UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Real-time analytics of concurrent adaptive video streams using next-generation protocols

Mike Vandersanden

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Peter QUAX

2021
2022

De transnationale Universiteit Limburg is een uniek samenwerkingsverband van twee universiteiten in twee landen: de Universiteit Hasselt en Maastricht University.

UHASSELT
KNOWLEDGE IN ACTION

www.uhasselt.be
Universiteit Hasselt
Campus Hasselt:
Martelarenlaan 42 | 3500 Hasselt
Campus Diepenbeek:
Agoralaan Gebouw D | 3590 Diepenbeek

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Real-time analytics of concurrent adaptive video streams using next-generation protocols

Mike Vandersanden

Scriptie ingediend tot het behalen van de graad van master in de informatica

PROMOTOR :

Prof. dr. Peter QUAX

Universiteit Hasselt

Master's thesis nominated for obtaining a master's degree in computer science

Networking and Security – Real-Time Analytics of Concurrent Adaptive Video Streams using Next-Generation Protocols

Author:

Mike Vandersanden

Promotor:

Prof. Dr. Peter Quax

Co-promotor:

Prof. Dr. Wim Lamotte

Mentors:

Drs. Joris Herbots
Dr. Maarten Wijnants

Academic Year 2021-2022

Acknowledgements

I would like to recognize the invaluable assistance of all the people who stood by my side throughout accomplishing this thesis, this would not have been possible without them.

Foremost, I am extremely grateful for the guidance given by Drs. Joris Herbots. Working together, and spending many hours discussing numerous subjects, resulted in insights that I would not have had otherwise. The frequent feedback I was able to receive made sure the thesis always kept a pace towards the objectives.

This thesis has been made in collaboration with the Networking and Secured Systems research department. I would also like to thank these people, with extra gratitude towards Dr. Maarten Wijnants for the mentoring he did, my promotor Prof. Dr. Peter Quax, and co-promotor Prof. Dr. Wim Lamotte. I would also like to acknowledge the help of Dr. Robin Marx, for the few but insightful conversations we had. My gratitude goes towards the assessors for taking the time to read this thesis.

Last, but definitely not least, I would like to thank all my friends and family to keep me sane and motivated. Explicitly, Olaf and Arno for joining me during "QUIC @ EDM – summer edition", and participating in the many discussions that arose. And I am grateful for the proofreading by Anne, Laura and Paulien, who made sure the text was clear.

Abstract

Video streaming is a significant portion of the daily activity on the Internet. With growing user bases, streaming over the HyperText Transfer Protocol (HTTP) is becoming the popular streaming method. HTTP Adaptive Streaming (HAS) provides adaptability to the network, by making in-the-moment decisions that will ensure the best possible experience for the user, influenced by the current network environment. Another benefit of using HTTP is being able to leverage the existing architectures that improve the performance of HTTP, such as a Content Delivery Network (CDN). By utilizing a next-generation of network protocols, the performance of HAS can be further improved. Streaming using the next-generation protocols HTTP/3 and QUIC can eradicate certain flaws in the more traditional solution that uses HTTP/1.1 and TCP.

Improvements can only be made when the flaws are first uncovered. A streaming client generates a plethora of logging data, that can be analyzed. Insights can be gained, and flaws will emerge. This thesis proposes an analysis service that can ingest the logging data in real-time and visualize it in interesting ways, in order to ease this analysis process. To facilitate the testing of applications in an academic setting, a framework can be used to ensure consistent tests. The evaluation of this thesis links the proposed analysis service to a testing framework in order to inspect the logging data of video streaming applications. Resulting in not just the ability to analyze the streaming session of multiple streaming clients, while they are streaming, but also being able to infer events of outside forces, such as network middleboxes, other streaming clients and various applications competing for network bandwidth.

Samenvatting

Videostreaming is een aanzienlijk deel van de dagelijkse activiteit op internet. Met een groeiend aantal gebruikers wordt streaming via het HyperText Transfer Protocol (HTTP) de populaire streamingmethode. HTTP Adaptive Streaming (HAS) biedt de mogelijkheid om zich aan het passen aan het netwerk door beslissingen te nemen wanneer het netwerk zich anders begint te gedragen. Dit garandeert de best mogelijke ervaring voor de gebruiker, gegeven de huidige netwerk omgeving, zoals geïllustreerd in figuur 1. Een ander voordeel van over HTTP te streamen is de mogelijkheid om gebruik te maken van de bestaande oplossingen die de prestaties van HTTP verbeteren, zoals een Content Delivery Network (CDN). Door gebruik te maken van een nieuwere netwerkprotocollen, kunnen de prestaties van HAS verder worden verbeterd. streaming met behulp van de nieuwer protocollen HTTP/3 en QUIC kan bepaalde fouten wegwerken in de meer traditionele oplossing, die streamt met HTTP/1.1 en TCP. Deze protocollen worden vergeleken in figuur 2.

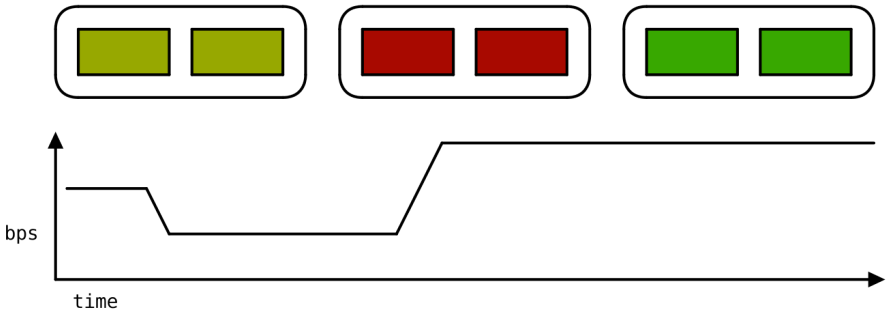


Figure 1: Een reeks videosegmenten, gestreamd door de applicatie, en de beschikbare bandbreedte in op ieder moment, uitgedrukt in bits per seconde (bps). De kleur van de segmenten representeert de kwaliteit, groen is de hoogste kwaliteit, rood de laagste kwaliteit. Telkens wanneer de bandbreedte daalt, heeft het volgende segment een lagere kwaliteit, terwijl wanneer de bandbreedte weer stijgt, de kwaliteit van het gekozen segment ook hoger zal zijn.

Verbeteringen kunnen alleen worden aangebracht wanneer de gebreken eerst worden ontdekt. Een streamingapplicatie maakt verschillende datapunten aan over de streamingsessie, die kunnen worden geanalyseerd, en zo worden er fouten ontdekt. Deze thesis stelt een analyseprogramma voor die de data van applicaties live kan opnemen en op interessante manieren kan visualiseren, om het analyseproces te vergemakkelijken. Om het testen van applicaties in een academische setting eenvoudiger te maken, kan een raamwerk worden gebruikt om de consistentie van testen te garanderen. De evaluatie van deze thesis koppelt het voorgestelde analyseprogramma aan een raamwerk om videostreamingapplicaties te testen, dit wordt getoond in figuur 3. Dit raamwerk zorgt ervoor dat testen worden uitgevoerd, en de gegenereerde data van de applicaties kan dan geanalyseerd worden door het analyseprogramma. Het resultaat is niet alleen de mogelijkheid om de streamingsessie van meerdere streamingapplicaties te analyseren, terwijl ze aan het streamen

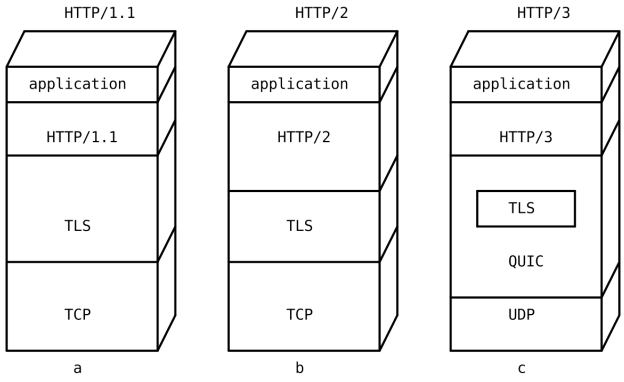


Figure 2: Een figuur waarin HTTP/1.1 (a), HTTP/2 (b) en HTTP/3 (c) worden vergeleken. HTTP/1.1 en HTTP/2 houden zich beide aan het gelaagde model. HTTP/3 houdt zich niet aan het gelaagd model, door de integratie van TLS in QUIC. QUIC heeft daarnaast ook overlap met TCP en HTTP/2.

zijn, maar ook om gebeurtenissen van externe krachten af te leiden, zoals andere apparaten die tussen de applicatie en de server staan, andere streamingapplicaties, of verschillende applicaties die strijden om netwerkbandbreedte.

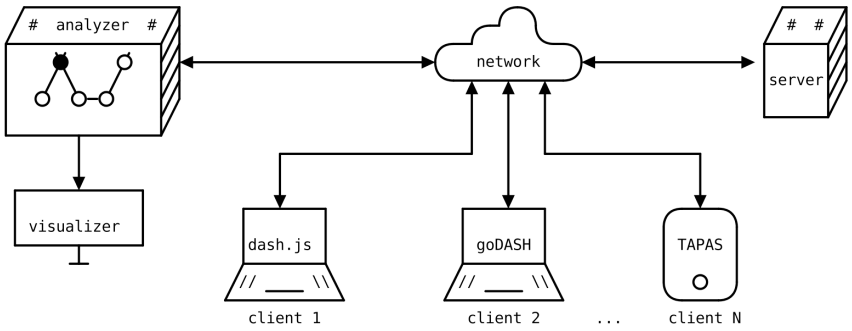


Figure 3: De onderdelen van het test- en analyseraamwerk. Link bevindt zich het analyseprogramma, bestaande uit de *analyzer* en de *visualizer*. Aan de rechterkant staat het analyseraamwerk waar testen voor videostreamapplicaties mee kunnen worden uitgevoerd.

Het is belangrijk om over de juiste datapunten te beschikken, zonder de juiste data is het onmogelijk om een goede analyse te bekomen. Zo wordt de doorvoersnelheid van het netwerk niet gemeten in de applicatie, omdat de applicatie niet weet hoeveel data de lagere lagen toevoegen, maar zal er gekozen worden om de nuttige data die doorgekomen is te meten, genaamd de *goodput*. Daarnaast is het ook belangrijk om de interessante gebeurtenissen op te vangen, zoals buffering van de streaming applicatie, omdat een negatief effect heeft op de gebruikerservaring. Nadat de data binnen is gekomen bij het analyseprogramma moet de analyse uitgevoerd worden. Om te beginnen moet de data op interessante manieren gevisualiseerd worden, dit is een uitdaging. Figuur 4 toont hoe een dataset die verbindingen tussen apparaten op verschillende manieren gevisualiseerd kan worden. Iedere visualisatie heeft zijn eigen voor- en nadelen, en zal een bepaald attribuut van de dataset aan het licht brengen. Een belangrijk aspect van de evaluatie was dan ook om visualisaties op te stellen die de juiste data op de correcte manier toont.

Tijdens de evaluatie wordt er gebruikt gemaakt van het raamwerk om te verzekeren dat de gekozen parameters vast staan. Er worden dan meerdere testen uitgevoerd, elke test past een bepaalde parameter aan. Er kan dan vergeleken worden tussen de resultaten van deze testen om

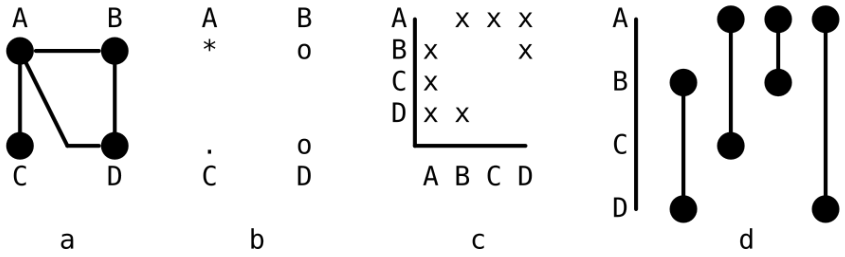


Figure 4: Verschillende visualisaties van eenzelfde dataset die een netwerk voorstelt. (a) De linkmap toont verbindingen tussen apparaten. (b) De knooppuntenkaart, gebruikt pictogrammen om gegevens over te brengen voor ieder apparaat, in dit geval vertegenwoordigt het pictogram het aantal totale verbindingen naar een apparaat. (c) De matrixweergave geeft aan of twee apparaten een verbinding hebben door het snijpunt te markeren. (d) De reeksweergave, het laat zien welke verbindingen verschijnen op welk moment in de tijd.

de impact van de aangepaste parameter vast te stellen. Zo is er vastgesteld dat het mogelijk is om de data die applicaties genereren live door te sturen naar het analyseprogramma, maar dat hiervoor genoeg bandbreedte nodig is. Daarnaast moet de manier waarop de data doorgestuurd wordt ook aangepast worden aan de huidige omstandigheden. Figuur 5 toont hoe verschillende methoden van data doorsturen een impact kunnen hebben op de streamingervaring. Andere testen tonen bijvoorbeeld dat meerdere applicaties die tegelijkertijd streamen zeer gelijkaardige metingen kunnen doen van het netwerk. Maar zoals figuur 6 aantoont, betekent dit niet altijd dat de applicaties dezelfde ervaring zullen hebben. De kleine verschillen tussen de metingen van deze applicaties zorgen ervoor dat een applicatie veel meer negatieve gebeurtenissen ervaart in vergelijking met de andere applicatie.

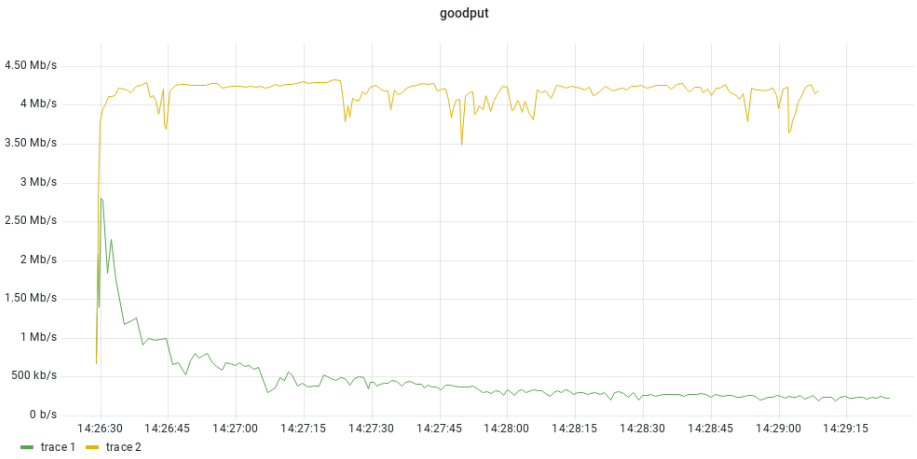


Figure 5: Het verschil in *goodput* tussen het streamen van de data gegenereerd door de applicatie een voor een (trace 1) en in groepen van 100 datapunten (trace 2). De groepen zorgen voor minder overlast op het netwerk.

