. #remove old qdiscs
4.   tc qdisc del dev $DEV root &>/dev/null
5.   tc qdisc del dev $DEV ingress &>/dev/null
6.   tc qdisc del dev ifb0 root &>/dev/null
7. #egress
8.   tc qdisc add dev $DEV root handle 1: htb
9.   tc class add dev $DEV parent 1:0 classid 1:1 \
10.    htb rate ${DOWNLOAD}kbit ceil ${DOWNLOAD}kbit
11.  tc qdisc add dev $DEV parent 1:1 handle 10:0 \
12.    netem delay ${DELAY}ms loss $LOSS limit 100000
13.  tc filter add dev $DEV protocol ip parent 1:0 prio 1 u32 \
14.    match ip dst 0.0.0.0/0 flowid 1:1
15.#ingress
16.  modprobe ifb
17.  ip link set dev ifb0 up
18.  tc qdisc add dev $DEV ingress
19.  tc filter add dev $DEV parent ffff: protocol ip u32 \
20.        match u32 0 0 action mirred egress redirect dev ifb0
21.  tc qdisc add dev ifb0 root handle 2: htb
22.  tc class add dev ifb0 parent 2:0 classid 2:10 \
23.    htb rate ${UPLOAD}kbit ceil ${UPLOAD}kbit
24.  tc qdisc add dev ifb0 parent 2:10 handle 20:0 \
25.    netem delay ${DELAY}ms limit 100000
26.  tc filter add dev ifb0 protocol ip parent 2:0 prio 1 u32 \
27.    match ip src 0.0.0.0/0 flowid 2:10
28.ENDSSH
```

Code 1: Client-sided script to manage network conditions

To test, the script is run with a download bandwidth of 16000 kBit/s and after with a delay of 200 ms and loss of 2%. Command line tools *ping* and *iperf* confirm the emulated network:

```
--- caddy-testbed.com ping statistics ---
88 packets transmitted, 86 packets received, 2.3% packet loss
round-trip min/avg/max/stddev = 400.381/400.871/401.176/0.151 ms
```

Figure 27: A *ping* test confirms the emulated network

Delay and loss are checked with *ping* and are measured separately from the bandwidth as they would affect the observed bandwidth drastically. Note that 2% loss was only applied to the download bandwidth (from a client's perspective) but delay on both ingress and egress resulting in a RTT of around 400 ms.

```
ID] Interval           Transfer     Bandwidth
 4]   0.00-10.00  sec  18.4 MBytes  15.4 Mbits/sec
```

Figure 28: An iperf speed test also confirms the emulated network

The observed bandwidth was always a little lower than the emulated bandwidth. Presumably, this is due to the overhead produced by TCP delivery. The testbed contains a shell script to start the encoder and the server instances in *Docker containers*. With this setup the client can now run the tests in varying network conditions. The hardware on which the server is deployed is listed in Table 7.

HP 250 G5 Notebook	
OS	Ubuntu 18.04
CPU	4x2,7 GHz Intel Core i7
RAM	16 GB
Ethernet	10 Gbit/s
Storage	250 GB SSD

Table 7: Server Hardware

9.2.4. Client Setup

In order to use QUIC with *Chrome*, TLS needs to be set up properly. While *Chrome* does not regard self-signed certificates as secure, chaining the certificate to a self-signed root certificate seems to work. Chaining of certificates includes the domain name, which is why the server ip-address needs to be linked with the domain used in the certificate. The current setup uses caddy-testbed.com as a domain. To do so the *hosts* file of the client is changed. Further details can be found in the *readme* of the testbed.

Now Chrome should be able to access the HTML site which contains the DASH.js player and exposes metrics of interest like in the following Figure:

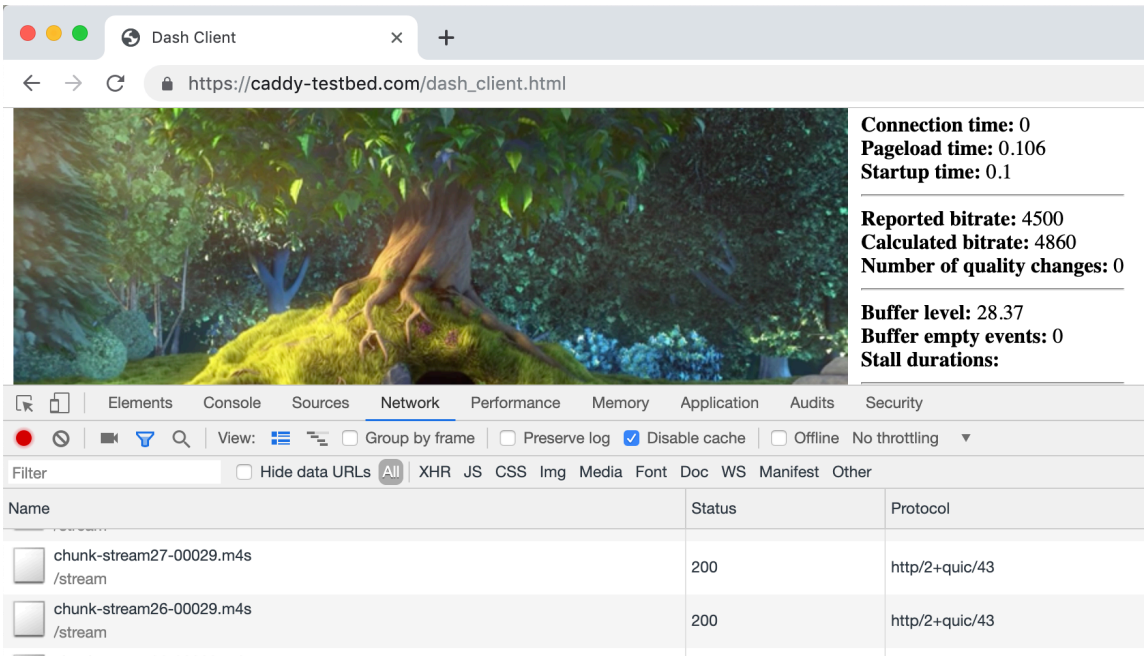

Table 29: The main HTML site contains DASH.js and a simple javascript logic for metrics

Upon loading *DASH_client.html*, DASH.js is retrieved and the player initialised. The player settings are set to autoplay and audio is muted. This is because automated playback of unmuted video in HTML5 without user input can cause autoplay to fail. Additionally, a simple *javascript* logic has been implemented to expose player and browser statistics to the document object model DOM.

The exposed metrics are:

- connection time, page load time, start up time
- reported bitrate, calculated bitrate, number of quality changes
- buffer level, number of buffer empty events, stall durations
- live latency

The initial concept was to *isolate* the client in *Docker*, same as the server, but this posed too many challenges which, as for now, could not be worked around. While *Chrome* provides the possibility to run headless i.e. running without GUI, *Wireshark* analysis has shown that using QUIC frequently fails silently. This could potentially be resolved by emulating a remote display to which Chrome could render but this has not been tested yet. Furthermore, TCP connections did not timeout as quickly, when running headless, leading to an incorrect measurement of startup times. It has therefore been decided to run *Chrome* “headful”, i.e., with *GUI* and thereby not to make use of the headless feature.

The current client setup is built in *OSX 10.14* and uses a combination of *bash* scripts and the *node* library *puppeteer* to automate streaming sessions in *Chrome* and retrieve the metrics, exposed in the DOM. The exact logic depends on the test-cases and will be explained in Section 9.4.

The tests were run on the following client hardware.

Apple MacBook Pro 15 Inch Mid 2014	
OS	macOS Mojave 10.14
CPU	4x2,2 GHz Intel Core i7
RAM	16 GB
Ethernet	10/100Mbps - Edimax EU-4208
Storage	250 GB SSD

Table 8: Client Hardware

9.3. Test Cases

To measure the performance impact of HTTP versions on live-HTTP streaming, a server hosts a live video and multiple streaming sessions are conducted with varying (simulated) network conditions and player configurations.