Er kan geconcludeerd worden dat het doel van de thesis bereikt is. Het analyseprogramma kan een videostreamsessie live analyseren, zonder de ervaring negatief te beïnvloeden. Maar er is nog manuele interventie nodig om dit tot een goed einde te brengen. Niet alleen moet de applicatie juist ingesteld worden om de negatieve impact te voorkomen, de analyse zelf moet volledig manueel uitgevoerd worden. De juiste aanpak vinden om dit manueel proces succesvol af te ronden vergt ervaring. Daarnaast is het live aspect van het analyseprogramma niet volledig tot zijn recht kunnen komen in deze opstelling waar het manueel verwerken van de data veel

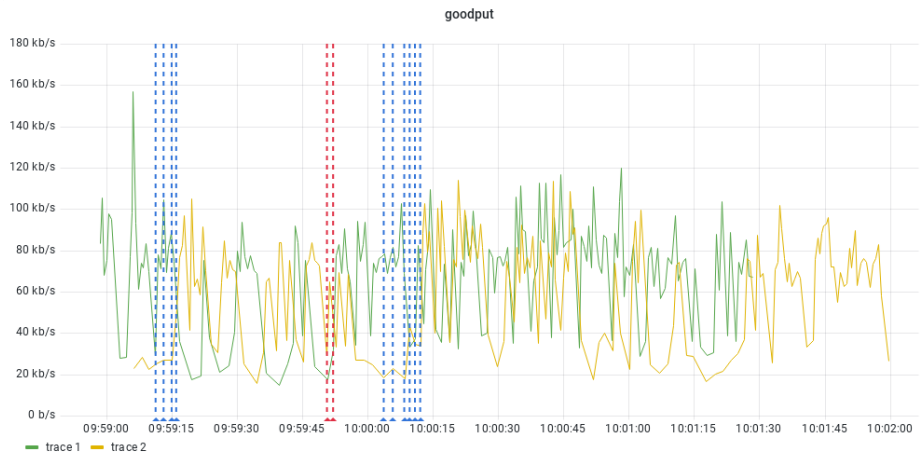


Figure 6: Het verschil in *goodput* tussen twee applicaties die tegelijkertijd aan het streamen waren, terwijl er ook andere applicaties op het netwerk bezig waren. De rode lijnen stelt buffering voor bij trace 1, de blauwe lijnen stelt buffering voor bij trace 2.

trager is dan de data binnenstroomt.

Contents

Chapter 1

Preface

Video streaming makes up a significant portion of the daily Internet traffic. According to Hootsuite, who surveys social media and digital trends, nine out of ten Internet users admit to using a portion of their daily Internet usage to watch any kind of video, while average time spent on the Internet each day is around seven hours [Hoo21]. The share of global internet traffic that goes to video streaming services like Facebook, TikTok, Twitch, and YouTube is growing as social media platforms compete with other platforms by adding rival features to increase the size of their user base and become the most popular [Thu22]. Aside from that, more traditional video streaming services, such as Netflix, Disney+, Amazon Prime, or even the set-top box of a local internet service provider, remain popular and have daily global usage.

With such a large user base, that can get millions of views on a video in a few days, streaming protocols and standards have diverged from traditional architectures in favor of more scalable alternatives that can provide a higher quality of experience. The HyperText Transfer Protocol (HTTP) Adaptive Streaming paradigm allows a streaming session to adapt to the network by leveraging algorithms that can make in-the-moment decisions to prevent events that degrade the experience. By using HTTP to transfer data, it can also take advantage of existing infrastructure and technologies that improve HTTP traffic performance, such as a Content Delivery Network (CDN) [Clo22c].

A next-generation of network protocols is being developed to improve performance even further by eradicating flaws in traditional protocols. HTTP/3 and QUIC are two next-generation protocols that work together and can be used for HTTP Adaptive Streaming instead of the traditional HTTP/1.1 and TCP solution. According to CloudFlare Radar, a service that tracks Internet trends, approximately 30% of all Internet traffic is already transferred using these next-generation protocols, which have been influenced principally by large corporations such as Google, Facebook, and Cloudflare [Clo22a]. This traffic almost certainly includes video streaming.

Analysis of video streaming service and next-generation protocol traffic is already being performed, as evidenced by reports published on a regular basis, such as those by Hootsuite or Cloudflare. The research community and the industry are working together as well, with the objective to come up with novel ideas to improve media delivery over the next-generation protocols [IET22]. These analyses and subsequent improvements will reassure content providers that their promises to deliver content of a certain quality can be maintained.

The objective of this thesis is to investigate whether existing analysis techniques can be used to perform real-time analysis on a video streaming session using next-generation protocols. It has already been established that video streaming services allow their numerous client applications to send back reports while streaming [Adh+12], and data analysis is an established part of

the improvement process of network protocols [Mar+20a]. The next logical step is to analyze these reports as they are returned. Modern streaming services have large user bases, in a household there can be multiple applications streaming concurrently. The interactions between the applications on the networks are suspected to result in events that will be interesting to analyze. It is hypothesized that a content provider can leverage this manner of analysis, proposed by the thesis, in order to inspect streaming session and gain insights about them. This can lead to determining what could be improved to increase the quality of the experience of the users. The sooner insights are gained, the sooner improvements can be made, which is why the real-time aspect seems promising.

This thesis is categorized as a case study. The following research question must be addressed in order to confirm the viability of the goal:

(RQ1): Is it possible to perform real-time analysis of an adaptive video stream when using next-generation protocols, without impacting the performance of the application negatively?

(RQ2): What is the difference in performance, when adaptive streaming using next-generation protocols, between having:

(a) different amounts of concurrent clients

(b) clients with diverse behavior

(c) various network environments

(d) assorted competing applications on the network

To answer the research questions, a solid understanding of various topics is required, ranging from networking and video streaming to simulation and data visualization. A proof-of-concept implementation combines this knowledge, and is needed to perform an assortment of tests. The results of those tests can then be used to provide answers to the questions asked. This process is described in the following chapters:

Chapter 2 An exploration of traditional and next-generation network protocols, and a comparison between them.

Chapter 3 An introduction to video streaming, and a discussion of modern streaming protocols and standards.

Chapter 4 Finding the means to make simulation of video streaming possible and convenient.

Chapter 5 A discussion on what kind of data can be generated by simulation subjects during a video streaming session, and how to facilitate its visualization and analysis.

Chapter 6 Clarifying the contributions made by the thesis, its implementation.

Chapter 7 Records of the evaluation phase.

Chapter 8 A conclusion that reflects on the thesis and its contributions.

Chapter 2

Next-Generation Protocols

A networked application is one that sends data over a communication network. It is made up of the application logic and the supporting network stack, which allows two applications to communicate with one another. Chapter 3 explores relevant streaming protocols for the application logic. Section 2.2 of this chapter looks at transport layer protocols. Following that, in Section 2.3, it investigates relevant application layer transfer protocols.

2.1 Networked Applications

The network stack of a networked application is a collection of protocols that interact with one another to provide the functionalities required to enable communication over a network with an equivalent network stack. In the context of this thesis, a protocol refers to a communication protocol, which is a set of rules that allows communicating entities to exchange data. Not only is the data format enforced, but so is the order in which it is transmitted, and what actions an entity must take when certain events occur [KR17]. Data is passed from one protocol to another like a conveyor belt, with each protocol changing the data to their own representation, for example, by wrapping the data with more metadata or changing the coding. Every representation ensures that the data is understood by the receiving protocol stack and facilitates various functionalities. Next-generation protocols are those that have recently been developed and focus on being evolvable, thereby future-proof [Bom22].

There are several ways to define the unit of communication between two networked entities. For the sake of clarity, this thesis defines packets as the unit of a transport protocol. Messages are the unit of communication in a transfer protocol. Both are structured data that is used to communicate between two entities that are using the same protocol.

The network stack can be represented in a variety of ways, one of which is the seven-layered OSI reference model [Sha22], illustrated in Figure 2.1a. Every layer is defined by its function, with a higher layer building on lower layers to add functionality for the network stack to function as expected by the networked application. A networked application may require secure transmissions, for example, which means that either the application or a layer within the network stack must implement this requirement. The distinction between layer 7, the application layer, and the networked application is critical. A networked application, such as a web browser, an online game, or a video chat app, provides a service to a user via a communication network. Layer 7, on the contrary, handles communication between the application and another entity, such as the File Transfer Protocol (FTP) or the Simple Mail Transfer Protocol (SMTP). An application creates layer 7 messages which transfer data to or request data from another entity in the network, and the protocol executes the data communication using the functionality

provided by lower layers. Layer 6, the presentation layer, receives the message and changes the data representation. To enable secure transmissions, cryptographic encryption and decryption of data is one example. The session layer, layer 5, manages the connection between the sender and the receiver. The session layer establishes the connection, terminates it when the session is over, and ensures that it is available when needed. For example, a socket is a combination of layer 4 port numbers and layer 3 IP addresses that define a connection between two hosts. The layers beneath the session layer determine the behavior of the connection. Aside from port numbers, layer 4, the transport layer, is responsible for controlling how data is transmitted, such as packet size, when a packet is sent, and what to do when a packet arrives and when it does not. The network layer, layer 3, provides network addresses such as IP addresses. Routers and other middleboxes use the addresses to forward packets to the next hop in the network, which leads to the destination. The data link later, layer 2, handles communication between two directly connected entities in the network. Layer 1, the physical layer, enables the entities to transmit packets from one another via a physical medium. The packets are converted into signals that are transmitted over a cable or wirelessly.

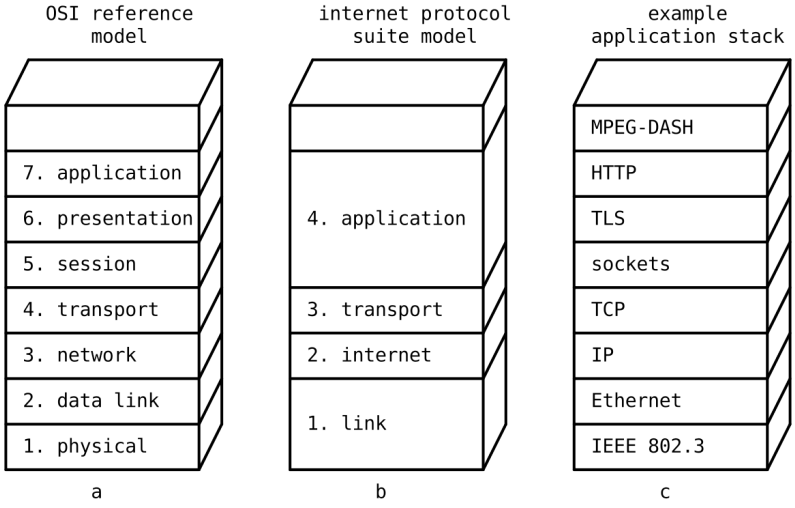


Figure 2.1: (a) The OSI reference model and (b) the internet protocol suite model, abstract representations of a network stack, next to (c) an example of such a stack for a streaming application.

The four-layered Internet Protocol (IP) suite model [Bra89a; Bra89b] is another representation, illustrated in Figure 2.1b. If compared to the OSI model, this model combines all layers above the transport layer into the application layer. The application layer provides the protocols that allow the overarching networked application to communicate with another application. The transport layer is a layer below that performs the same function as the OSI model, providing logical communication and managing communication. The internet layer is very similar to the OSI network layer, since it routes packets through various networks to the right host. Finally, the link layer, which combines the data link and physical layer of the OSI model, handles communication within the local network.

While the network stack encapsulates the lower layers required for network communication, any layers above this stack encompass the higher-level protocols as well as application-specific business logic. The layers of the stack are the building blocks of the application and they work together to ensure that the required features of the application are present [MH21]. Figure 2.1 depicts the OSI and IP suite models, as well as an example stack for a streaming application. A protocol is used to fill in the gaps in each layer of the application. The highest layer,

application-specific business logic, is MPEG-DASH, a streaming protocol described in Section 3.4. These higher-level layers are not defined in the abstract models. This thesis makes several assumptions regarding the network stack of an application. We assume that the lowest layers, the physical layer and the data link layer, transmit data correctly. IP is used for the network layer in all network communication; an application binds to a socket, which establishes the connection.

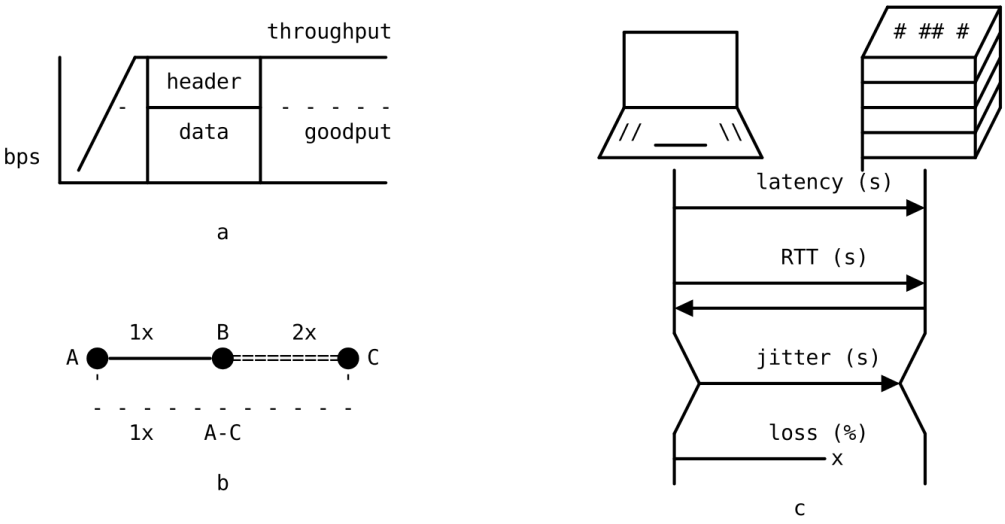


Figure 2.2: These illustrations show the relationships between different network metrics. (a) Goodput is similar to the throughput, but measured without the overhead of lower layers. (b) A connection is limited by the lowest bandwidth between the two networked devices. The bandwidth of the indirect connection between `A` and `C` is limited by `A-B`, even though `B-C` has double the bandwidth. The logical connection `A-C` will have the same bandwidth as `A-B`. (c) Latency is the delay in one way, RTT is delay in both directions, jitter is the variability in latencies, and loss is whenever a packet does not arrive.

Figure 2.2 depicts a number of network metrics that are important to this thesis [KR17]. Foremost, the bandwidth of the network, the maximum amount of data that can be transferred over a connection, expressed in bits per second (bps). The one gigabit per second (Gbps) bandwidth of a Cat5e network cable, or the twenty megabits per second (Mbps) bandwidth of a home internet connection are a few examples. A connection can be either direct or indirect. A direct connection physically connects the sender and receiver, whereas an indirect connection has their packets forwarded by intermediate devices. The lowest bandwidth connection between two devices of the overarching connection limits the bandwidth of a connection. Throughput is measured similarly, but it represents the actual amount of data that an application can transfer over the connection at that time. This value might be lower than the bandwidth for a variety of reasons, including multiple applications communicating over the network, lowering the share of the bandwidth that each application can take ownership of. Furthermore, at higher layers it is difficult to measure the throughput accurately, because lower layers add their own headers to the data, of which the size cannot be measured by the higher layers. The term goodput refers to the useful amount of data transferred over the network without the overhead of lower layers, which is again expressed in bps. This can be measured accurately at the higher layers. Data sent over a network by a communicating party does not arrive instantaneously at the receiver; there is a latency expressed in time units. The latency of a connection is proportional to its physical length. The time it takes for data to be sent to another party and for the response of that party to arrive is referred to as the round trip time (RTT), which is the sum of latencies in both directions. Another critical measure is the amount of round trips required before application

data is transmitted. Any communication without data required to execute the main function of the application; such as handshakes or metadata exchange, will cause the application to start late.

The connection, however, may have fluctuating measurements over time. Congestion is a major cause of this, which occurs when more data is sent than the network can handle. Middleboxes may be forced to drop data, resulting in packets not being forwarded. The packet loss metric represents the percentage of packets lost during transmission. Lost packets that are retransmitted will have a higher latency compared to packets that get delivered on the first attempt. The fluctuation in latency is referred to as jitter.

Bufferbloat is another issue that causes measurement irregularities. Many network devices have large buffers in an attempt to reduce packet loss. The device can receive packets, temporarily store them in the buffer, and then forward them while maintaining maximum throughput. However, if the network becomes congested, the buffer will approach its maximum capacity, and the device will be unable to forward packets as quickly as they arrive. Because no loss is detected, the applications sending data are not notified of the congestion in a timely manner. This can result in increased latencies and jitter [The22a; Get11].

2.2 Transport Protocols

Logical communication is provided by the transport layer. Two applications can communicate as if they were directly connected via the transport layer. If there is no direct connection, the lower layers handle forwarding packets to the correct host [KR17]. There are two traditional transport protocols relevant to this thesis: User Datagram Protocol (UDP) and Transmission Control Protocol (TCP). A connectionless and unreliable protocol and a connection-oriented and reliable protocol, respectively. The next-generation protocol QUIC will be discussed afterwards.

The three-page specification for UDP [Pos80] describes a very bare-bones protocol. A UDP datagram, as shown in Figure 2.3, only provides the source and destination port numbers, the payload length, and the payload checksum. The port numbers are required for a host to multiplex connections. An application binds to a socket created with a port number. When a packet arrives at the host, it is routed to the right socket, ensuring that it reaches the intended application. As a result, a host can have multiple connections open at the same time. The other fields show the payload length, the cumulative amount of bytes of the header and the payload, and a checksum, to ensure data integrity. Application data can be found after 64 bits of overhead. Furthermore, UDP is connectionless and unreliable. There is no handshake between the two communicating entities, and no explicit connection is established. Every packet sent will be delivered with in a best-effort manner [KR17]. Because application data can be sent immediately, the minimal overhead results in good performance, as shown in Figure 2.4. Packets, on the other hand, can be lost and will never be retransmitted. The purpose of this protocol is to allow applications to send messages while using as few protocol mechanisms as possible.

A few years before UDP, the first in-depth specification of the Transmission Control Protocol (TCP) [CYC74] was released. TCP should be used whenever an application requires ordered and reliable data stream delivery, according to the UDP specification. The TCP datagram is depicted in Figure 2.5. The first 32 bits represent the source and destination port numbers. The sequence and acknowledgement numbers ensure reliability. The sequence number expresses the offset of the first byte transported during this TCP session; for example, if 5000 bytes must be transported and the network can only handle 500 bytes at once (excluding the TCP header), 100 TCP packets will be created, with sequence numbers: 0, 500, 1000, ... , 4500. The next expected byte is represented in the acknowledgement number; for example, if the first packet with 500 bytes arrives, the acknowledgement number will be 501. The offset specifies where the

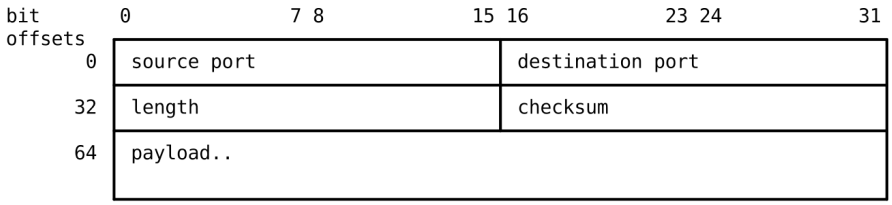


Figure 2.3: UDP datagram.

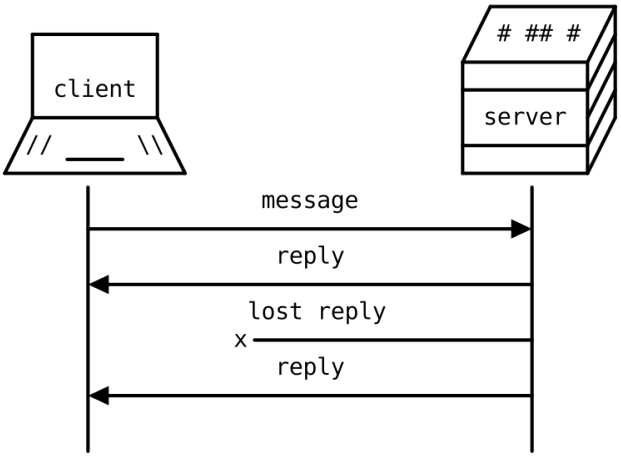


Figure 2.4: A diagram showing UDP communication between a client and server. Without any prior communication, the client sends a message to the server, which sends a response back.

header ends and the payload begins, and it is determined by the number of options enabled. TCP, being a more complicated protocol, utilizes a number of flags to represent the purpose of a packet. A receiver, for example, can distinguish between handshake, data, acknowledgement, and termination packets in this manner. There is also a window size field, which is required for congestion control and can be used to calculate the maximum amount of data that can be sent at once. A checksum is also included in this datagram to ensure its integrity. Finally, there is an urgent pointer which can be used to notify the receiver of important data. A number of options can be optionally added to the header. These options can alter the function of the TCP implementations; for example, by enabling selective acknowledgements or allowing for larger window sizes [KR17]. There is at least 160 bits of overhead, but this could increase to 512 bits depending on the number of options available.

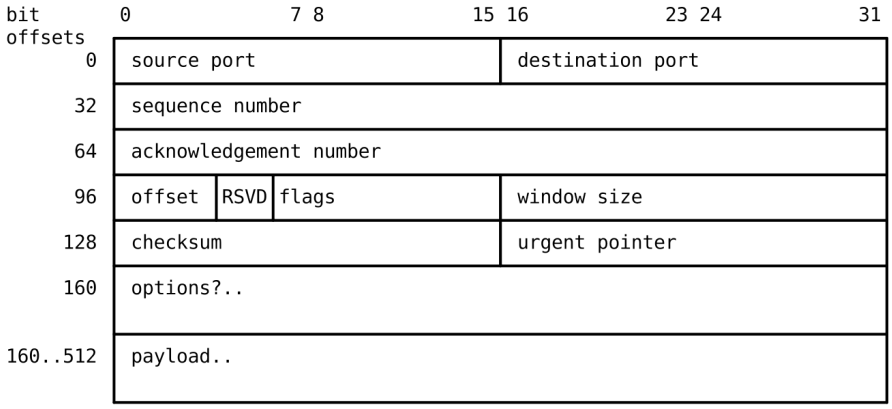


Figure 2.5: TCP datagram.

TCP communication is depicted in Figure 2.6. The client initiates a connection with the server by sending a SYN packet, which is acknowledged by the server. The connection is established in a single round trip, and data can then be communicated. To finish the session, FIN packets must be sent. Congestion control manages the connection during the TCP session, attempting to avoid a state where the network is congested. There are numerous congestion control algorithms available to accomplish this. By sending increasing amounts of data, the network will be probed to estimate its bandwidth. If congestion is detected, the congestion controller will decrease the amount of data sent and resume probing. This cat-and-mouse game will continue throughout the session in order to maximize bandwidth usage.

Transport Layer Security (TLS) [Res18] can be used on top of TCP to provide secure transmission. By encrypting the communication, TLS provides a secure channel that abides by the CIA triad. Only the client and receiver can read the encrypted transmissions, ensuring confidentiality. Transmissions cannot be altered by a third party, guaranteeing integrity. Authentication ensures that communications are carried out with the correct entity by authenticating the server and, optionally, the client [KR17]. Figure 2.7 demonstrates the establishment of a connection when TLS is used on top of TCP. Following the TCP handshake, a TLS handshake is performed, which requires two round trips. The secure connection requires three round trips to be established, before application data can be transferred.

QUIC (not an acronym) is a transport protocol, initially devised by Google, but later adopted by the IETF. This new protocol runs on top of UDP, which is already supported by the vast majority of devices. Figure 2.8 shows that QUIC does not conform to the standard layered representation; however, because most devices already support UDP, this simplifies adaptation. According to Cloudflare Radar [Clo22a], a website that shows daily Internet statistics, around

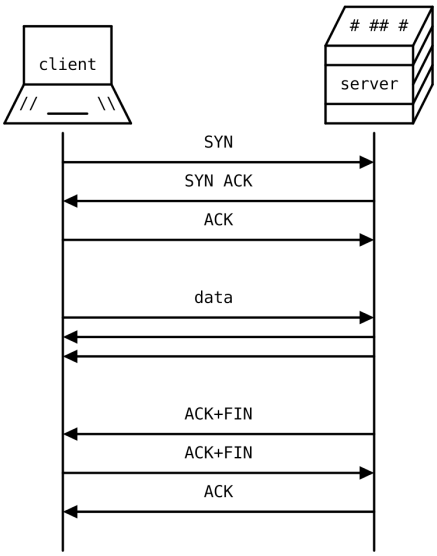


Figure 2.6: A diagram detailing TCP communication between a client and server. Both parties have to establish a connection by performing a handshake, after which data can be exchanged. The connection is terminated after before ending the communication.

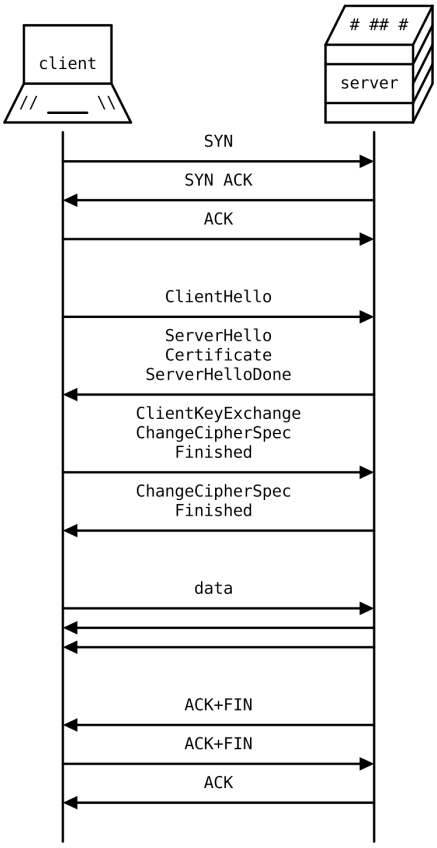


Figure 2.7: A diagram detailing TLS communication over TCP between a client and server. Both parties have to establish a connection by performing a handshake, after which the TLS handshake can be carried out. Both parties send encrypted messages that can only be decrypted by the receiver. The connection is terminated after before ending the communication.

30% of traffic is already using QUIC. The sheer amount of QUIC traffic is primarily the result of some large companies, such as Google, Facebook, and Cloudflare, pushing this new protocol. Contrary to popular belief, using UDP does not automatically make QUIC a faster protocol; it does not improve performance. QUIC implements the features that make TCP slower than UDP; for example, both protocols use similar congestion control, which manages the amount of data that can be sent over the wire, and provide reliable communication, which necessitates acknowledgements and retransmissions [Mar21b]. QUIC can be faster, but only because it is smarter [Bom22].

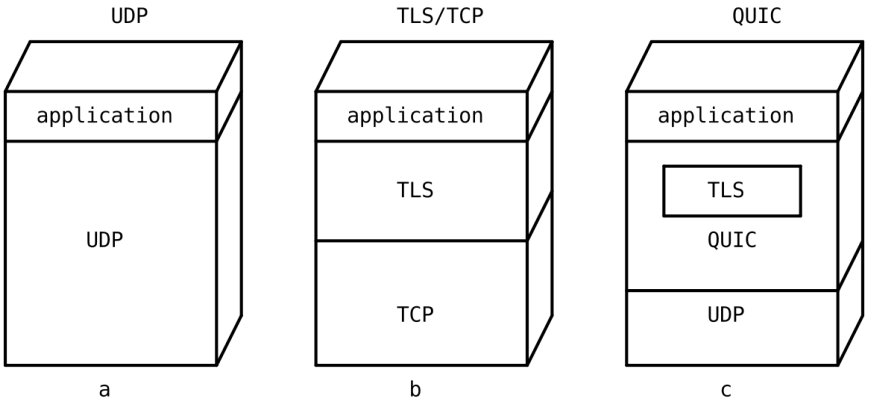


Figure 2.8: A comparison between an application using UDP (a), TLS/TCP (b), and QUIC (c). QUIC has overlap with TCP because it implements similar features on top of UDP, and QUIC integrates TLS.

QUIC is an alternative for TCP. It is adaptable and can be used with any application protocol that runs on top of it, and it exists to address some flaws in TCP. The first flaw is the layered model, while it is very modular and allows for protocols that do not know about how another protocol functions, it is not very efficient. For example, running TLS on top of TCP requires three round trips for a connection to be established. The ability to evolve over time is a second flaw. In practice, it has been proved that adding new features to TCP takes a long time, if at all. TCP Fast Open, which should remove a round trip from successive connection establishments with the same server, is a feature that improves TCP; however, firewalls flag these packets as malformed [Mar21a]. QUIC addresses the first flaw by integrating TLS. This enables QUIC to establish the connection and perform the TLS handshake at the same time, reducing the number of round trips required, as shown in Figure 2.9. This is related to the performance enhancement that will have the greatest impact on the majority of users, 0-RTT. The ability to reconnect to a server using previously established connections rather than performing a new handshake. While TLS1.3 supported this, it still required the establishment of a new TCP connection, requiring a single round trip. When reconnecting with 0-RTT, QUIC allows application data to be transferred in the first packet [Bom22; Mar21b].

The second flaw is avoided by encrypting the packet headers, which is also made possible by TLS integration, making adoption easier. Middleboxes can no longer scan the packets and because there is no plaintext visible on the wire, packets cannot be flagged as malformed when a new extension is used; all packets appear the same. There is already an unreliable datagram extension that allows data to be sent unreliably over QUIC while looking exactly like any other QUIC packet on the wire [PKS22]. In practice, this can lead to adaption issues as well because firewalls are no longer able to scan traffic. Allowing any connection through the firewall could be argued to be less secure. Middleboxes, on the other hand, are permitted to read the connection ID, which uniquely identifies a QUIC connection. These IDs can be used for routing,

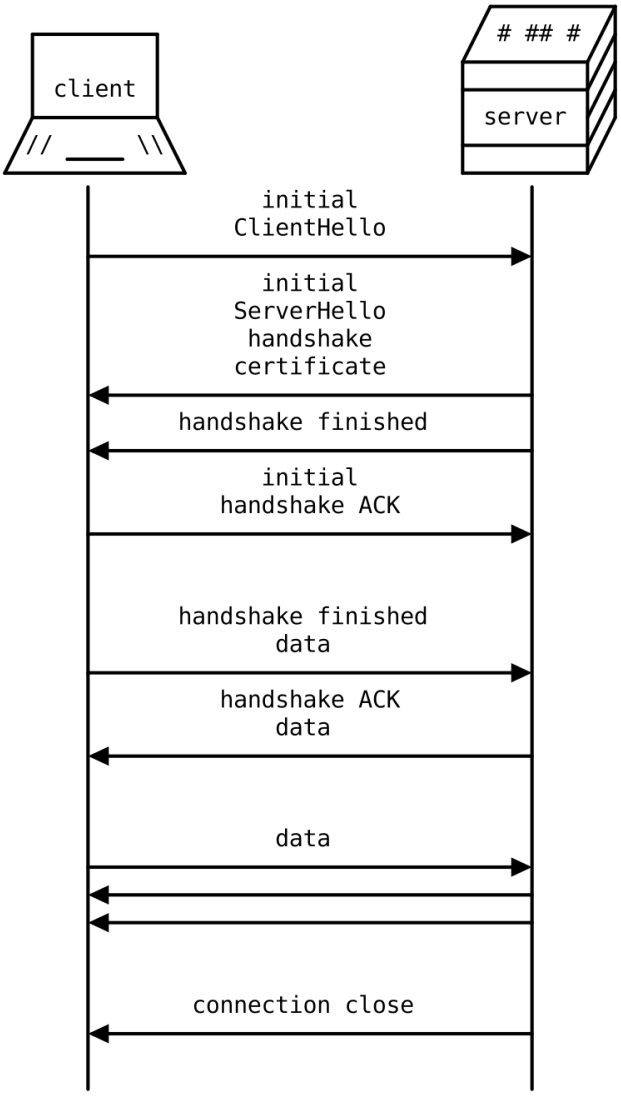


Figure 2.9: A diagram showing QUIC communication between a client and server [Dri22].

ensuring that a connection always reaches its intended destination. Additionally, these IDs are used for connection migration. Every time the network connection changes, TCP requires a new connection that starts with a blank slate, whereas QUIC can continue to use the existing connection by matching the connection ID [Bom22]. Since such IDs can be used to track a connection, multiple connection IDs are agreed upon for linkability prevention. Because they are part of the encrypted payload until utilized, these IDs are unknown to the middleboxes. A unique logical connection ID will be assigned to each physical connection. When the physical connection changes, the middleboxes will notice a new connection ID, which the client and server can match to existing connections. Furthermore, because the client and server use different connection IDs for a single connection, data flow can only be observed in one direction by middleboxes [IT21; Mar21a].