The first part follows the study conducted by Bitmovin [BitQ] and evaluates the total network overhead during a streaming session. Afterwards, industry standard QoE parameters are observed during streaming sessions in varying network conditions. The last part will compare the protocols’ adaptation performance to changing network bandwidths. Ideally this test-

bed will show to what extent QoE is affected by the choice of network protocols and shall be connected with theoretical part of this work. This could possibly help stream providers decide to upgrade to newer protocols.

9.3.1. Network Overhead of a Streaming Session

The network overhead during a streaming session will be calculated as shown in formula (1) and will be tested for different media bitrates:

$$network\ overhead = 1 - \frac{media\ bytes}{total\ bytes\ received}$$

Formula 1: Network overhead

There are many ways to retrieve information on the number of downloaded bytes but programming forums mentioned a bug in Chrome's performance API. This bug leads to presenting the uncompressed download size instead of the actually transmitted compressed size. The developer tools in *Chrome* show both the compressed and uncompressed and the complete download size is manually extracted for the 14 representations. Then, *TShark* is used to record all network traffic between the client and server, during repeated streaming sessions of fixed period. Explanation of the code is given in the comments:

```
#!/bin/sh
LIST_OF_SERVERS=[…] #location of different http versions
LIST_OF_BITRATES=[…] #list of bitrates to analyse overhead
DURATION=65 #stream length[s] + 5s to account for start up time
RUNS=5 #how many times to loop

for server in $LIST_OF_SERVERS
  for bitrate in $BITRATES
    for i in $RUNS
  #record network traffic on interface en7 for 65s & save in tmp
     tshark -a duration:$DURATION -i en7 -w tmp.pcap &
     #open chrome without cache
     Google Chrome \
      --args \
       —disk-cache-dir=null \
       —disk-cache-size=1 \
       —media-cache-size=1 \
  #& stream in each quality of LIST_OF_BITRATES
      "$server/single_bitrate_client.html?r=$bitrate" \
  #wait 65s and then close chrome
    & read -t $DURATION; pkill -a -i "Google Chrome”
  #read recorded traffic and get statistics
      tshark -r tmp.pcap -q -z conv,ip \
  #select only traffic with server
  | grep '192.168.0.4' \
  #extract total bytes received and save to file
  |awk -v b="$bitrate" '{print b >> “overhead”}'
```

Code 2: Recording metrics for overhead calculation

After, the measurement results are put into formula 1 and visualised and interpreted in Section 10.1.

9.3.2. QoE in Networks of Varying Bandwidth, Delay and Loss

The next test-case measures standard QoE metrics by monitoring DASH player metrics during repeated streaming sessions in varying network conditions. Firstly the metrics of QoE are recapitulated, followed by an explanation of the emulated network and lastly the process of automating streaming session and collecting data is depicted.

Metrics

The following metrics for representing QoE will be measured and calculated:

- *rebuffering ratio* is the most important metric regarding QoE. Studies have shown that rebuffering/stalling events significantly degrade user experience of streaming and is the most common reason for users to quit watching a stream. [QoE]

$$rebuffering\ ratio = \frac{total\ time\ playing}{total\ not\ playing}$$

Formula 2: Rebuffering ratio

- *start-up time* measures the time from the first GET request until playback start. Lower values imply better user experience.
- *average bitrate* is the average of the reported or measured bitrate. A higher average bitrate implies a better video quality level, leading to improved user experience.
- *number of quality* changes is the count of changes of the selected representation. Studies have shown that an increased number of changes worsens QoE. This metric is to be considered in combination with the average bitrate and describes the stability of the stream.

Network

The script of Section 9.2.3 will be used to emulate the following network conditions in four groups of five: degrading bandwidth, increasing delay and loss and all combined. All emulated scenarios are shown in Table 9.

Automation and collecting data

The automation is implemented in a combination of a *shell* script, to set the network conditions, and a *node* script, controlling the browser and retrieving information.

The *shell* script loops through a list of all predefined network conditions at each of which it runs the *node* script. Setting the network conditions will be done as described in Section 9.2.3.

In the *node-based* script, *puppeteer-core* is used to control *Chrome* and to retrieve data from the DOM. *Chrome* is run in headful mode with cache disabled. The script then regularly retrieves the relevant information from the site using puppeteer. Once the amount of samples is collected, data is written into a JSON file and the process repeats. When running

the script it accepts parameters which determine how often, how many times, at which interval samples are retrieved and also the adaptive bitrate algorithm the player is supposed to use.

Emulation	Bandwidth	Delay	Loss
Bandwidth 1	16000	0	0
Bandwidth 2	8000	0	0
Bandwidth 3	4000	0	0
Bandwidth 4	2000	0	0
Bandwidth 5	1000	0	0
Delay 1	0	50	0
Delay 2	0	150	0
Delay 3	0	250	0
Delay 4	0	350	0
Delay 5	0	450	0
Loss 1	0	0	0.5
Loss 2	0	0	1.5
Loss 3	0	0	2.5
Loss 4	0	0	3.5
Loss 5	0	0	4.5
Combined 1	16000	50	0.5
Combined 2	8000	150	1.5
Combined 3	4000	250	2.5
Combined 4	2000	350	3.5
Combined 5	1000	450	4.5

Table 9: QoE Test - 20 Emulations of varying bandwidth, delay and loss

9.3.3. Adaptation Performance

The last test will measure how well the combination of congestion control on the transport level and on application level (ABR-controller) work together to adapt to varying bandwidths.

Most transport level congestion control algorithms settle on a sending rate by transmitting increasing amounts of data until the detection of loss. Most HAS algorithms make their download choice by estimating the network throughput. To achieve good QoE in streaming the interaction of ABR algorithm and transport layer congestion control needs to work symbiotically.

This will be evaluated by streaming for 150 seconds and changing the available bandwidth every 20s. Meanwhile media player-data on buffer fill and selected quality level (reported/calculated bitrate) are recorded and later put into relation with the available bandwidth.

9.4. Evaluation

9.4.1. Network Overhead of a Streaming Session

In Figure 30 we observe that HTTP/1.1 without TLS/SSL adds the smallest overhead at around 4.8%, which increases to ca. 5.4% when encryption is used. Encrypted HTTP/2 sits just above at 5.6% and the HTTP/2 mapping on top of QUIC averages at around 6%. The overhead of all protocols is relatively constant throughout most media bitrates and only increases at media bitrates below 500 kBit/s. The observed median and maximum MTU for all TCP-based downloads was at 1514 Bytes and UDP-based QUIC had a single maximum MTU of 1395 Bytes but the median was at 1295 Bytes. Based upon the measured MTU, theoretical overhead approximations for both the TCP and UDP stack are depicted in grey.