Several types of headers can be seen when inspecting QUIC traffic. Figures 2.10 and 2.11 depict a datagram with a long and a short header. QUIC attempts to minimize overhead by sending only the necessary data and storing parameters rather than transmitting them with each packet. The first packets will always use the long header format, which is used to negotiate between the parameters of the communication session. These parameters can then be used, and a short header will suffice for any subsequent communication. For example, the length of the connection ID is not specified and can be negotiated during the handshake; once the length is determined, only the IDs themselves should be sent [IT21]. Since QUIC runs on top of UDP, it will always have at least the overhead of the UDP header, as well as the basic needs of the QUIC protocol. However, there may be less overhead than with TCP. A long header can be as short as 120 bits or as long as 424 bits. The short header can be as little as 80 bits long, which is only 16 bits longer than a UDP header, and as much as 264 bits long.

Another improvement over TCP is the method of sending acknowledgements. By default, QUIC supports selective acknowledgements, which is an extension for TCP. The sender and receiver can also negotiate when an acknowledgement must be received. There are still limits to how much data can be sent, however the receiver can request that an acknowledgement be sent only once in a while [Mar21b]. This demonstrates yet another way in which QUIC is a very adaptable protocol.

Multiplexed streams is a new feature of QUIC that is not available in TCP. A TCP packet contains any data handed to it by the layer above, whereas a QUIC packet contains frames that can exist independently of other frames. Figure 2.12 illustrates how multiple frames can be contained in a single QUIC datagram. A sender can send a packet containing frames from multiple streams, which the receiver can demultiplex. This has a number of advantages, beginning with retransmissions. If a packet gets lost, only the frames that should be retransmitted must be sent again. In a scenario where streams have different priorities, the highest priority stream frames can be retransmitted earlier than the lower priority frames. Furthermore, if the layers above implement data stream multiplexing, each stream can be mapped to a QUIC stream. With TCP, each packet would contain a part of various streams. Since there is no knowledge of what data is contained, this can result in Head of Line (HoL) blocking. This means that a TCP packet will not be decoded until all previous packets have been received, even if the data contained might not be part of any other data in later packets, blocking all streams at once. With QUIC, every packet can be decoded, and only the streams with missing frames are blocked [Bom22; Mar21a].

QUIC, on the other hand, is still dealing with some issues as a result of being such a new protocol. Large organizations, such as Microsoft, have found that the UDP stack is less optimized than the TCP stack, owing to the fact that TCP was the primary scenario for a long time. This causes QUIC implementations to be bottlenecked by the UDP stack in specific use cases, resulting in skewed results [Ban22]. Furthermore, applications and protocols have optimizations that

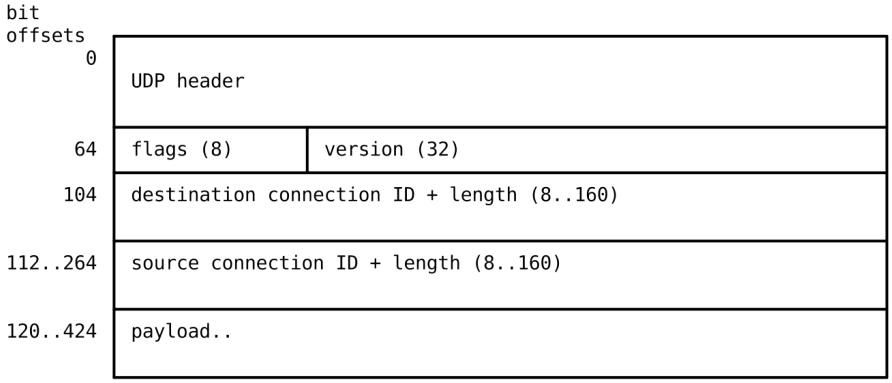


Figure 2.10: QUIC datagram, with a long header.

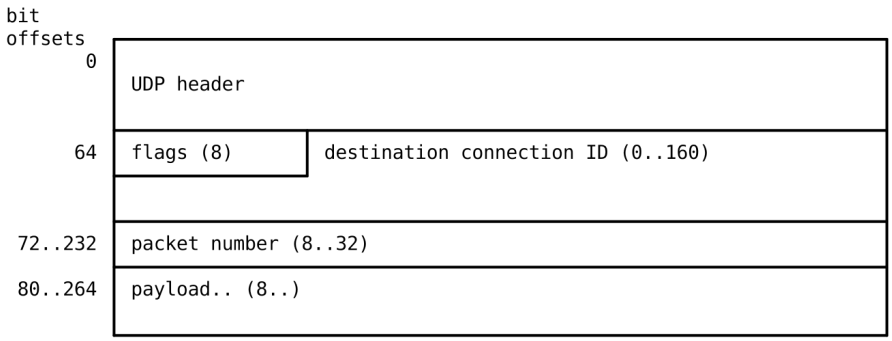


Figure 2.11: QUIC datagram, with a short header.

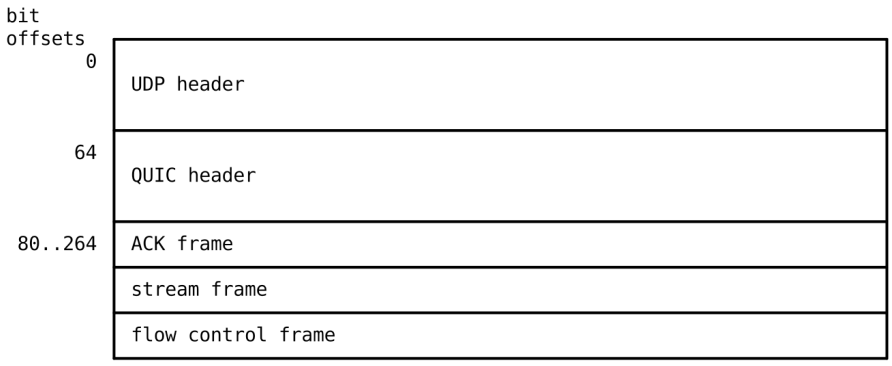


Figure 2.12: QUIC datagram, with payload containing numerous frames.

improve TCP performance; however, these optimizations may not be desirable in the case of QUIC [BRZ17]. QUIC, on the contrary, can outperform TCP in certain use cases, such as seeking through a video stream [AB18].

Another study found that the current QUIC implementations are very heterogeneous. At the time of writing, there are approximately fifteen maintained implementations, each with significantly different implementation choices. Congestion control, for example, can be New Reno, Cubic, or BBRv1 [Mar+20b]. At times, the code and documentation are out of sync, making it even more difficult to figure out what is implemented, such as with quic-go, where congestion control is referred to as Cubic but is implemented as New Reno[1].

Because the protocol is evolvable, extensions that aim to improve performance by employing novel techniques already exist. One of these techniques is to remove the slow start phase from congestion control. This phase can be bypassed by estimating the bandwidth and communicating this estimate to the other networked entity, reducing the time required to maximize bandwidth usage [Rüt+19]. Others have argued that it would be beneficial to deviate further from the layered model and share information between layers. This cross-layer information sharing would enable protocols to respond to a greater number of events and make decisions based on a larger knowledge base, improving performance [Her+20].

TCPLS [Roc+21] combines TCP and TLS into a single protocol to achieve similar benefits to how QUIC can make smarter decisions by integrating TLS. TLS/TCP is compared to TCPLS in Figure 2.13. TCPLS appears on the wire exactly like TCP, but the encrypted payload contains TLS records. These records, like QUIC frames, enable features like multiplexed streams and connection migration. Using multiple TCP connections helps to avoid HoL blocking. TCPLS manages the various connections and divides streams among them. Streams from other connections cannot be blocked in this manner, but they can be blocked by streams from their own connection.

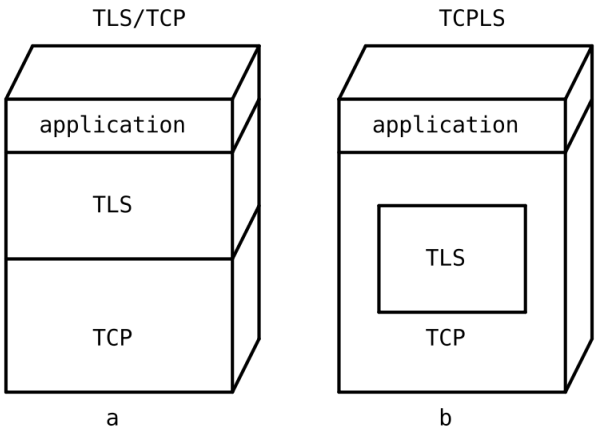


Figure 2.13: A comparison between TLS/TCP (a) and TCPLS (b). Instead of running TLS on top of TCP, TCPLS integrates TLS within the TCP protocol.

2.3 Transfer Protocols

A transfer protocol is an application layer protocol, which directly speaks to the overarching application. As the name implies, its primary purpose is to add functionality that allows two

[1]There is an issue on their GitHub repository with a person confused about what congestion control is available to them: issue #3189

networked applications to exchange data. The thesis will solely focus on the protocols that are relevant to the streaming protocols discussed in Section 3.4, namely the HyperText Transfer Protocol (HTTP).

HTTP was part of the initial implementation of the World Wide Web by Tim Berners-Lee. The single line protocol, later dubbed version 0.9, that only allowed the GET method, evolved into an extensible protocol that is used for a myriad of applications [Mc21]. Version 1.0 mostly added metadata to the protocol, enabling client and server to share their status and information about the content [NFB96]. Version 1.1 was soon after standardized. This version not only expanded on the previously defined metadata, but also made some protocol improvements [Fie+97]. Notably, persistent connections improve performance on documents with many embedded resources because TCP congestion control has already surpassed the initial slow start. This version also includes range requests, which allow streaming clients to request only a portion of a segment. Another new feature is Chunked Transfer Coding (CTC), which allows a sender to stream content in chunks rather than sending it all at once. This is an important feature for a number of the streaming protocols discussed in Section 3.4, as it reduces latency. Furthermore, version 1.0 states that HTTP should not be used for sensitive data transmission. HTTP suffers from lower layer overhead, which means that every first message will have the three round trip overhead of the TCP handshake, later messages can overcome this using persistent connections.

The Hypertext Transfer Protocol Secure (HTTPS) protocol is used to secure HTTP communications. It uses TLS over TCP instead of a TCP connection to implement the CIA triad. This implies that, while an HTTPS connection is secure, it also suffers from the additional overhead introduced by TLS over TCP.

HTTP/2 [BPT15] introduced the next generation of transfer protocols. Instead of using human-readable plain text to represent requests, this version employs binary encoded frames. These frames form messages, which are then transmitted via a stream. Another improvement is the ability to contain multiple streams within a single TCP connection. This can be used by a server to perform a server push. This is a feature that allows the server to send data to the client without the client requesting it. When the server is certain that a resource will be requested in the future, it can initiate the response in a new stream. In the form of HPACK [PR15], another performance feature added to HTTP/2 is header compression. The compression is accomplished by keeping a dictionary of header names and values. Every header has the following format, which is composed of two parts: “name:value”. The full header can be stored in the dictionary for maximum compression, but only if the header will not change. If the header is subject to change, frequent names or values can be stored separately and referred to instead [Kra16].

All HTTP RFCs were updated in June 2022, and the HTTP/3 RFC was added. The HTTP semantics and caching are generally described in their own documents [FNR22b; FNR22a]. A separate document describes the version-specific features and behavior for HTTP/1.1, HTTP/2, and HTTP/3 [FNR22c; TB22; Bis22].

HTTP/3 [Bis22] could be described as HTTP/2 running over QUIC. The specification describes all HTTP/2 features, including those subsumed by QUIC. The initial goal was to run HTTP/2 over QUIC, but this caused numerous issues, such as having streams and prioritization in both HTTP/2 and QUIC. The solution was to develop HTTP/3, which is mandated to run on top of QUIC and has features shared by both protocols. HTTP/2 and HTTP/3, for example, both use multiplexed streams; however, HTTP/2 streams are subject to TCP HoL blocking, as described in Section 2.2, whereas HTTP/3 uses QUIC streams, which do not suffer from this problem. The HTTP/2 and QUIC streams are very similar, but the layer at which they operate is what distinguishes them. The primary application for QUIC is HTTP/3 [Bom22]. HTTP/3 includes header compression, which is now known as QPACK. It is a variation of HPACK, but reduces HoL blocking [KBF22].

Figure 2.14 compares HTTP/1.1, HTTP/2, and HTTP/3. The main distinction between these protocols is that HTTP/3 employs QUIC rather than TCP; the comparison between these two is already made in Section 2.2. Furthermore, HTTP/2 introduced some features that were later subsumed by QUIC, resulting in incompatibilities between HTTP/2 and QUIC.

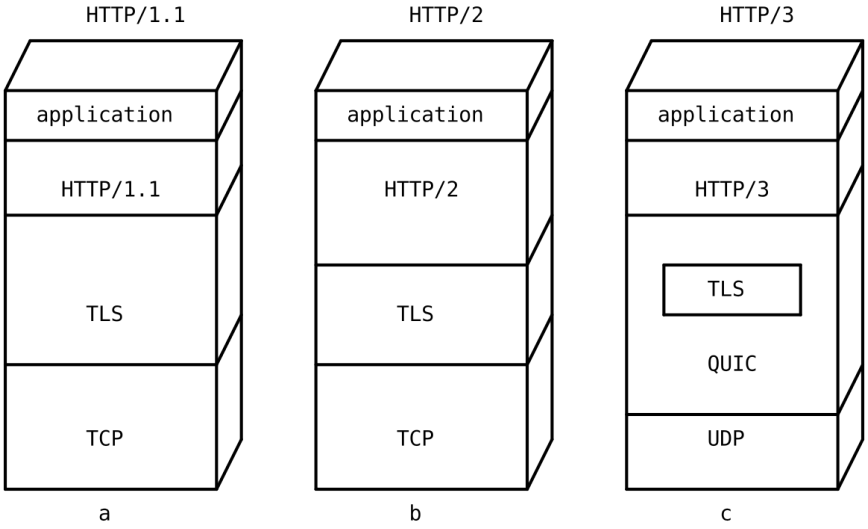


Figure 2.14: A figure comparing HTTP/1.1 (a), HTTP/2 (b), and HTTP/3 (c). HTTP/1.1 and HTTP/2 both conform to the layered model, the features of the protocols in each stack are similar to their corresponding protocol, except for the stream multiplexing added to HTTP/2. HTTP/3 is quite different, since QUIC implements TCP features on top of UDP, integrates TLS, and provides some functionality previously in HTTP/2.

Figure 2.15 compares data stream multiplexing with various HTTP versions. With HTTP/1.1, there is no multiplexing within a single connection, but there is no HoL blocking; connections are handled independently. HTTP/2 supports stream multiplexing, but it suffers from TCP HoL blocking. While streams can be combined in a single connection, the connection must be handled completely in order to decode the individual streams. Since HTTP/3 uses QUIC stream multiplexing, which is free of HoL blocking, streams can be handled independently, just as they would with HTTP/1.1, but with the added benefit of reducing overhead by using a single connection.