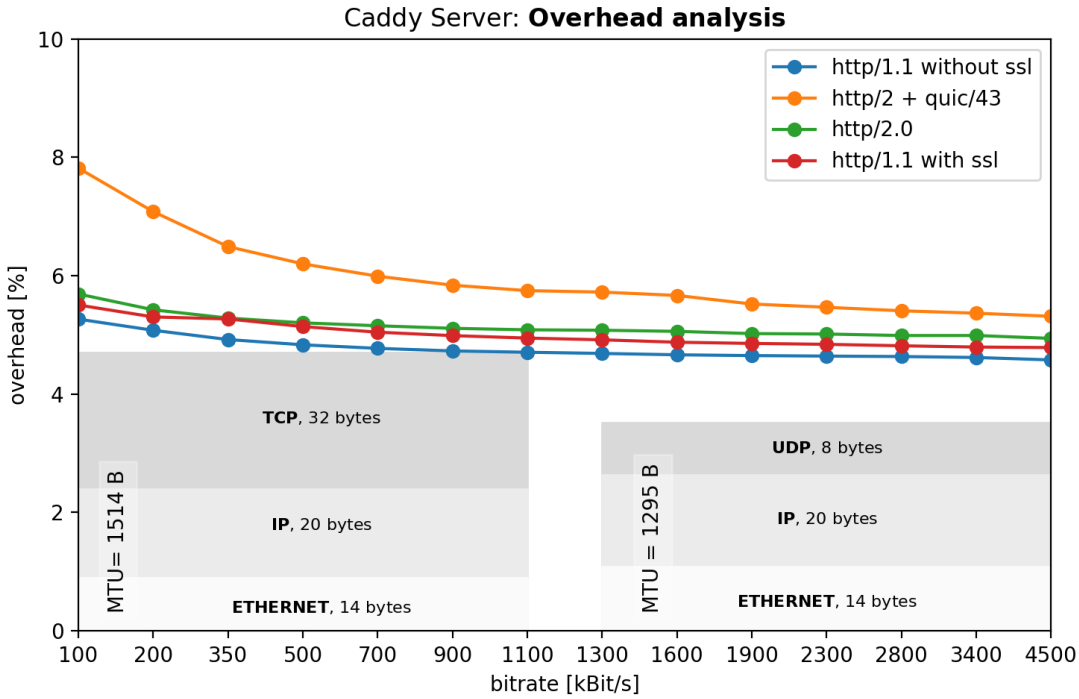

Figure 30: Measured network overhead (coloured), estimate of the underlying protocols based on the observed MTU (greyscale). The network is not lossy and has a RTT < 10 ms.

The overhead of HTTP/1.1 without TLS compared to secure HTTP/1.1 is a constant factor of around 0.4% percent. HTTP/2.0 adds extra 0.2% for multiplexing HTTP requests compared to HTTP/1.1. HTTP/2-over-QUIC is built on UDP which has a lower overhead than TCP. As it implements reliability and congestion control in the application layer, the corresponding application overhead is a little bit higher but still low at 8%. More critical to the final throughput could be that Caddy chooses a very low MTU for almost all traffic. As a result overhead is higher as the same data is chunked into more smaller packets.

The measurements confirm Bitmovin's study [BIT1], for HTTP/1.1 has the lowest overhead both with and without ssl. The overhead of HTTP/2-over-QUIC is a little bit higher on average than what they measured of the SPDY-over-QUIC but performs better at low media bitrates. The higher overhead on average is presumably because QUIC has since been further decoupled from HTTP/2, in order for it to become a general purpose protocol. Better performance at low bitrates could be because of a better implementation of the protocol.

9.4.2. QoE in Networks of Varying Bandwidth, Delay and Loss

The following will analyse the 4 QoE-metrics measured during 20 streaming sessions in degraded network conditions of 4 different mappings of HTTP, showed in Figures 30-33. The order of analysis is metric by metric, left to right, i.e., first decreasing network bandwidths, second increasing network delays[84], third increasing network loss and last combined-degraded (all previous limitations together).

Rebuffering ratio

All protocols performed well with only a single degrading parameter of bandwidth, delay or loss but performances diverged in combined-degraded networks.

Throughout all degraded conditions HTTP/2-over-QUIC manages to remain a rebuffering ratio of 1.0 (stall free) more than 80% of the time and the worst case is a rebuffering ratio below 1.025 (during 1Mbit/s,500ms RTT, 4.5% loss). With 100s of streaming this would be around 2.5s. Second best was HTTP/1.1 without SSL/TLS, stall-free 50% of the time, even in poor network conditions but the worst-cases were higher at 1.07 (~6.5s). Third was secure HTTP/1 with the same average as without TLS but worse outliers at 1.16. This would be around 14 seconds of not playing the 100 second stream but only accounted for ~20%. On average worst performed HTTP/2 with only ~40% stall free streaming sessions and a worst case of 1.1 for a rebuffering ratio.

Start-up time

Degrading bandwidths only caused minor average differences in start up time, where HTTP/2 was a little less stable with outliers slightly worse than other protocols.

In case of increasing delays HTTP/2-over-QUIC is almost 1s faster on average(at 500 RTT) with an average start up time of 4.3s and low variance through the 20 streaming sessions. Second best during RTT of 500 ms is HTTP/2 with an average start-up time of 5.2s but at higher variance between the runs. The mean of HTTP/1.1 with and without TLS is at around 6s during a RTT of 500 ms but the unencrypted version shows more variance and worst start up time > 10s.

All protocols managed network loss up to 3.5% and HTTP/2 performed minimally better than other protocols at 4.5% loss. The difference does not exceed 0.4s in start-up time.

The protocols' performances diverged most in combined-degraded conditions. During high bandwidth (16 Mbit/s), medium delay (100 RTT) and medium loss (0.5%), HTTP/2 and HTTP/1 without TLS performed a little better than the other protocols with differences up to ~1.3 seconds. As combined-degraded network conditions worsen, HTTP/2-over-QUIC remains stable at a start-up time of 5s. Other protocols show similar start-up times more the 3 seconds higher on average but HTTP/2 shows higher variance than both secure and simple HTTP/1.

Unexpectedly HTTP/2-over-QUIC does not seem to improve but worsen the connection time. HTTP/1.1+TLS and HTTP/2 are a little lower and simple HTTP/1 showed the lowest connection time.

---

[84] 1 RTT = 2 x network delay

Reported bitrates

Degraded bandwidths were dealt with equally well by all protocols.

With increasing delay in the network HTTP/2-over-QUIC outperforms other protocols and only starts reducing playback quality with RTTs >= 400 ms. Second best is HTTP/2 which already starts request lower bitrates at RTTs >= 300 ms and equally worse perform both encrypted and simple HTTP/1.1, already selecting lower bitrates at RTTs > 200 ms.

Loss alone up to 4.5% did not affect the selected bitrate.

In combined-degraded networks HTTP/2-over-QUIC overall maintained the highest selected bitrate up to a bandwidth of 4 Mbit/s at 300 ms RTT and 2.5% loss, even though HTTP/2 performed a little better than in lightly combined-degraded networks.

Number of quality changes

During degrading bandwidths all protocols maintain a similar stability in the choice of quality, only HTTP/2 showed an increase during very low bandwidths with an average of 8 changes.

With increasing delays HTTP/2-over-QUIC remained stable up to RTTs >= 500 ms. HTTP/2 shows an increase in changes at RTTs >= 300 ms and HTTP/1 both simple and encrypted at RTTs >= 200 ms.

Loss alone did not affect any protocols.

In combined degraded networks HTTP/2-over-QUIC provides most overall stability, with ~4 quality changes in the most degraded conditions while all other protocol versions share similar (light) instability with an average of ~6 quality changes per streaming session.

Summary

HTTP/2-over-QUIC significantly improved QoE of streaming in combined-degraded networks (low bandwidth, high latency, high loss) with overall lower rebuffering ratios, higher average bitrates with less quality changes and faster starting streams. HTTP/2 showed a slight performance boost to HTTP/1.1 in high-delay networks with higher average bitrates and faster starting streams, but performed worse in lossy and combined-degraded networks. Using HTTP/1.1 with TLS compared to without TLS, only negatively affected QoE in networks of high bandwidth, low delay and low loss with slight decrease of the average bitrate selected bitrate.

Interpretation

Performance improvements of HTTP/2-over-QUIC in poor network conditions are linked to QUIC's multiplexing without head-of-line blocking. Even though Caddy server did not succeed to immediately open a QUIC connection[85](connection times where higher than all other protocols) the effect was insignificant in low delay networks and in combined-degraded networks, the slower connection establishment was fully compensated by achieving better overall throughput.