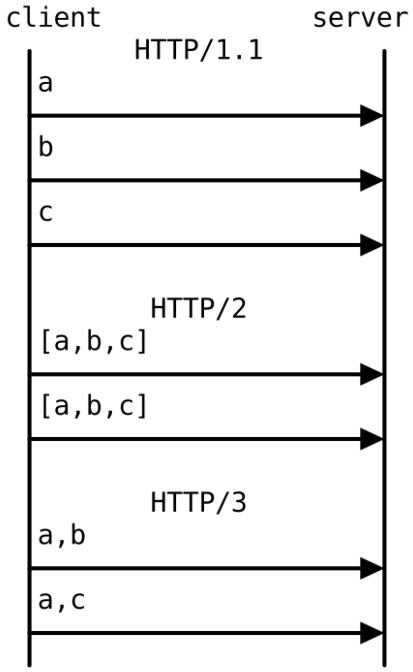


Figure 2.15: A figure comparing multiplexing streams in HTTP/1.1, HTTP/2, and HTTP/3. The client sends data using three streams: a, b, and c. With HTTP/1.1, the protocol uses a single connection for every stream, there is no multiplexing, however, the protocol knows which packet contains which stream. Multiplexing is available in HTTP/2, but it does suffer from HoL blocking caused by TCP. The protocol knows which streams exist, but not exactly in which packets the streams are contained. HTTP/3 uses QUIC streams for multiplexing, that do not suffer from HoL blocking, the protocol is aware of which stream is available in which packet.

Chapter 3

Streaming

Streaming is the continuous transfer of data that allows immediate processing or playback [Dic21b]. A client uses streaming to access content in this manner, while a server enables streaming by providing streaming-friendly content. This chapter gives an overview of streaming and how streaming protocols function. It emphasizes some older techniques while clarifying modern protocols pertinent to the thesis.

Content is a very broad term in the context of streaming. It can refer to anything from a simple list of numbers to a more complex type of media, such as video. In other words, meaningful data that a client can access. A piece of content should be streamable if it can be used when it is only partially available. Unless otherwise stated, streaming refers to video streaming and content refers to video. Figure 3.1 depicts video as a dynamic type of media, consisting of a sequence of images or frames. It may or may not include audio or subtitles. A client can play back the content from any point in the frame sequence. Because any subsequence of frames can be played back[1], this implies that video is a streamable type of media.

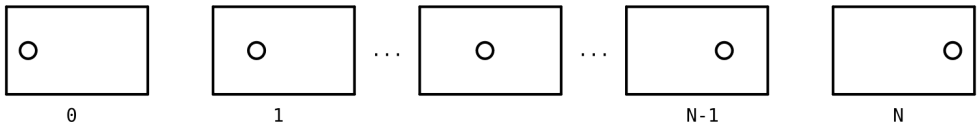


Figure 3.1: Representation of a video, consisting of `N` frames, where an object moves from left to right. The movement of the object is captured by the frames, starting from frame zero, and can be observed by playing back the frames in chronological order.

During streaming, two entities interact with each other: the client and the server. A client, also known as a consumer, is an application that wants to access, retrieve, and play back content. The content is hosted by the server, also known as the provider, and is made available to the client. There is typically a many-to-few relationship, with many clients accessing content on a limited amount of servers.

Furthermore, there are two types of video streaming: video-on-demand (VOD) [Dic21c] and livestream [Dic21a]. VOD allows a client to watch the content at any time, and the entire video is available at the provider. By allowing fast-forward and rewinding, VOD gives the client

[1] In order to achieve compression there are different types of frames, some are dependent on other frames and do not contain all data necessary for decoding. This means it is not always possible to play back a video from any frame without having the other frames which it depends on. Section 3.1.1 states how this problem is avoided while video streaming.

more control over how the content is consumed. A livestream, on the contrary, is either a live broadcast or prerecorded content presented as if it were live. The client can only access the video up to the current time, limiting how the client can consume the content. It is not possible to fast-forward past this point, but everything prior is still available. After the livestream has ended, and the content remains available, it can be considered as VOD content. This thesis will focus on VOD content, but the insights can be applied to live content as well.

When streaming, the Quality of Experience (QoE) uses multiple metrics to assess the quality of the stream as perceived by the consumer, both by the application and by humans [HSA17]. Metrics can be objective or subjective. The objective metrics represent the quality of the content. This can be accomplished by employing an absolute value, such as bit-rate, which expresses the number of bits encoded per second of content. A higher bit-rate indicates higher quality content when both pieces of content are encoded using the same codec, which is elaborated in Section 3.1.1. Another way to represent objective quality is to use relative values that compare one piece of content to another. The Peak Signal-to-Noise Ratio (PSNR) and the Structural Similarity Index Measure (SSIM) are two examples of relative values [Set21], where content is compared to the highest quality version of that content. ITU-T Recommendation P.1203 is another method of scoring the quality of content [Int17]. The P.1203 standard defines different modes that can be used to evaluate the quality. One mode is entirely based on metadata, specifically, bit-rate, frame-rate, and resolution. Another mode uses metadata and header data, adding frame types and sizes. The last two modes uses all of the above and different amount of content data. Subjective metrics assess how a human will perceive the content. This can be accomplished by conducting a survey, interviewing test subjects about their experiences, or measuring events that are known to impact the experience. Buffering events, for example, affect the smoothness of playback, with smoother playback indicating a higher QoE. Many factors influence these metrics, ranging from the client device to the human itself. The network-related metrics, detailed in Chapter 2, that influence QoE, are, however, the most important to this thesis. Buffering occurs when the time it takes for a segment to arrive exceeds the playback time of that segment, forcing the client to wait for a new segment to arrive with nothing to show. This means that in order to have a smooth streaming experience with no buffering, the throughput should be at least equal to the bit-rate of the content. The client can try to avoid buffering by keeping a buffer with future segments to show, ensuring that there is something to show if the available throughput suddenly drops. However, the buffer is only a short-term solution that can run out. Latency has an impact on QoE, especially when it fluctuates due to jitter. This causes an unstable throughput, making throughput-based decision-making difficult. Packet loss within the network is detrimental to the QoE. After a loss event, the client must notice the loss, and then the packets must be retransmitted. This adds a significant amount of latency for the lost packet to arrive, increasing the chance of buffering.

3.1 Content

The content must be repackaged to be suitable for streaming. Figure 3.2 depicts the source content and the repackaged content, the latter encoded for streaming. The example is most relevant to the streaming techniques discussed in Section 3.4, but it will provide general insights into how this process works. In this example, the original video, audio, and subtitles are separated into their own layer. In practice, these layers can be merged; for example, the video can include audio or subtitles.

3.1.1 Encoding

Encoding is converting data into another format that represents the same data [Tec21], a codec executes this task. This can be both a lossless or a lossy operation. Lossless encoding ensures that

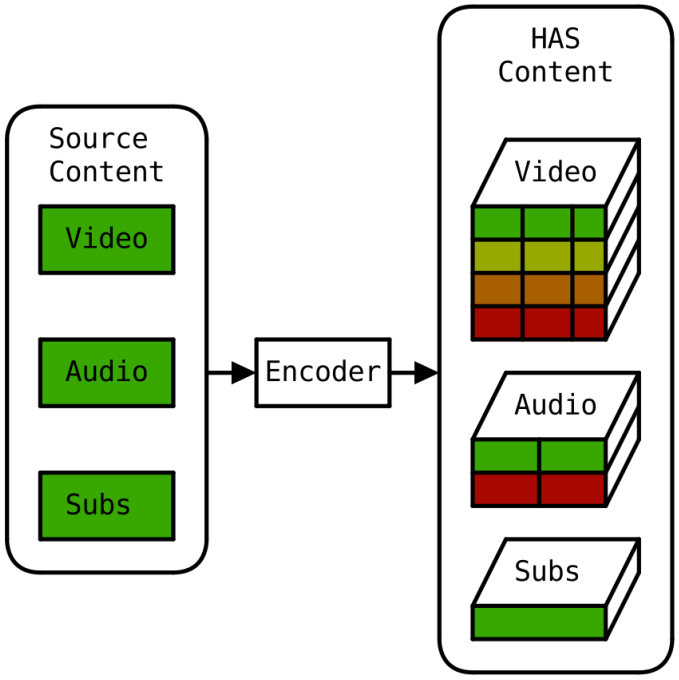


Figure 3.2: Encoding process of source content into multiple adaptations and segment sizes for each layer of the content, in order to facilitate HTTP Adaptive Streaming (HAS), detailed in Section 3.4.

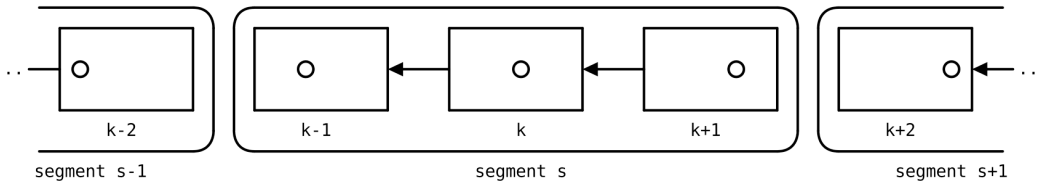


Figure 3.3: A diagram showing a subsequence of frames from a video. The arrows indicate a dependency between frames; frame `k` is dependent on frame `k-1`, meaning both frames are required to decode frame `k`. There is no dependency between frames `k-1` and `k-2`. Segment `s` contains frame `k`, all its dependencies, and its dependents. There are no dependencies between frames belonging to different segments, resulting in the independence of segments.

both formats represent identical data, the content is not changed. With video, lossy encoding is more common, the data of the content is changed in order to achieve compression. A common approach is exploiting temporal redundancies, consecutive frames contain similar regions. A region is only stored once, others refer to this and store a difference with the reference. However, this can introduce unwanted artifacts, especially with high compression, and can lead to a lower QoE. A codec can be tuned using parameters, which change the rate of compression. The technical details of the encoding process are out of scope for this thesis.

An encoder takes the source content and performs two major tasks relating to video streaming. The first task is to create multiple adaptations of the source, each with a different bit-rate [Bit20]. In the example depicted in Figure 3.2, the color of an adaptation represents the bit-rate, with green depicting the highest bit-rate and red depicting the lowest bit-rate. In this context, bit-rate is directly proportional to quality, which means that there will be high and low quality versions of the source material. As shown in the example, not every part of the content must have the same amount of adaptations. Video, which typically has a very high bit-rate, has the most adaptations. The high bit-rate makes smooth streaming challenging in a network with insufficient throughput. More adaptations mean there are more options when it comes to selecting the best adaptation for a given scenario, maximizing QoE. The dynamic range of the corresponding organ is another factor that influences the number of adaptations. The human ear has a much larger dynamic range than the eye. Due to this discrepancy, audio is more susceptible to compression artifacts [FJT02]. It is preferable for audio to make fewer adaptations at bit-rates that introduce fewer artifacts. Because audio has a lower bit-rate than video, it does not present as many network bandwidth challenges as video does, which is why fewer high-quality adaptations are preferred. Finally, the subtitles have only one adaptation, the text has only one version. Multiple adaptations, such as translations into different languages or subtitles for the deaf and hard of hearing, would be possible. Adaptations can also represent alternative content, such as censored video or audio in different spoken language.

The second task is to make segments [Bit20]. A segment is a temporal piece of content that contains all data from a specified start time to a specified end time. It is the smallest unit that will be made available for streaming[2]. In practice, a segment can represent a single frame or a sequence of frames lasting milliseconds to seconds. The encoder divides a layer into segments that do not overlap. Figure 3.3 depicts a segment and the frames contained within. The segment contains all the data required for it to exist independently, which means it can be decoded without requiring the retrieval of any other segments. Each layer can have a different segment length, a different time interval between the start and end time. In this example, the video has the smallest segments, which keeps the total binary size of a segment relatively low even at high bit-rates. Because the average bit-rate of an adaptation is lower, the audio has longer segments. Furthermore, the subtitles are only available as a single segment that contains all the data of the content. In practice, the segment size of each adaptation within a single layer can vary.

3.1.2 Manifest and Container

A manifest includes all the metadata needed to stream a piece of content [14]. It describes all adaptations, their segments, their encoding, and their location in the network. The manifest is required by the client in order to bootstrap the stream. A manifest may include application-specific metadata in addition to the content description. For example, Netflix provides a list of cloud providers as well as a ranking, allowing the client to choose which provider to use [Adh+12]. A manifest for VOD describes all segments that comprise the content. However, for livestreaming, a manifest only describes the segments that are available when the manifest is

[2]While a segment is the smallest unit for representing a portion of the content, certain streaming techniques can request partial segments. This is explained in Section 3.4

retrieved, because future segments may not yet exist[3]. This means that the client must update its manifest on a regular basis while livestreaming in order to keep track of all newly available segments.

A container is not the same as a manifest. A container contains both the metadata and the content data, whereas a manifest only contains the metadata [Mat20]. As shown in Figure 3.4, the content for streaming is wrapped in containers; the adaptations in a manifest refer to different containers with differently encoded content. Not all containers are created equal, as they support different codecs and have different limitations. As a result, different containers are supported by streaming protocols, which means that a provider may have to host content represented by multiple types of containers if they want to support multiple streaming protocols.

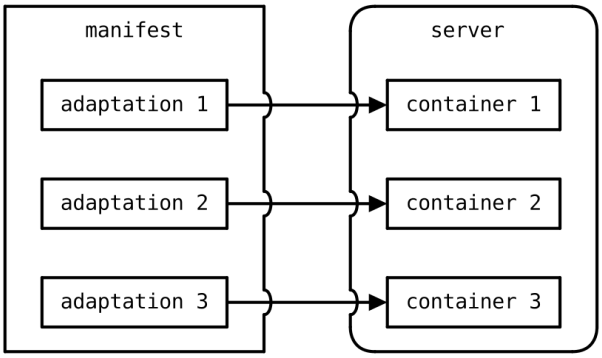


Figure 3.4: The relationship between an adaptation and a container. An adaptation is listed in the manifest and refers to a container on the server to retrieve the content.

Apple and Microsoft proposed the Common Media Application Format (CMAF) [20] in an attempt to create unity among streaming protocols. This format standardizes the segmented content and can be used in any type of media application or manifest. Any streaming protocol can use this single container format [Bit22]. This new standard is being adopted by video developers and is gradually gaining popularity over existing standards [Bit21]. This demonstrates the need for a less fragmented and more coherent technology space.

The CMAF hierarchical model is depicted by Listing 1. CMAF defines a logical model, which can be converted into a manifest, and CMAF Addressable Media objects, that contain the content [App22]. The logical model consists of a *Presentation* at the highest level, which contains synchronized *Selection Sets*. Every *Selection Set* describes the same temporal piece of the same layer of content, and allows for a client to select between different *Switching Sets*, which describe the same content and can be used interchangeably. A client switches between *Tracks*, in which the CMAF Addressable Media objects are described. Such an object can be one of a few types. A *Header* has initialization data, not actual content. Smaller parts of the content are defined by *Chunks*, while larger parts are defined by *Segments*.

[3]Certain streaming protocols do mention future segments in order to improve the performance of the protocol; for example, Low Latency HLS, detailed in Section 3.4.2.

```
<Presentation>
    <SelectionSet>
        <SwitchingSet>
            <Track>
                <Header/> <Chunk/> <Chunk/> <Chunk/> <Chunk/> <Chunk/> <Chunk/>
            </Track>
            <Track>
                <Header/> <Segment/> <Segment/>
            </Track>
        </SwitchingSet>
        <SwitchingSet>
            ...
        </SwitchingSet>
    </SelectionSet>
    <SelectionSet>
        ...
    </SelectionSet>
</Presentation>
<Presentation>
    ...
</Presentation>
```

Listing 1: Hierarchical representation of the CMAF model.

3.2 Streaming Architecture

A streaming client performs a variety of important functions, which can be divided into four components, as shown in Figure 3.5. This is based on the MPEG-DASH specification, as detailed in Section 3.4.1. This architecture, like Section 3.1, is most relevant to the streaming techniques discussed in Section 3.4, but it will provide general insights into how a streaming client works.