---

[85] *Chrome* was launched with the flags to immediately use QUIC but without cache. It was therefore expected to constantly perform connection establishment in 1 RTT, as it should with initial connections. Later *wireshark* analysis showed that the flags did not take effect and first, a TCP connection was established which was then upgraded to a QUIC connection. This should be considered in future work.

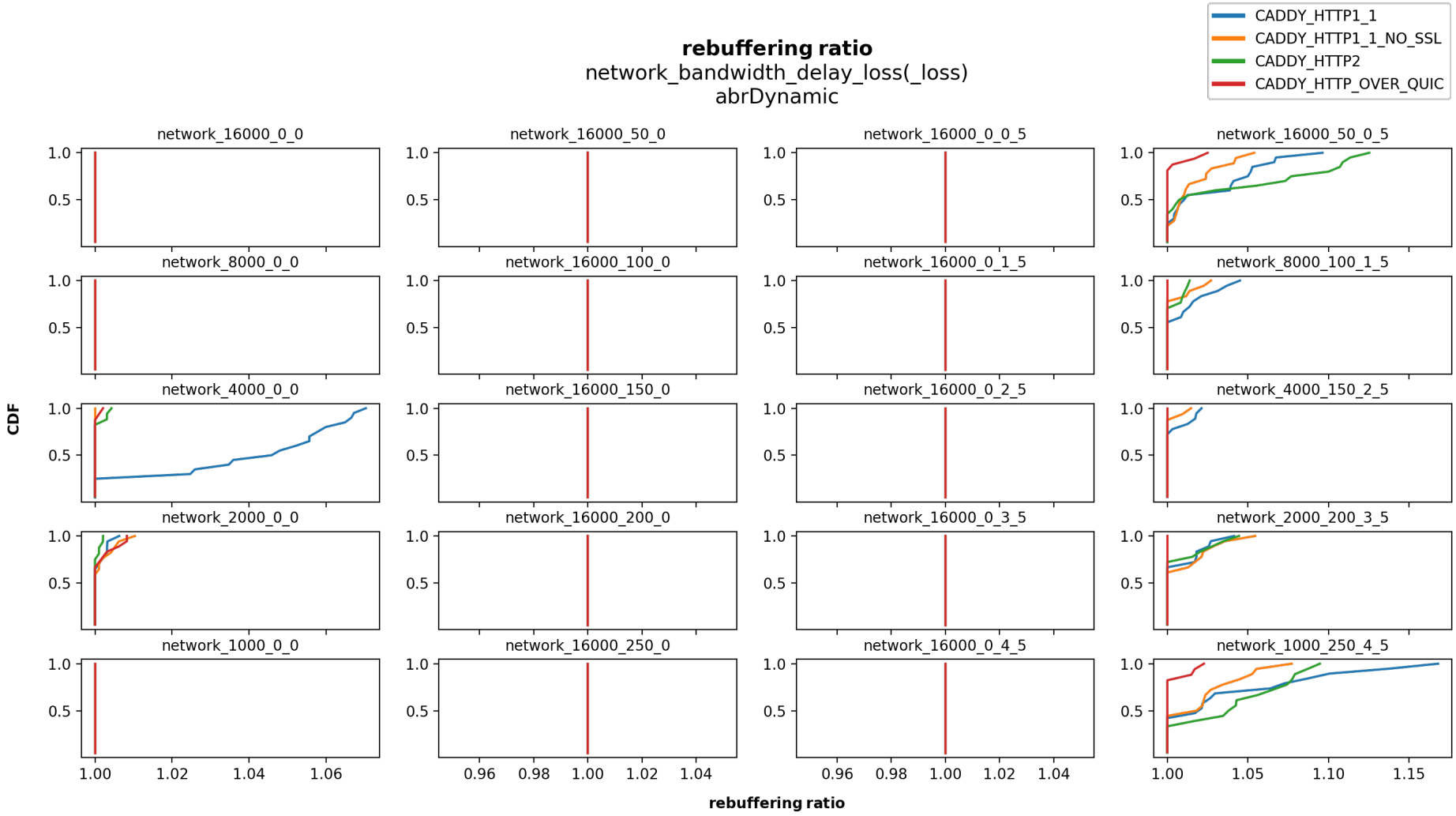

Figure 30: QoE - rebuffering ratio

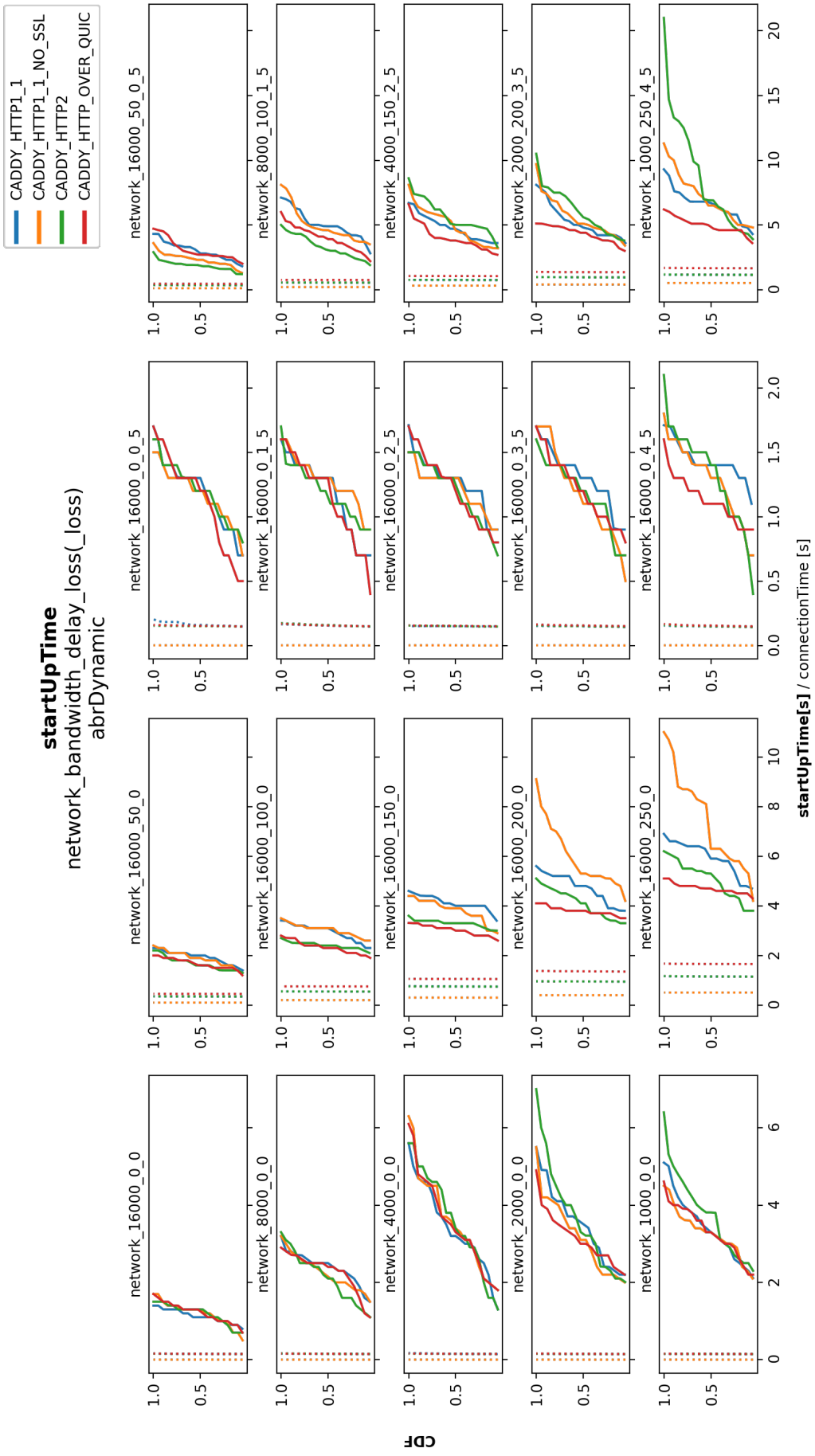

Figure 31: QoE - startup time

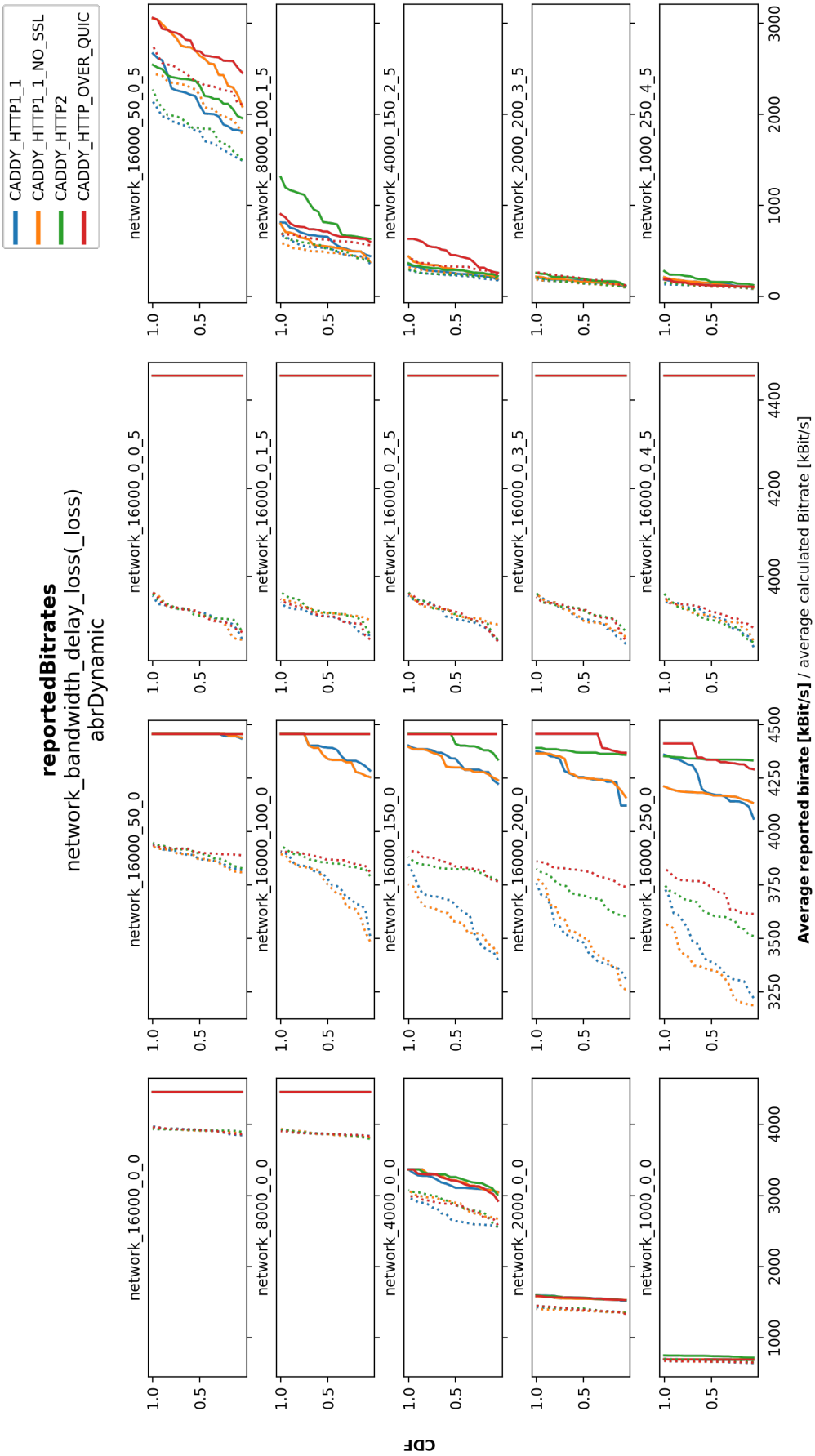

Figure 32: QoE - reported bitrate

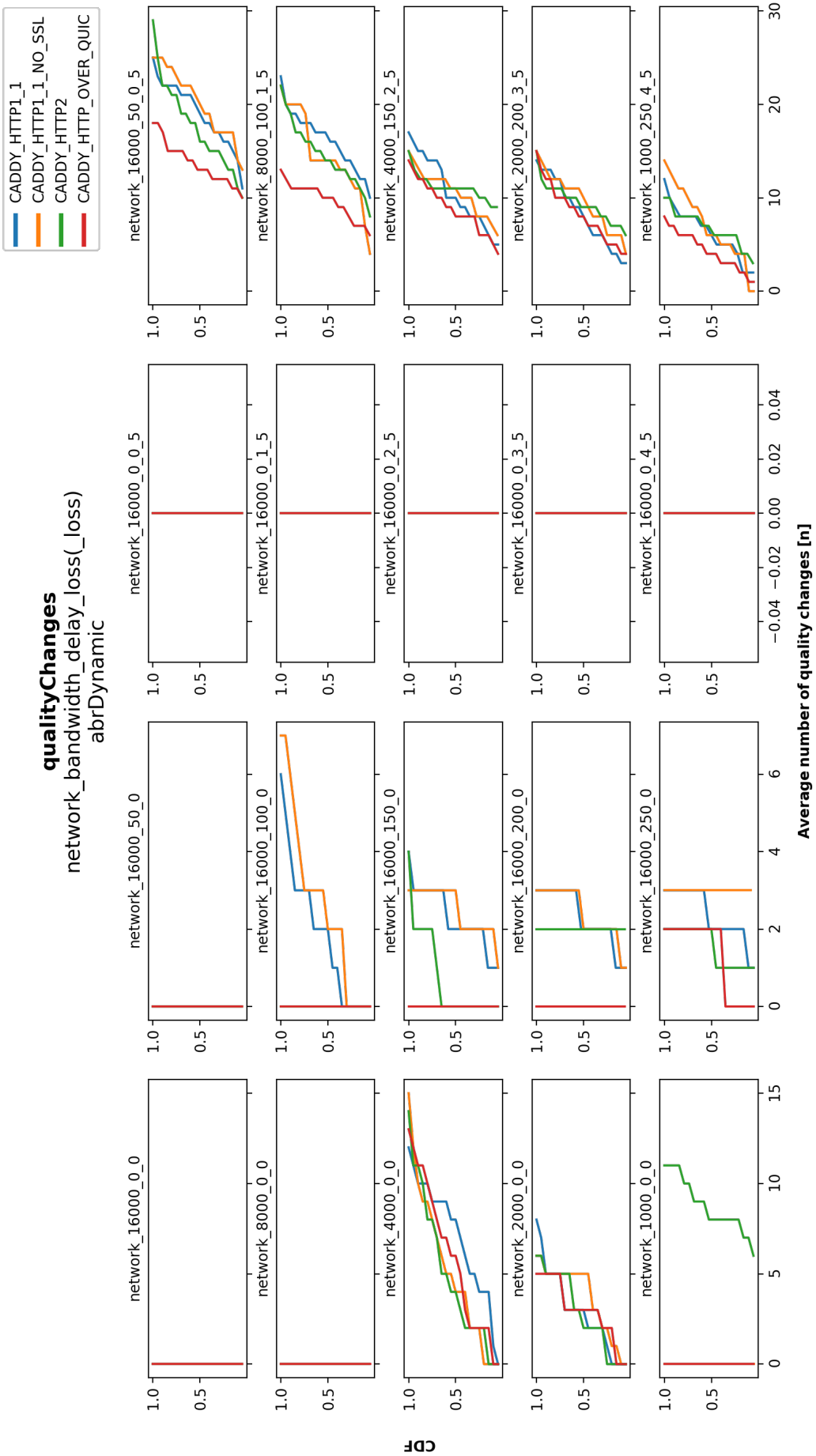

Figure 33: QoE - quality changes

9.4.3. Adaptation Performance

The final test evaluated the interaction between adaptation control algorithm and transport layer congestion control. The black line indicates the available bandwidths (~1-5 Mbit/s) over time and the coloured lines represent the respective observations of bitrate selection. The selected (reported) bitrates are in bold and according with the legend in Figure 34.