The client application interacts directly with the *control* component. This component determines which resources must be requested from the server. A decision is made based on the state of the client, such as the playhead position, whether the client is paused and the playback speed. The most basic decision-making technique is based on the playhead, as described in Section 3.3, or a more calculated decision can be made based on specific algorithms, as described in Section 3.4. The specific inner workings of these algorithms are unimportant to the thesis; all that matters is that they make decisions based on internal parameters and metrics. These parameters and metrics are derived from a variety of sources. Client device capabilities and user preferences are examples of parameters that can be read from the client. Other metrics can be gathered by other components that perform measurements. The control component computes throughput based on RTT measurements taken by the *access* component. The latter performs the retrieval of external resources in the order determined by the *control* component. With network access, it is possible to collect network metrics. The retrieved resource is routed to the *segment* component, which decodes it and passes it on to the final component, the *media* component. This component manages the media engine, displaying the appropriate segment at the appropriate time and synchronizing segments of various media types.

The majority of streaming techniques operate on a simple network architecture. It is based on the client-server model, in which the server contains all the files required to stream the content,

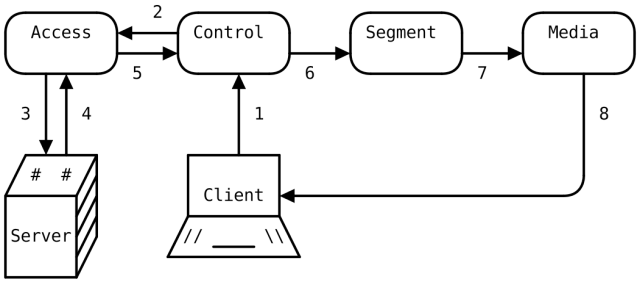


Figure 3.5: Interaction between client, server, and HAS components.

and the client can request these files, or parts of them, at any time. In practice, however, the networks are far more complex. It is unrealistic to connect to a server directly; there will be at least a few hops in the network, such as the Internet Service Provider (ISP). Streaming platforms use Content Delivery Networks (CDNs) to improve the distribution of their content, as shown in Figure 3.6. Instead of a single server delivering content to each client, content is delivered to the CDN. A CDN is a distributed network of servers, dispersed all throughout a geographical location. The servers are placed as close to the edge of the network as possible, usually connecting to the ISP. Content is distributed across the CDN to every edge server. The client accesses the content at the closest edge server rather than the source server, reducing network hops. Distance, in this case, is measured by the latency to a server, since the lowest latency gives the best performance [Clo22c].

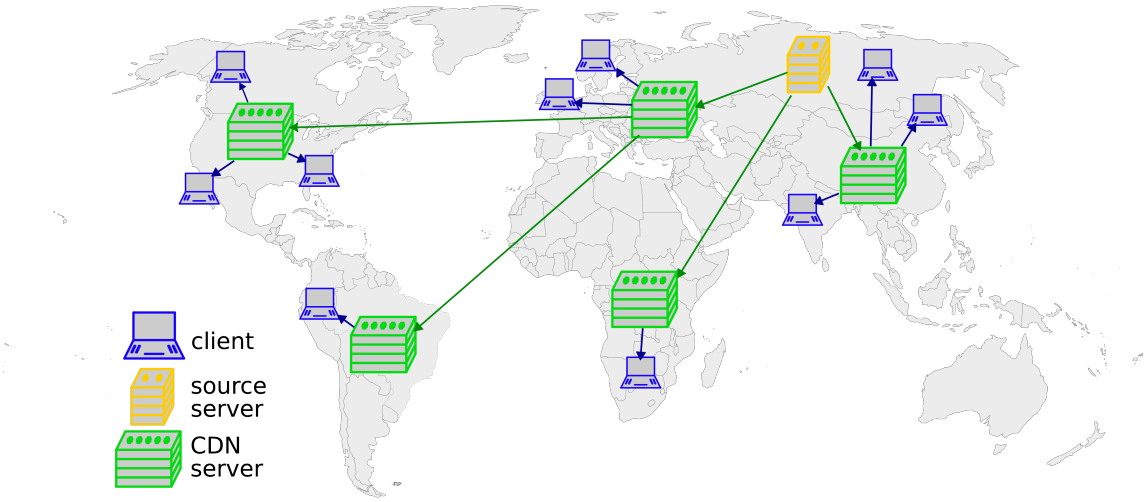


Figure 3.6: The path for content to reach a client when using a CDN. The source server generates content, passes it onto the CDN, which distributes it to every CDN edge server, where clients can access the content at the edge. This figure is based on resourced by the Cloudflare Learning Center [Clo22c].

3.3 Streaming Techniques

Progressive streaming is an early and simple streaming technique that streams a file sequentially. The file contains only one adaptation of the content. A server may offer multiple adaptations, each at a different bit-rate; however, the client explicitly selects only one adaptation to stream [Bit19]. One disadvantage of this technique is that the average bit-rate is constant, but network conditions can change. To achieve the best QoE, the client can decide which adaptation best suits the current network conditions, such as selecting the adaptation with the bit-rate closest

to the throughput of the network without exceeding it. This method achieves the best possible QoE as long as network conditions do not change. However, as conditions change, the QoE will deteriorate. Because the bit-rate is too high, the throughput can be reduced, and buffering occurs. Because of interference from other network traffic during a throughput measurement, the throughput may be underestimated. As a result, the client selects a lower bit-rate adaptation while another adaptation provides a higher QoE. However, the client is not able to change its decision.

The Real Time Streaming Protocol (RTSP) [RLS98] was one of the first streaming-focused protocols, and most commonly uses the Real-time Transport Protocol (RTP) [Fre+96] to transfer data. It supports VOD, including seeking, as well as livestreams. It does, however, suffer from the limitation of only having a single bit-rate, but it does provide a solution. Instead of encoding all content at a low bit-rate, a mixer can be placed in areas that cannot handle high bit-rate content. A mixer receives an RTP stream, re-encodes it at various bit-rates, and then forwards it. Clients can connect to the mixer to receive a lower bit-rate option, or directly to the source to receive the highest bit-rate available. The quality of the service metadata communication is handled by the RTP Control Protocol (RTCP) [Fre+96]. While RTP can work with any suitable underlying transport protocol that provides multiplexing and checksums, applications typically employ UDP, as described in Section 2.2. Since this is an unreliable protocol, there is no retransmission, so any lost packets remain lost, resulting in missing data. The most recent RFC relating to RTP was released in 2016, twenty years after its first RFC, with the goal of transferring modern codec video streams over this protocol.

Real-Time Messaging Protocol (RTMP) is a TCP-based streaming protocol [dac22b]. It was initially created as a real-time protocol for Adobe Flash Player, but it has since found use in livestreaming applications. There are several RTMP variants, each with a unique set of features. There is a version that supports SSL, which encrypts the stream and improves security. Another variant encapsulated the stream within HTTP to bypass firewalls. Real-time communication is possible with a UDP version. It is even possible to switch between adaptations while streaming. Twitch [Int22], a well-known streaming platform, used RTMP for distribution before switching to HTTP Adaptive Streaming (HAS).

HAS is adaptive streaming, as defined in Section 3.4, that transfers data using HTTP, as defined in Section 2.3. Because of caching, HTTP is a more cost-effective method of streaming to large audiences. CDNs can be used to distribute content to major geographical areas. One disadvantage of using HTTP semantics is that latency increases [Kal+17]. Instead of establishing a connection and sequentially streaming the segments, the segments are identified by a unique Uniform Resource Identifier (URI). Every segment is explicitly requested by the client via its URI. From this point forward, the thesis will focus on HTTP streaming.

Interaction between the broadcaster and the audience is common in livestreaming applications. Messages are sent in both directions and are influenced by latencies introduced by both the network and protocols used. The broadcaster experiences the least latency because the audience interacts via a separate real-time channel, such as a text chat. The latency introduced by the video stream, which is dependent on the protocol used, is noticed by the audience. Protocols that scale best introduce the most latency, resulting in increased latency if the application supports a large audience. The latency difference between the two parties causes issues during this interaction [Kal+17]. Because one-on-one interaction between a broadcaster and an audience member is not possible in real time, every message and its response is influenced by latency. It is impossible to converse fluently. Meanwhile, one-to-many interaction, in which the broadcaster sends a message to the audience, can make them feel excluded or disadvantaged if the message is lost or arrives later. For example, an audience member watching a sporting event does not want to know when a point is scored by hearing a neighbor cheer, but rather by seeing it and

cheering simultaneously. A client wants to be as close to the live edge as possible, which is the most recent frame available for playback.

The goal of low-latency livestreaming is to reduce video streaming latency as much as possible while keeping the playhead as close to the live edge as possible. The primary goal is to reduce two latencies: segmentation and advertisement latencies. The result of real-time encoding of segments is segmentation latency; the encoder is unable to encode content faster than it can be produced. There is a segmentation latency of two seconds if segments are defined to be two seconds long. Advertisement latency is caused by the need of the client to update its manifest, which advertises the available segments. After encoding has finished, the segment can be appended to the manifest. There is an advertisement latency of a couple of round trips to retrieve the updated manifest.

The use of Chunked Transfer Coding (CTC) when using HTTP/1.1, as explained in Section 2.3, avoids segmentation latency. A client can initiate the request and begin streaming data before the segment has finished encoding. This is made possible by the codec, when it allows for the streaming of partial segments. Ahead-of-time advertisements eliminate advertisement latency by listing segments that do not yet exist but will in the near future. Figure 3.7 depicts a segment sequence as well as the current live edge. A manifest with ahead-of-time advertisements will include both the segments preceding and following the live edge. Manifest updates can be less frequent, and updates can be delayed in favor of more important data, without affecting the ability of the client to request the most recent segments [THE20]. It takes at least six round trips, the amount of round trips for application data to be sent over the network using HTTP/1.1 as seen in Section 2.3, to retrieve the manifest and the first segment using traditional HTTP streaming of segmented content [Cur22a]. If there is no ahead-of-time advertising, each segment will require six round trips if the client wants to stay on the live edge. With this improvement, the client can retrieve multiple segments that will all exist on the live edge before another manifest update is required.

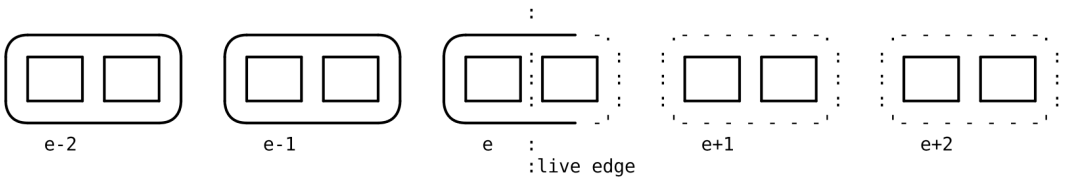


Figure 3.7: A sequence of segments, relative to the live edge. Segment e is at the live edge, the dashed outline of segments indicates that the data is not available at the server; future segments only exist logically.

3.4 Adaptive Streaming

By using multiple bit-rates, adaptive bit-rate (ABR) streaming can improve the streaming experience over progressive streaming and does not require intermediate systems to re-encode content. The source, which handles the requests of the client, serves the content in multiple adaptations with various bit-rates. The ABR client can switch to a different bit-rate to make the most of the network and provide the best QoE. The bit-rate can fluctuate over time, as depicted in Figure 3.8. If network conditions deteriorate, for example, the client can retrieve a lower bit-rate adaptation and continue streaming without encountering buffering. At any time, the most appropriate adaptation of the content that fits the device capabilities, user preferences, and networked environment can be selected.

An application can use a variety of ABR algorithms to determine which adaptation is best suited. Many factors, such as throughput and buffer fullness, can be taken into account by

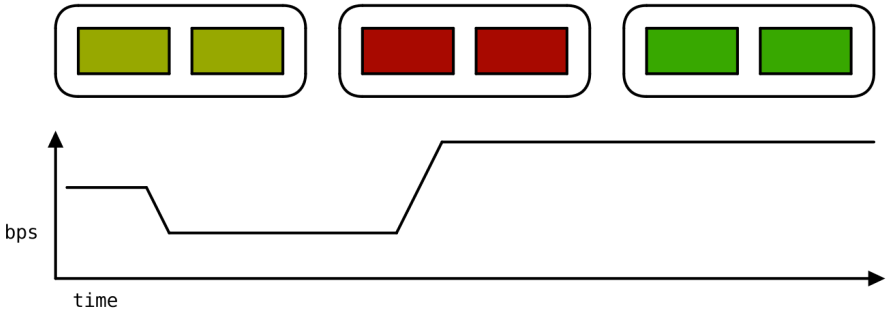


Figure 3.8: A sequence of segments, streamed by the client, and the available throughput at that time. Whenever the throughput lowers, the following segment will have a lower bit-rate, whereas when the throughput rises again, the bit-rate of the chosen adaptation will be higher as well.

algorithms. The specifics of the algorithms are out of scope for this thesis.

The sections that follow go over several popular HAS streaming protocols. Unless otherwise specified, the thesis focuses on HAS, referred to as adaptive streaming.

3.4.1 MPEG-DASH

The Moving Picture Experts Group (MPEG) developed the Dynamic Adaptive Streaming over HTTP (MPEG-DASH) [14] HAS standard, which is referred to as DASH in this thesis. Although it is standardized as a general-purpose streaming technique, it is best known and most commonly used for video streaming. One of the first large-scale public live tests with DASH took place during the 2012 Olympic Games. The event was covered live by the public broadcaster of the Flemish Community in Belgium (VRT) [Uni12].

The manifest uses the Media Presentation Description (MPD) format, an XML-based hierarchical data model. Listing 2 is an example of an MPD, depicting single-layered content with four adaptations. The content is divided into a series of *Period* elements, each representing a temporal segment of the content. This sequence represents the content from beginning to end. A *Period* contains all of the segments needed to play back a specific length of content. An *AdaptationSet* represents each layer of the content, such as audio, video, or subtitles. The standard uses MIME types to distinguish between various types of media, in this case represented by a layer. There are *Representation* elements for these layers, which represent various adaptations of the content, such as the same content at different bit-rates or alternative content, such as subtitles in a different language. Every *Representation* listed within the same *AdaptationSet* should be interchangeable. To choose between these adaptations, the DASH application will employ an ABR algorithm. A *Representation* lists the segments that the client accesses on the server. Segments can be represented by a *SegmentList*, which contains explicit *SegmentURL* elements, or by a *SegmentTemplate*, which shows a template that can be filled in for each segment. It should be noted that the DASH specification is quite extensive, and the method described here is not the only way to represent content using an MPD.

Every segment is synchronized by a shared timeline. There is at least one period in the timeline. The adaptations are mapped to the timeline, making it possible to interchange between adaptations where segments have different lengths. The standard specifies various profiles [14]. A profile limits the scope of what can be defined within a manifest. This ensures interoperability between players and the content. So, while the standard defines a very broad streaming solution, the actual profile defines the possibilities for a given manifest and the described content. For

```
<?xml version="1.0" encoding="UTF-8"?>
<MPD
  xmlns="urn:mpeg:dash:schema:mpd:2011" profiles="urn:mpeg:dash:profile:isoff-live:2011"
  minBufferTime="PT1.500S" type="static" mediaPresentationDuration="PT0H3M0S" >
  <Period duration="PT0H3M0SPT0H9M55.46S">
    <AdaptationSet
        segmentAlignment="true" group="1" par="4:3"
        maxFrameRate="24" maxWidth="480" maxHeight="360" >
      <SegmentTemplate
        initialization="video_$Bandwidth$/segment_init.mp4" startNumber="1"
        media="video_$Bandwidth$/segment_$Number$.m4s" timescale="96" duration="96" />
      <Representation
        id="320x240 47.0kbps" mimeType="video/mp4" sar="1:1" startWithSAP="1"
        width="320" height="240" frameRate="24" bandwidth="46980" codecs="avc1.42c00d" >
        <SegmentTemplate
          initialization="video_46980/segment_init.mp4" startNumber="1"
          media="video_46980/segment_$Number$.m4s" timescale="96" duration="96" />
      </Representation>
      <Representation
        id="854x480 538.0kbps" mimeType="video/mp4" sar="1:1" startWithSAP="1"
        width="854" height="480" frameRate="24" bandwidth="537825" codecs="avc1.42c01e" >
        <SegmentTemplate
          initialization="video_537825/segment_init.mp4" startNumber="1"
          media="video_537825/segment_$Number$.m4s" timescale="96"  duration="96" />
      </Representation>
      <Representation
        id="1280x720 1.7Mbps" mimeType="video/mp4" sar="1:1" startWithSAP="1"
        width="1280" height="720" frameRate="24" bandwidth="1662809" codecs="avc1.42c01f" >
        <SegmentTemplate
          initialization="video_1662809/segment_init.mp4" startNumber="1"
          media="video_1662809/segment_$Number$.m4s" timescale="96"  duration="96" />
      </Representation>
      <Representation
        id="1920x1080 4.7Mbps" mimeType="video/mp4" sar="1:1" startWithSAP="1"
        width="1920" height="1080" frameRate="24" bandwidth="4726737" codecs="avc1.42c032" >
        <SegmentTemplate
          initialization="video_4726737/segment_init.mp4" startNumber="1"
          media="video_4726737/segment_$Number$.m4s" timescale="96"  duration="96" />
      </Representation>
    </AdaptationSet>
  </Period>
</MPD>
```

Listing 2: Example of an MPD, describing a video encoded in four different bit-rates.

example, the VOD profile requires the *type* attribute to be `'static'`, whereas the live profile allows for the same value if the livestream has ended and the content is kept as VOD content. Low-latency DASH (LL-DASH) is a DASH extension that can provide livestreaming with less latency by utilizing CTC and encoding the content into shorter segments [THE20]. This reduces the segmentation latency of HTTP livestreaming.

3.4.2 HLS

Apple developed its own proprietary HAS protocol, called HTTP Live Streaming (HLS) [PM17]. As a manifest, it employs the M3U format, which was originally used to describe MP3 playlists. There are two types of manifests for HLS, as shown in Listing 3. The Media Playlist defines all segments and can be used to stream content directly by a client. The Master Playlist contains references to Media Playlists, which list all existing content adaptations. In contrast to DASH, which is considered a general streaming technique, Apple designed HLS specifically for video streaming. Playlists, for example, only support a limited set of media types, including audio, video, subtitles, and closed captions.

```
#EXTM3U
#EXT-X-STREAM-INF:BANDWIDTH=150000,RESOLUTION=416x234,CODECS="avc1.42e00a,mp4a.40.2"
http://example.com/low/index.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=640000,RESOLUTION=640x360,CODECS="avc1.42e00a,mp4a.40.2"
http://example.com/high/index.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=64000,CODECS="mp4a.40.5"
http://example.com/audio/index.m3u8
```

```
#EXTM3U
#EXT-X-PLAYLIST-TYPE:VOD
#EXT-X-TARGETDURATION:10
#EXT-X-VERSION:4
#EXT-X-MEDIA-SEQUENCE:0
#EXTINF:10.0,
http://example.com/movie1/fileSequenceA.ts
#EXTINF:10.0,
http://example.com/movie1/fileSequenceB.ts
#EXTINF:10.0,
http://example.com/movie1/fileSequenceC.ts
#EXT-X-ENDLIST
```

Listing 3: Example of a HLS Master Playlist, which lists two adaptations of the video stream and a single audio stream. An adaptation is represented by a HLS Media Playlist, this example is for VOD, listing three segments.

HLS is the most popular streaming technology, according to the 2021 Bitmovin Video Developer Report [Bit21], with 73% of the developers using it and another 10% planning to implement it. Only 64% of developers reported using DASH. This is due to iOS, the mobile operating system of Apple, only supporting HLS. Most devices will be supported by HLS, but a considerable market share of users will be excluded by only supporting DASH [dac22a].

Twitch discovered a couple of flaws when using HLS for livestreaming [Cur22a]. With two-second segments, an adaptation switch can occur only after two seconds, which could be enough time

to exhaust the buffer if the network deteriorates. Furthermore, the segment length, manifest updates, and other factors cause a noticeable latency of at least a few seconds. It also takes at least six round trips to retrieve the first segment when using HTTP and TCP, and the buffer requires multiple segments before playback starts.

There are two options for achieving low-latency streaming with HLS. The first option is Low Latency HLS (LHLS), which was developed by Twitter for their streaming platform Periscope [Kal+17]. To reduce segmentation and advertisement latency, it relies on the two improvements mentioned in Section 3.4: CTC and ahead-of-time segment advertisements. Twitch adopted this as a replacement for HLS in order to reduce latency [Cur22a]. It introduced some new challenges; for example, ABR algorithms have difficulty determining the adaptation. However, the most significant issue, latency, was greatly reduced, from multiple seconds to hundreds of milliseconds.

Low-Latency HLS (LL-HLS) is another low-latency option developed independently by Apple [App21]. Five improvements to HLS have reduced latency. Segments are subdivided into much smaller subsegments, as small as 200 milliseconds, using techniques such as CMAF. The playlist updates become more efficient by allowing delta updates, which only transfer the changes that must be made to the playlist. In addition, requests to update a playlist on the server are blocked until there is a meaningful change. A server can notify clients of upcoming segments that can be requested, but the response will be delayed until the segment becomes available[4]. Finally, metadata is added to requests, making switching between adaptations more efficient by reducing the number of round trips required to make that change. Twitch did not use this protocol because it was found to be less performant than LHLS [Cur22a]. In comparison, they observed a quadrupling amount of client requests. Aside from that, latency is significantly lower than in HLS but higher than in LHLS. The adaptation switching, on the other hand, has been improved.

3.4.3 Warp

As next-generation protocols gain popularity, as discussed in Chapter 2, organizations are beginning to develop streaming protocols which utilize specific features of these new protocols. The Media over QUIC (MoQ) mailing list [IET22] is a gathering place for like-minded people discussing many topics, including streaming over QUIC. Warp, a streaming protocol developed by Twitch, is an example of such a protocol.

Warp seems to be the LHLS successor, running on top of HTTP/3 and QUIC. It transports segments in parallel using multiplexed streams, a QUIC functionality. During periods of congestion, these streams can be prioritized to ensure that the most critical data is delivered first. It requires that the segmented content meet a number of criteria, all of which are met by CMAF [Cur22b].

It has numerous advantages over LHLS [Cur22a]. Content data is transferred in only two round trips, allowing for very low-latency streaming in combination with segments that can have lengths defined by the number of frames contained. Another improvement is video underflow, which allows video data to be lost while continuing playback with only audio data. This is not ideal for QoE, but reduces buffering and keeps the client at the live edge. However, as stated in Section 2.2, using QUIC, and therefore UDP, is less efficient than TCP.

[4]In combination with this smaller segments, this is an alternative for CTC.This is very important to enable low-latency streaming with HTTP/2 and HTTP/3, which do not support CTC, as elaborated in Section 2.3.

Chapter 4

Simulation

In a research setting, reproducibility ensures that results are consistent and not a one-off. Simulation helps the research process by securing behavior and parameters that can be observed or controlled from outside. If three main actors are present: the client, the server, and the network, the video streaming architecture discussed in Section 3.2 can be simulated. This chapter will look into the possibility of providing these actors with a simulation.

The term simulation is not well-defined, as it is used interchangeably with the term emulation in very similar contexts. Emulation has the additional goal of attempting to represent the internal state of the target, whereas simulation uses a model to give the outside appearance of a simulated target. In other words, an emulation can replace the target, but a simulation cannot. Emulation is a simulation, but not the other way around [Sta17; Sta13]. To avoid ambiguity, the term simulation will be used throughout the thesis.

The primary goal of simulation is to generate reproducible results; however, the secondary goal can vary depending on the research needs at the time. A simulation can attempt to mimic real-world behavior by being as realistic as possible, which is incredibly challenging. It can also be the goal to approach a theoretical model and try to adhere to determined behavior, such as having worst or best case performance in a given setting.

4.1 Framework

A simulation framework automates the simulation, taking away a part of the manual labor that would otherwise be required. Not only should the framework be capable of performing the simulation, but it should also make simulation setup and reproducing simulations easier. Figure 4.1 illustrates the requirements for a simulation framework for video streaming. It should allow for a single server to act as a content source and connect to a simulated network. The network forwards packets between clients and the server and shapes the network based on a model. The framework should allow for the simulation of multiple clients at the same time, which could be multiple instances of the same client or different clients.

The goDASHbed [RMQ20; ORQ20] is a framework that was developed in tandem with goDASH, as detailed in Section 4.2. It can only test goDASH out of the box and uses the Mininet network simulator, as described in Section 4.4. It does, however, provide the option to test HAS traffic with VoIP background traffic, generated using D-ITG, Distributed Internet Traffic Generator [Búc17]. Furthermore, it can run tests with multiple clients running at the same time.

The next-generation transport protocol QUIC has about fifteen heterogeneous implementations, which is elaborated upon in Section 2.2. The QUIC community maintains the QUIC interop

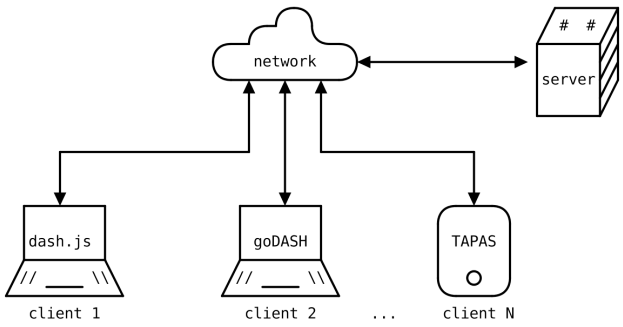


Figure 4.1: Architecture of a simulation framework. A simulated network connects the multiple clients to a server.

runner, a project that aims to document the interoperability of various QUIC implementations. At different time intervals, the most recent version of each available implementation is retrieved, and these versions are tested against each other. This framework also makes use of Docker containers. A reference container shows the interface that testing subject containers need to implement to be compatible. This allows the framework to differentiate between tests that are not implemented and those that, in fact, pass or fail. The tests range from testing specific features to testing the throughput that an implementation can reach. Tests with background traffic can be performed by leveraging iPerf3, a tool developed to measure the maximum achievable bandwidth of a network, but that can be retrofitted to generate background traffic as well [Dug+22].

Vegvisir, a video streaming testing framework based on the QUIC interop runner, was provided by the Networking and Secured Systems research department. It only supports one test, which enables HTTP/3 and requests that the server serve a specific folder. Apart from Docker containers, it also allows host applications to be used as clients. The graphical interface, which is not available with the QUIC interop runner, allows for the management of tests, results, and Docker containers.

4.2 Client

The goal of an authentic client in video streaming is to stream a video with the highest QoE possible. Only HAS clients will be considered for this thesis. To achieve this goal, an application that implements the required features of a HAS client, as detailed in Section 3.4, is required. In summary, the application must be capable of HTTP communication as well as implementing an ABR algorithm.

There are two types of applications that can be considered: a full client or a headless client. A full client is indistinguishable from an authentic client because it implements all four components discussed in Section 3.2, whereas a headless client only requires the control and access components. This means that a headless client will request the segments, but may not use them. There may not be a buffer filled with video data, and the video may not be displayed. While a headless client has less overhead, which may skew results, it also has less stringent hardware requirements to run tests.

The reference DASH client dash.js [Das22b] is an authentic open source client. Based on the qlog logging format discussed in Section 5.2, the research department provided a modified dash.js version with extensive qlog logging capabilities. Even though it requires a browser that is not in headless mode, this client can be used in an automated testing setup. Because browsers keep the application in a sandbox, certain tasks, such as detecting the client finishing its test

or gathering logs, are not trivial. The Vegvisir framework addresses detecting if it is finished by detecting a specific file that is written to the host device after finishing. Because of browser security features, logs may not be written to any file on the host system, changes to the browser settings are required.

A headless client is a standalone application that runs on the host; no browser is required. TAPAS [De +14] is a tried-and-true headless client that supports DASH and HLS streaming. It has over fifty citations, according to Google Scholar. Despite being a headless client, it implements the segment and media components. The video segments are decoded and sent to the media component, which either is a sink or displays the video on the screen. The latter, however, does increase the hardware requirements. While very promising, the client did not appear to be very stable. Even after following the official documentation and running the client in a virtual machine with the recommended OS and software, the client did not work in most cases. In addition, the client does not support many DASH profiles; only a small number of manifests can be correctly parsed.

goDASH [RMQ20; ORQ20] is a more modern headless client that only supports DASH. It has the same flaws as TAPAS in that it only supports a limited amount of DASH profiles and manifests. However, stability is much better. This client does not implement a segment or media component; the buffer is completely simulated. This does, however, allow this client to run in a Docker container. It also offers additional analysis features. It can calculate the QoE of a video segment using an ITU P.1203 implementation[1]. Additionally, it supports the next-generation protocol QUIC. The research department provided a modified goDASH client with several improvements. It not only adds ABR qlog, as discussed in Section 5.2, but it also fixes some serious issues with the application, such as not using persistent connections, wasting network resources for RTT measurements, and a bug in which the client would ignore ABR decisions.

Load generation frameworks that generate specific payloads to test a server are also available [RKM20]. These frameworks may be the furthest removed from a realistic client; for example, it lacks an ABR algorithm and instead has a target bit-rate. They do, however, ensure that the server and network are pushed to their limits if desired. Even with multiple clients, the overhead of a realistic client will give the server time to recover from any problems that may arise. Because of the lower overhead, the simple client used by these frameworks is easily scalable, and fewer clients are required to stress the network and the server.

There are numerous HAS datasets available; however, because most clients only support a limited number of profiles, these may not be directly usable. TAPAS, unlike goDASH, does not provide its own dataset and instead relies on publicly available videos. Other datasets, such as the ITEC DASH dataset [ITE12], contain a large number of video options in a variety of adaptations. It was possible to stream a video from the ITEC dataset using goDASH after some manual adjustments to the manifest.

4.3 Server

The server is the easiest actor to get working correctly within a simulation. VOD requires nothing more than an HTTP file server capable of serving all files mentioned in the manifest. The content is static, and no changes are required over time.

Since the files are not completely static when livestreaming, simulation becomes more challenging. Because the HAS client is based on the manifest, the content can be static. The client will

[1]The ITU-T Rec. P.1203 Standalone Implementation can be found on GitHub: `https://github.com/itu-p1203/itu-p1203`.

be able to livestream the content if the manifest is updated over time, as if it were generated in real-time. This is similar to what is required when VOD content is presented as a livestream. There is no overhead in generating and encoding the content as this is already done, and the segments are added to the manifest chronologically over time. This method could also be used to simulate low-latency livestreaming. This method is implemented by an open source live source simulator maintained by the DASH Industry Forum [Das22a].

4.4 Network

The streaming architecture, as discussed in Section 3.2, is based on a simple client-server model. The transport protocol provides logical communication as if it was a direct connection. The physical connection, on the other hand, is more complicated and must be routed through multiple intermediate networks. To begin a video stream somewhere, you will most likely need to connect to the Internet via your ISP. In addition, as discussed in Section 3.2, the client-server model is inefficient in many modern streaming scenarios. Rather than connecting directly to the server, a client connects to the edge of a CDN. In practice, the intermediate network is regarded as a black box because the endpoints have no way of knowing which path their packets will take. The goal of a network simulation is to simulate packets taking a more complex path rather than being directly connected, without the need to set up an actual complex network, as shown in Figure 4.2. This can be accomplished with either a theoretical or a more realistic model.

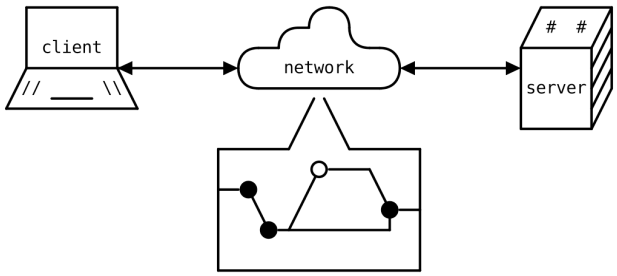


Figure 4.2: The client and server are connected to a network, and do not know what it physically looks like, there is a mist that surrounds the network. A network simulation will act as this black box, while simulating a complex network.