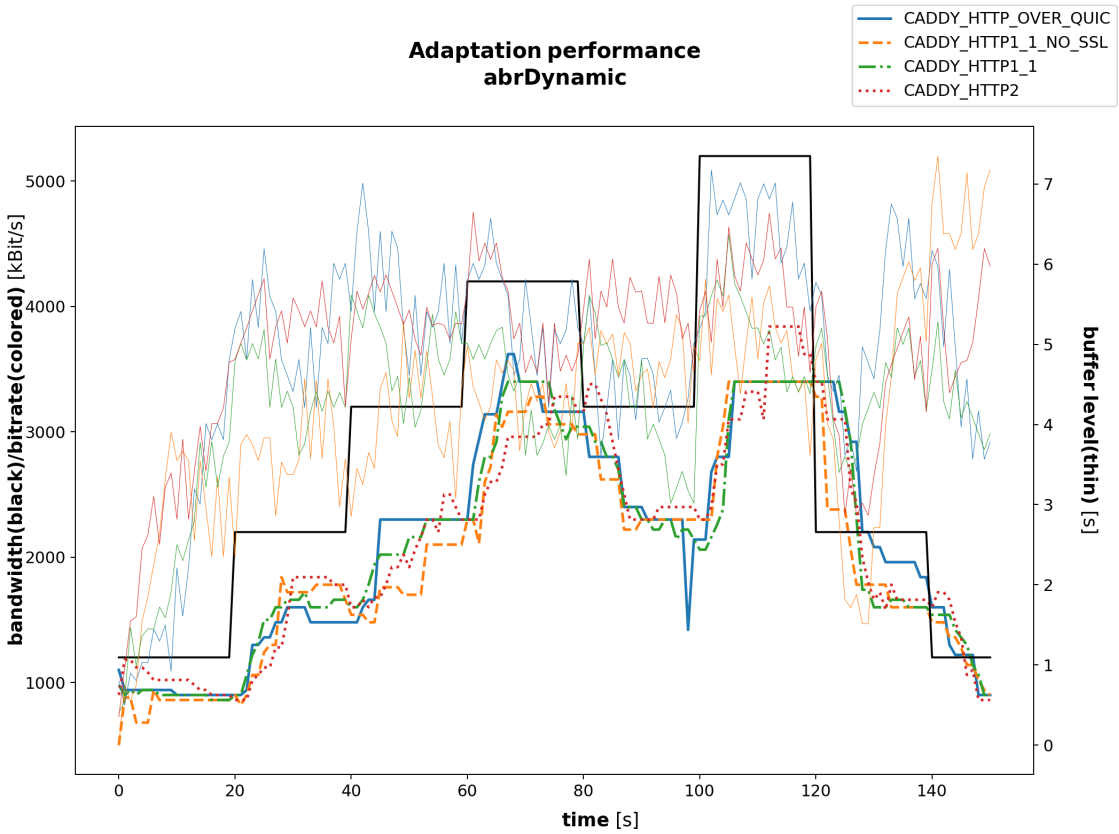

Figure 34: Adaptation performance test

Responsiveness, i.e., the time interval until the change of bandwidth was realised (reacted to), was similar among all protocols. The offset between the available bitrate and the selected bitrate is due to the latency of how network conditions are applied. HTTP/2-over-QUIC showed slightly increased bitrate stability compared to other protocols.

Overall adaptation logic did not seem to be significantly affected by the choice of transport protocol or HTTP/2 version. This corresponds to the fact that all protocols use similar mechanisms for congestion control and could change with the fast evolving CC of QUIC.

Conclusion and Outlook

The goal of this thesis was to analyse how and to which extent HTTP versions, including the use of different transport protocols, impacted HTTP streaming performance. In the beginning, concepts of streaming media over HTTP were explained and related to the underlying network protocols. Afterwards, a qualitative analysis of the context-related elements of the network protocol stack established key differences between the transport protocols TCP and QUIC and how they relate to different HTTP versions. Furthermore, the workflow and technologies of live HTTP adaptive streaming (HAS) were elaborated, followed by industry standards for measuring HAS performance. Together with prior qualitative protocol comparisons, this knowledge was used to design, implement and discuss a testbed which practically evaluates these differences in a simplified live streaming environment. The metrics included an overhead analysis, the impact on QoE in varying network conditions as well as the adaptation performance during changing bandwidths.

The qualitative analysis showed that HTTP messages required the delivery over a reliable transport protocol. Therefore, differences between HTTP versions mainly consisted of the mechanisms for reliability, connection management and the efficiency of sending HTTP messages over the network. Noticeable improvements on the transport layer included QUIC's multiplexing without head-of-line blocking along with an improved connection establishment.

In the testbed, results confirmed the improvements of using QUIC as transport protocol in degraded network conditions. HTTP-over-QUIC added slightly more network overhead than HTTP/2 and HTTP/1.1 over TCP in networks of high bandwidth, small delay and no loss, but not enough to negatively affect QoE. In contrast, in degraded networks of low bandwidth, high delay and high loss-rates HTTP-over-QUIC significantly improved QoE with overall lower rebuffering ratio, higher average selected bitrates at greater stability and faster starting streams.

HTTP-over-QUIC is currently being standardised as HTTP/3 and primary streaming metrics, of the current state of the protocol, promise a significant increase in QoE, compared to HTTP/2 or HTTP/1.1. This performance improvement could be at the cost of energy inefficiency, due to the fact that QUIC runs in user-space, and not like TCP in an optimised kernel. Before deploying QUIC, further research in this area is necessary, particularly when considering global warming and video streaming’s impact on global energy consumption.

Apart from that, the HTTP streaming industry is fast evolving and recent developments involve scalable low latency streaming over HTTP. The basic idea is to make parts of segments (fragments) available before the encoding of a segment is complete (which is possible with the new container format CMAF or a special version of HLS) [MUX]. This removes the previous lower bound for live streaming delays of one segment’s length (~2s) which in reality is even higher, because media players commonly store three segments before playing, in order to cope with variable network conditions. HTTP/3’s low latency connection establishment in conjunction with better multiplexing, potentially further improves this workflow and could therefore be an interesting topic for research.

Appendix

Document structure of the deliverable:

```
/
├── BA_KOCH_2019_ID_11105043.pdf          Bachelor's Thesis as PDF
│
├── Sources                               All sources in PDF format
│
├── README.txt                            Steps and Requirements to
│                                         Reproduce the testbed
├── Client
│   ├── adaption_performance_scripts
│   │   ├── adaptation_performance.js     Script to collect data
│   │   ├── adaptation_plot.py            Create graphs from data
│   │   ├── apply_network.sh              Helper to adapt network
│   ├── data                              data of the tests
│   ├── graphs                            graphs of the tests
│   ├── overhead_scripts
│   │   ├── overhead.py                   Create graphs from data
│   │   └── overhead_analysis.sh          Script to collect data
│   └── qoe_scripts
│       ├── QoE.js                        Helper to collect data
│       ├── QoE_test.sh                   Script to collect data
│       └── plot_cdf.py                   Create graphs from data
│
├── FFMPEG\ COMMANDS.txt                  FFMPEG pre-encoding
│                                         commands
├── Server
│   ├── Dockerfile                        Dockerfile for Caddy
│   ├── conf
│   │   ├── Caddyfile_H1                  Caddy HTTP/1.1 config
│   │   ├── Caddyfile_H2                  Caddy HTTP/2 config
│   │   ├── Caddyfile_QUIC                Caddy HTTP-over-QUIC conf
│   │   └── ssl                           Chained TLS certificates
│   ├── root
│   │   ├── adaptive_dash.mp4             Prepared streaming media
│   │   ├── dash.all.min.js               DASH.js sources
│   │   ├── dash_client.html              HAS client + metrics
│   │   ├── index.html                    Bachelor's Thesis as PDF
│   │   ├── overhead
│   │   │   └── stream.mpd                Static MPD for overhead
│   │   ├── single_bitrate_client.html    HAS client + metrics
│   │   └── stream                        Static Media for overhead
│
│   ├── start_encoder.sh                  Start encoder in docker
│   └── start_servers.sh                  Start servers in docker
│
├── caddy-testbed.com.pem                 Root certificate for TLS
│
└── Setup_TLS                             How to setup TLS correctly
```

Sources

[AYB] Alisu, Yildiz, Begen. Game of protocols - Is QUIC ready for prime time streaming. 07/2018. URL: https://doi.org/10.1002/nem.2063 (accessed May 2019)

[BIT1] Lederer. Optimal Adaptive Streaming Formats MPEG-DASH & HLS Segment Length 04/2015. URL: https://bitmovin.com/mpeg-dash-hls-segment-length/ (accessed July 2019)

[BIT2] Mueller. MPEG-DASH (Dynamic Adaptive Streaming over HTTP). 04/2015. URL: https://bitmovin.-com/dynamic-adaptive-streaming-http-mpeg-dash/ (accessed April 2019)

[BitQ] Timmerer, Bertoni. ADVANCED TRANSPORT OPTIONS FOR THE DYNAMIC ADAPTIVE STREAMING OVER HTTP. June 2016. URL: https://www.researchgate.net/publication/303750010_Advanced_Transport_Options_for_the_Dynamic_Adaptive_Streaming_over_HTTP (accessed July2019)

[BMVI] BMVI. Aktuelle Breitbandverfügbarkeit in Deutschland (Stand Ende 2018). 2018. URL: https://ww-w.bmvi.de/goto?id=451090 (accessed may 2019)

[BUF] Shuai. Dynamic Adaptive Video Streaming with minimal buffer sizes. 2018. URL: https://publikatio-nen.sulb.uni-saarland.de/bitstream/20.500.11880/27179/1/Diss_YS_final.pdf (accessed may 2019)

[CDVL] Codavel. QUIC vs TCP+TLS - and why QUIC is not the next big thing. 09/2019. URL: https://medi-um.com/codavel-blog/quic-vs-tcp-tls-and-why-quic-is-not-the-next-big-thing-d4ef59143efd (acces-sed July 2019)

[ChrQ] The Chromium Projects. QUIC, a multiplexed stream transport over UDP. URL: https://www.chro-mium.org/quic

[CisQ] Cisco IoS Technologies. Quality of Service. - .URL: https://www.cisco.com/c/en/us/products/ios-nx-os-software/quality-of-service-qos/index.html (accessed July 2019)

[ClQ] Jones. Get a head start with QUIC. 2018. URL: https://blog.cloudflare.com/head-start-with-quic/ (accessed May 2019)

[Corb] Corbillon. Enable the next generation of interactive video streaming. 02/19 . URL: https://tel.archi-ves-ouvertes.fr/tel-02011760/document (accessed may 2019)

[DASN] A Testbed to compare Dynamic Adaptive Streaming over HTTP network configurations. 16. URL: https://www.duo.uio.no/bitstream/handle/10852/53112/master_peder_thorup_final.pdf?sequence=8&isAllowed=y (accessed June 2019)

[DIF] DASH-Industry-Forum. ABR Logic. 11/2018. URL: https://github.com/Dash-Industry-Forum/dash.js/wiki/ABR-Logic (accessed July 2018)

[EYE] Eyevinn Technology. How to build your own streaming video HTML player. 03/2018. URL: https://medium.com/@eyevinntechnology/how-to-build-your-own-streaming-video-html-player-6ee85d4d078a (accessed July 2018)

[exM] Ex Machina. The Ultra-Low Latency video streaming roadmap:from WebRTC to CMAF. 03/2018. URL:https://medium.com/exmachinagroup/the-ultra-low-latency-video-streaming-roadmap-from-webrtc-to-cmaf-5b0d8b4ceec2 (accessed July 2018)

[EYE2] Eyevinn Technology. Demystifying the HTML5 Video Player. 10/2017. URL: https://medium.com/@eyevinntechnology/demystifying-html5-video-player-e480846328f0 (accessed July 2018)

[FECQ] Michel, De Coninck, Bonaventure. QUIC-FEC: Bringing the benefits of Forward Erasure Correction to QUIC. URL: https://arxiv.org/abs/1904.11326

[FLS] McAfee Labs. Despite Decline in Use of Adobe Flash, Vulnerabilities Will Continue to Cause Concern. 04/2018. URL: https://securingtomorrow.mcafee.com/other-blogs/mcafee-labs/despite-decline-use-adobe-flash-vulnerabilities-will-continue-cause-concern/ (accessed June 2019)

[FSTL] Iyengar. Why Fastly loves QUIC and HTTP/3. 03/2019. URL: https://www.fastly.com/blog/why-fastly-loves-quic-http3 (accessed May 2019)

[GlMF] encoding.com. 2019 Global Media Format Report. 2019. URL: https://www.encoding.com/blog/2019/04/04/encoding-com-releases-its-2019-global-media-format-report/ (accessed June 2019)

[GQ] A. Langley et al. The QUIC Transport Protocol: Design and Inter-Scale Deployment. 2017. URL: https://static.googleusercontent.com/media/research.google.com/de//pubs/archive/46403.pdf (accessed May 2019)

[GreT] Greer. TCP Series. 07/2017. URL: https://accedian.com/enterprises/blog/network-packet-loss-retransmissions-and-duplicate-acknowledgements/ (accessed May 2019)

[h3] QUIC Working Group. Hypertext Transfer Protocol Version 3(HTTP/3). 07/2019. URL: https://quicwg.org/base-drafts/draft-ietf-quic-http.html (accessed: August 2019)

[hpbn1] O'Reilly. Networking 101, Chapter 2 Builiding Block of TCP. URL: hpbn.co

[IFB] The linux foundation. netem. last modified 09/2018. URL: https://wiki.linuxfoundation.org/networking/netem#how_can_i_use_netem_on_incoming_traffic3f (accessed July 2019)

[LnX] The linux foundation. netem. last modified 09/2018. URL: https://wiki.linuxfoundation.org/networking/netem#how_can_i_use_netem_on_incoming_traffic3f (accessed July 2019)

[Marx] Marx. QUIC and HTTP/3: Too big to fail?!. 12/2019. URL: https://calendar.perfplanet.com/2018/quic-and-http-3-too-big-to-fail/ (accessed July 2019)

[MDN] MDN web docs. Evolution of HTTP. last edited 05/2019. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP (accessed July 2019)

[Mil] Millicast. Millicast 2.o and Simulcast - Why you want it - Why nobody else has it. 04/19. URL: https://medium.com/millicast/millicast-2-0-and-simulcast-why-you-want-it-why-nobody-else-has-it-945644a596e1 (accessed July 2019)

[MUX] Cluff. The Low Latency Live Streaming Landscape in 2019. 01/19. URL: https://mux.com/blog/the-low-latency-live-streaming-landscape-in-2019/ (accessed June 2019)

[NET1] Dordal. An Introduction to Computer Networks (Release 1.9.19) - Chapter 1.3 Packets. 07/19. URL: http://intronetworks.cs.luc.edu/current/ComputerNetworks.pdf

[NET2] Dordal. An Introduction to Computer Networks (Release 1.9.19) - Chapter 7.1The IPv4 Header. 07/19. URL: http://intronetworks.cs.luc.edu/current/ComputerNetworks.pdf

[NET3] Dordal. An Introduction to Computer Networks (Release 1.9.19) - Chapter 11 UDP Transport. 07/19. URL: http://intronetworks.cs.luc.edu/current/ComputerNetworks.pdf

[NET4] Dordal. An Introduction to Computer Networks (Release 1.9.19) - Chapter 13 TCP Reno and Congestion Management. 07/19. URL: http://intronetworks.cs.luc.edu/current/ComputerNetworks.pdf

[Osi2] Bahl. OSI Model Layers - “Explained”. 04/18. URL: https://medium.com/@madhavbahl10/osi-model-layers-explained-ee1d43058c1f (accessed May 2018)

[QC22] IETF. draft-ietf-quic-transport-22. 07/19. URL: https://tools.ietf.org/html/draft-ietf-quic-transport-22#section-4 (accessed July 2018)

[QoE] Bovik, Yeganeh. Study of the effects of stalling events on the quality of experience of mobile streaming videos. 02/2015. URL: https://www.researchgate.net/publication/283229572_Study_of_the_effects_of_stalling_events_on_the_quality_of_experience_of_mobile_streaming_videos (accessed July 2019)

[QWG] QUIC Working Group. Using TLS to Secure QUIC. 07/2019. URL: https://greenbytes.de/tech/webdav/draft-ietf-quic-tls-latest.html#protocol-overview (accessed: June 2019)

[SGIR] Sandvine. The Global Internet Phenomena Report 2018. 10/2018. URL: https://www.sandvine.com/phenomena (accessed July 2019)

[SWGDE] SWGDE. Technical Overview of Digital Video Files. 07/2017. URL: https://www.swgde.org/documents (accessed July 2019)

[Sl17] Dreier. The State of Live Video 2017. 03/2017. URL: https://www.streamingmedia.com/Articles/Editorial/Featured-Articles/The-State-of-Live-Video-2017-116974.aspx (accessed May 2019)

[SRI] Srivastava. Performance Analysis of QUIC Protocol under Network Congestion. 05/19. URL: http://web.cs.wpi.edu/~claypool/ms/quic/quic-thesis.pdf (accessed May 2019)

[StrM] Streaming Media. Video: How Do We Define Quality of Experience for Streaming Video? 10/2018. URL: https://www.streamingmedia.com/Articles/ReadArticle.aspx?ArticleID=126354 (accessed June 2019)

[Sys] Peterson, David. Computer Networks: A Systems Approach. 2012. URL: https://book.systemsapproach.org/congestion/queuing.html

[THb] Dennstedt. Analyse der Robustheit und Ausfallsicherheit von Live-HTTP-Streaming Systemen. Bachelor Thesis TH Köln.

[thrG] GoogleChrome. throttling. last updated 05/19. URL: https://github.com/GoogleChrome/lighthouse/blob/master/docs/throttling.md (accessed July 2019)

[Tri] Saloni. HTTP Video Streaming. The Quality - Reliability - Latency Triangle. 02/2019. URL: https://event.unitn.it/smartcities-trento/Trento_WP_Saloni2.pdf (accessed June 2019)

[UnS] Unified Streaming. Stop Numbering: The under appreciated power of DASH's SegmentTimeline. 2018. URL: https://www.unified-streaming.com/blog/stop-numbering-underappreciated-power-dashs-segmenttimeline (accessed June 2018)

[w3t] w3tech. Usage statistics of QUIC for websites. August 2019. URL: https://w3techs.com/technologies/details/ce-quic/all/all

[WIK2] Wikipedia. URL: https://en.wikipedia.org/wiki/Uniform_Resource_Identifier (accessed July 2019)

[WIK3] Wikipedia. Best effort delivery. URL: https://en.m.wikipedia.org/wiki/Best-effort_delivery (accessed July 2019)

[WkP] Wikipedia. Progressive Download. URL: https://en.wikipedia.org/wiki/Progressive_download#cite_-note-2 (accessed June 2019)

[WZA1] Ruether. What Is CMAF. 01/19. URL https://www.wowza.com/blog/what-is-cmaf (accessed May 2019)

List of Figures

Figure 1: HTTP streaming technologies 2

Figure 2: HTTP Adaptive Streaming concept by [BUF] 4

Figure 3: Network overhead, encapsulation and the OSI model, Zhang 5

Figure 4: Ethernet header, wikipedia 6

Figure 5: IPv4 header, Cisco 7

Figure 6: UDP header, wikipedia 7

Figure 7: 3-way-handshake in TCP by [hpbn1] 10

Figure 8: TCP Congestion Control and Congestion Avoidance by [hpbn1] 11

Figure 9: TCP header by Cisco 12

Figure 10: HTTP/2 over QUIC compares to HTTP/2+TLS over TCP, IETF 12

Figure 11: Connection Establishment TCP, TCP+TLS and QUIC (1.Repeat connection, 2.First connection), chromium 14

Figure 12: Connection Establishment QUIC [GQ] 14

Figure 13: QUIC: stream multiplexing without head-of-line blocking. 16

Figure 14: QUIC's header is more secure than that of TCP [Marx] 16

Figure 15: In blue the standardised HTTP versions of the IETF and in green google's proprietary protocols as predecessors 17

Figure 16: The binary framing layer of HTTP/2. The textual HTTP header gets encoded into a HEADERS frame and the body is parsed in a DATA frame 18

Figure 17: HTTP Adaptive Streaming workflow, own after Famaey 20

Figure 18: What happens in the DASH encoder - Timmerer at Bitmovin 21

Figure 19: Media segments need to start with IDR Frames to allow switching between representations of different bitrates. 23

Figure 20: An illustration of how digital containers store video, audio and meta data 24

Figure 21: Advantages of storing media in CMAF [exM] 24

Figure 22: Javascript + MSE for streaming protocol support [EYE2] 28

Figure 23: Bursty behaviour of HAS players [BUF] 28

Figure 24: Structure of the MPD [DASN] 31

Figure 25: Testbed streaming server setup: one encoder container, three server containers 39

Figure 26: Complete testbed setup to emulate network conditions 41

Figure 27: A ping test confirms the emulated network 42

Figure 28: An iperf speed test also confirms the emulated network 42

Figure 29: The main HTML site contains DASH.js and a simple javascript logic for metrics 43

Figure 30: Measured network overhead (coloured), estimate of the underlying protocols based on the observed MTU (greyscale). The network is not lossy and has a RTT < 10 ms. 48

Figure 30: QoE - rebuffering ratio 52

Figure 31: QoE - startup time 53

Figure 32: QoE - reported bitrate 54

Figure 33: QoE - quality changes 55

Figure 34: Adaptation performance test 56

List of Tables, Codes, Formulas

Table 1: Features of the standardised HTTP/1.1 and 2.0 and the new HTTP/2-over-QUIC which will soon become HTTP/3 18

Table 2: Last-mile streaming protocols 24

Table 3: Streaming protocol and MSE support in modern browsers according to caniuse.com 26

Table 4: HAS-protocol feature comparison: green = yes, red = no 31

Table 5: Pre-transcoded video and the resulting bitrates 36

Table 6: Server Configurations 37

Table 7: Server Hardware 41

Table 8: Client Hardware 42

Table 9: QoE Test - 20 Emulations of varying bandwidth, delay and loss 46

Code 1: Client-sided script to manage network conditions 42

Code 2: Recording metrics for overhead calculation 45

Formula 1: Network overhead 45

Formula 2: Rebuffering ratio 46

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorgelegte Abschlussarbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Köln, den 07.08.2019

Unterschrift

Martin Emanuel, Koch

TH Köln
Gustav-Heinemann-Ufer 54
50968 Köln
www.th-koeln.de

Technology
Arts Sciences
TH Köln