When using a theoretical model, the network will be shaped in accordance with what the model mathematically describes. Packets can be delayed by a predefined amount of time, and a percentage of packets can be dropped or reordered, to name a few features of the shaper. The tc-NetEm [Hem11] simulator uses virtual network interfaces on the host, and manages a queuing discipline (qdisc) that defines the behavior of packets passing through the interface. A qdisc is a queue that takes in packets, and schedules when and the manner of forwarding the packet. The class of a qdisc defines the behavior, a simple example class is First-In First-Out (FIFO), which forwards packets in order of arrival [The22b]. Running commands on the host system sets up the interfaces. Mininet [Clo22b] and ns-3 [nsn22] are network simulators that provide an API to set up the network. Mininet uses process-based virtualization, whereas ns-3 simulates each individual device and uses discrete-event models. Both create virtual network devices that can communicate with the outside world.

By plugging a more realistic model into the previously mentioned simulators, more realistic simulation can be achieved. Akamai, a CDN company, has a model based on 50,000 network traces [Goe+17]. Traces are classified based on their loss percentage, resulting in groups of traces with varying amounts of QoE. Because loss does not always occur, this classification

causes 68% of the traces to be ignored. A set of parameter buckets is created, which are used to define the average network conditions over a specified time period. This model uses 70 ms bucket size, since this was the median RTT. Bandwidth, latency, and loss percentage are among the parameters. The parameters can be replayed indefinitely, one by one, simulating how a network changes over time. Another model is based on measurements obtained in Ghent [Hoo+16]. This model only provides the parameters; it does not provide a means to play them back. Furthermore, in the context of this thesis, the only actual useful network parameter it provides is bandwidth; no other metrics are available. A solution similar to the Akamai model, where the bandwidth is constantly changed to the next value in the model, would be required. A model based on measurements taken in Ireland [Rac+18] is also available. This model captures network metrics using a mobile app and is capable of logging numerous data points related to the cellular network; however, once again, the only useful network parameter is bandwidth. In addition to the Irish measurements, it includes some models based on synthetic measurements generated with ns-3.

However, despite the fact that these models are based on real-world measurements and are referred to as realistic, it is impossible to determine how realistic a model is. The tc-NetEm random loss model, for example, is known to be very unrealistic and thereby deprecated, but it is still used by the Akamai model. The use of the loss model by Akamai, which involves changing the percentage of loss at specified intervals, has not been verified to be realistic. It is also unknown what the impact of changing the qdisc at runtime is, and how buffered packets are affected.

Finally, background traffic can be generated to simulate a network with unstable conditions. Allowing background traffic to interfere with client and server traffic causes their network measurements to vary. Loss and jitter can be simulated without resorting to mathematical models.

Chapter 5

Data and Visualization

Any analysis process requires data. The ability to have high confidence answers to a hypothesis is made or broken by having access to the correct data. This chapter will explore the data that can be generated, how the data can be meaningfully visualized, and how analysis can be facilitated, with a focus on video streaming over HTTP/3, and therefore QUIC.

Actors within the simulation can generate any types of data, which is collected by a framework. In this thesis, for video streaming, each actor generates a unique set of data points. The client generates data from both the transport layer and the ABR algorithm, whereas the server only generates data from the transport layer. Meanwhile, the network simulation can create packet captures of all traffic that passes through it. Furthermore, if real-time analysis is intended, the data format should support streaming.

5.1 Network Logs and Visualization

Packet capturing is the most well-known type of network traffic logging. A packet capture is a file that contains all of the packets and messages sent over the network. The data on the wire is recorded and parsed into a format that includes the headers of each layer, with the payload of the next layer. Applications, such as Wireshark [Wir22], provide a user interface for inspecting packet captures. One drawback of this method of capturing is that the data on the wire may be encrypted, as with QUIC. The application requires the encryption keys in order to correctly parse the packets. Furthermore, an application may have issues detecting the correct protocol, such as by tagging QUIC packets as UDP packets, which is technically correct. A packet capture is a streamable data structure in which each packet can be parsed separately. Certain packets make reference to other packets, such as the response to a request, but these references cause no troubles during parsing. Wireshark can even show packets in real time while capturing traffic. There are network simulators that use proprietary logging formats; however, these formats are not taken into account because they are not standardized and are not used by more than one application, and the most important events can be read out from the packet capture.

A packet capture can be used to visualize a network, as shown in Figure 5.1. A map can be built by tracking the flow of packets and which nodes are communicating with one another. This map can be represented in a variety of ways. A link map shows which nodes are connected; however, if there are many connections, this can become very cluttered, especially if a single node has proportionally more connections than the other nodes. To reduce clutter, a node map only shows certain attributes of each node, such as a size proportional to the number of connections. Another way to reduce clutter is to use a more abstract matrix display, in which connections can be read by finding the cell at the intersection of two nodes [BEW95]. The massive sequence view

visualization can be valuable if it is important to see the traffic flow over time [Elz+14].

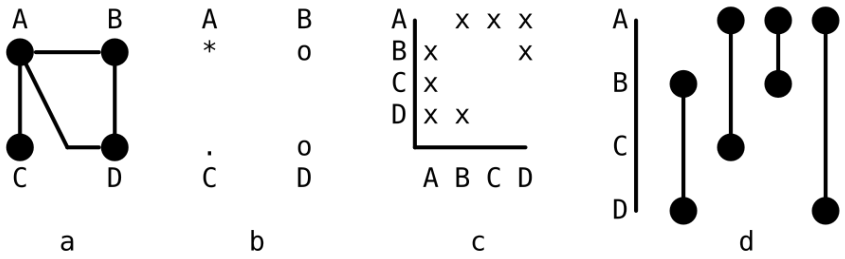


Figure 5.1: (a) The link map, shows connections between nodes. (b) The node map, uses icons to convey data, in this case the icon represents the amount of total connections to a node. (c) The matrix display, shows if two nodes have a connection by marking the intersection of the nodes. (d) The massive sequence view, it shows which connections appear over time.

5.2 Client and Server Logs

Most HTTP/3 implementations support data logging at both the transport and application layers. Logs can be exported in either a proprietary or standardized format; some implementations support both. While a proprietary format may give valuable insights, the fact that it is incompatible with other implementations forces the thesis to seek out a well-adopted standardized format.

The majority of QUIC implementations support the high-level logging format qlog [Mar+18]. It describes a set of principles for event-based logging. Events can be categorized and typed, allowing an analysis application to differentiate between them, whilst timestamps ensure that they are ordered chronologically. The general format and principles are defined by a main schema [MNS22b]. Other documents specify protocol events, such as QUIC [MNS22c] and HTTP/3 and QPACK [MNS22a]. Listing 4 represents a qlog file in JSON notation, a human-readable data-interchange format [Ecm17]. A qlog can be built from both the client and server perspectives. While it is standardized, it does allow for some flexibility in implementation, which can be beneficial or detrimental. Timestamps, for example, could be logged in a variety of ways. While parsing was not a problem, it was extremely difficult to correlate two qlog files because the starting point was not always documented in a comparable manner. The summary field in the top level of the file is a good example of freedom because it allows for any data to be represented, allowing extra metadata to be shared. The general structure of a qlog is as follows: a top level object describes the file and contains traces. A trace is the log of a single connection, from a certain perspective. The trace includes metadata as well as a list of events. Since qlog defines event definitions using schemas, it is possible to define a new schema for a protocol. An early draft of a qlog schema for a HAS application and the ABR algorithm was provided by the research department. Most major events, such as starting and stopping the video player or when there is buffering, are defined.

NetLog is used by Chromium-based browsers to perform event-based logging of network protocols [RM22]. NetLog focuses on application events and is not intended to replace lower-layer tools such as those that generate packet captures. There are, however, numerous network-related events that are also logged. NetLog supports QUIC and HTTP/3, allowing it to obtain logs of how the QUIC implementation of the browser functions, which does not support qlog. However, as mentioned in Section 5.3, qvis can convert a NetLog file to a qlog file.

```
{
    "qlog_version": "0.3", "qlog_format": "JSON",
    "title": "Name of this qlog",
    "description": "Description for this group of traces",
    "summary": {
        "trace_count": 1, "max_duration": 5006,
        "max_outgoing_loss_rate": 0.013,
        "total_event_count": 568, "error_count": 2
    },
    "traces": [
        {
            "title": "Name of this trace",
            "description": "Description for this trace",
            "configuration": {
                "time_offset": 150
            },
            "common_fields": {
                "ODCID": "abcde1234", "time_format": "absolute"
            },
            "vantage_point": {
                "name": "backend-67", "type": "server"
            },
            "events": [
                {
                    "time": 1553986553572,
                    "category": "transport", "type": "packet_sent",
                    "data": {
                        "packet_size": 1280,
                        "header": {
                            "packet_type": "1RTT", "packet_number": 123
                        },
                        "frames": [
                            {
                                "frame_type": "stream",
                                "offset": 456, "length": 1000
                            },
                            {
                                "frame_type": "padding"
                            }
                        ]
                    },
                    "protocol_type":  ["QUIC","HTTP3"],
                },
                ...
            ]
        }
    ]
}
```

Listing 4: An example qlog file in the JSON format. This example was adapted from the main schema RFC [MNS22b].

The qlog format is based on JSON, but there are alternative representations. NDJSON is a format where multiple JSON objects can be contained within a file, if they are split by a newline character. While JSON is not streamable, NDJSON was developed to enable structured data to be streamed [ndj22]. Another streamable format, based on JSON, is JSON Text Sequences (JSON-SEQ). In this case, every JSON object is split using the special record separator character [Wil15]. With these streamable formats, the top level and trace metadata in qlog will be a single JSON object, followed by a single JSON object for each event. Some attempts have been made to create formal abstract definitions of JSON, which can be represented by a tree, as well as methods for querying these structures [BRV20; FF13]. Each has advantages and disadvantages. GraphQL [Fou] is a querying language developed by Facebook for querying graph-like structures. The queries and mutations use a format that resembles JSON, and return actual JSON data. A query retrieves data, while a mutation updates the data. A schema defines the queries, mutations, and data types; an example is shown in Listing 5.

```
type Person {
  id: ID!
  name: String!
  favorite_number: Float
}

type Query {
  persons: [Person!]!
  person(id: ID!): Person
}

type Mutation {
  addPerson(input: PersonInput!): Person!
}

input PersonInput {
  name: String!
  favorite_number: Float
}
```

Listing 5: A GraphQL schema. A type to represent a person is defined, with some attributes. Queries are defined to either retrieve all people or only a single person by their ID. A person can be added by using a mutation, which uses an input type, since the ID is generated by the data source in this example.

5.3 Client and Server Visualization

The qlog format, as discussed in Section 5.2, appears to be the most appropriate logging format. The QUIC and HTTP/3 qlog events can be visualized using the tool qvis [Mar+20a]. It not only supports the qlog format, but also packet captures and NetLog, which are internally transformed into qlog. It features five analysis views after loading data. A view shows some global statistics about the qlog file and the traces it contains. The other views give a more detailed view of the traces. Figure 5.2 shows an example of a sequence view displaying the communication flow between a client and a server. All streams and the way are multiplexed have their own view, and the packetization of the transferred data is shown in another view. Finally, as shown in Figure 5.3, a view displays both congestion control metrics and RTT measurements throughout the session. Data cannot be streamed to qvis for real-time analysis. This type of visualization

could be classified as a guided experience because the views are pre-planned. Grafana [Lab] is an application that facilitates making dashboards to visualize data queried from a data source. It comes with a variety of predefined visualization and data source types, but custom plugins can be developed. The main use case of Grafana is querying a data source at timely intervals, to visualize real-time data. Grafana is very flexible; almost everything can be modified by parameters. Since it is less of a guided experience, except for some dashboards that might be pre-planned, it can be daunting to get hands-on with a Grafana dashboard. Figures 5.4 and 5.5 show two examples of Grafana dashboards and the possible built-in visualizations.

When visualizing events, keep the three Gestalt principles in mind: closure, proximity, and similarity. These principles are depicted in Figure 5.6. Because of the closure principle, humans tend to perceptually complete objects that are not, in fact, complete. This principle goes hand-in-hand with the proximity principle, which states that humans perceive objects that are close to each other as a group. According to the similarity principle, humans will find similar appearing objects and categorize them into groups. There are also some significant temporal patterns. Trends are changes in the value of a metric that occur over time, whereas a counter trend is a change that contradicts a previously observed trend. There can also be periodic repetitions of certain events or metric values, or a disruption in the repetition. There is an anomaly whenever a pattern emerges that behaves differently than all previously observed patterns [Elz+14]. It is also essential to have control over the visualization parameters in order for the data to be understandable. By emphasizing what is determined to be more important to view, parameter focussing avoids visual overload. This also implies that unwanted data must be removed. The identification of data points will aid comprehension of what is being examined, and animations can be used to visually distinguish obvious unexpected changes [BEW95]. It can be concluded, that while a guided analysis tool can be straightforward to use, it is important to have enough freedom to ensure understandable views.

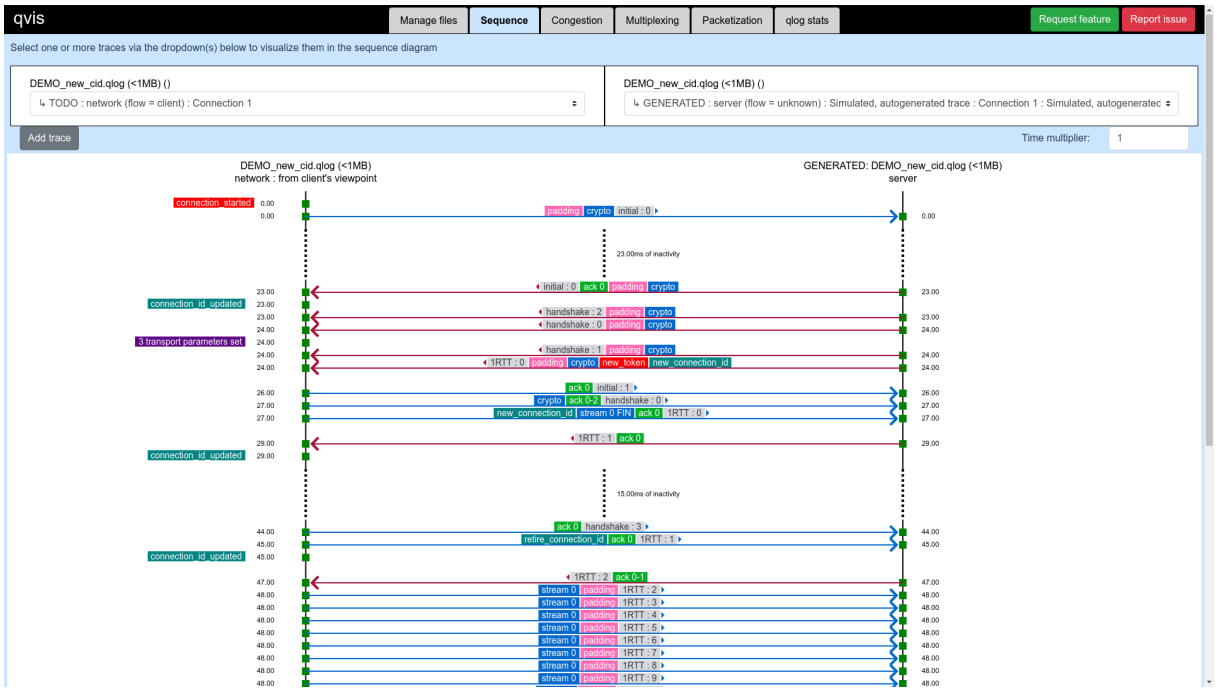


Figure 5.2: The qvis view showing a sequence diagram, detailing the communication between a client (left) and a server (right).

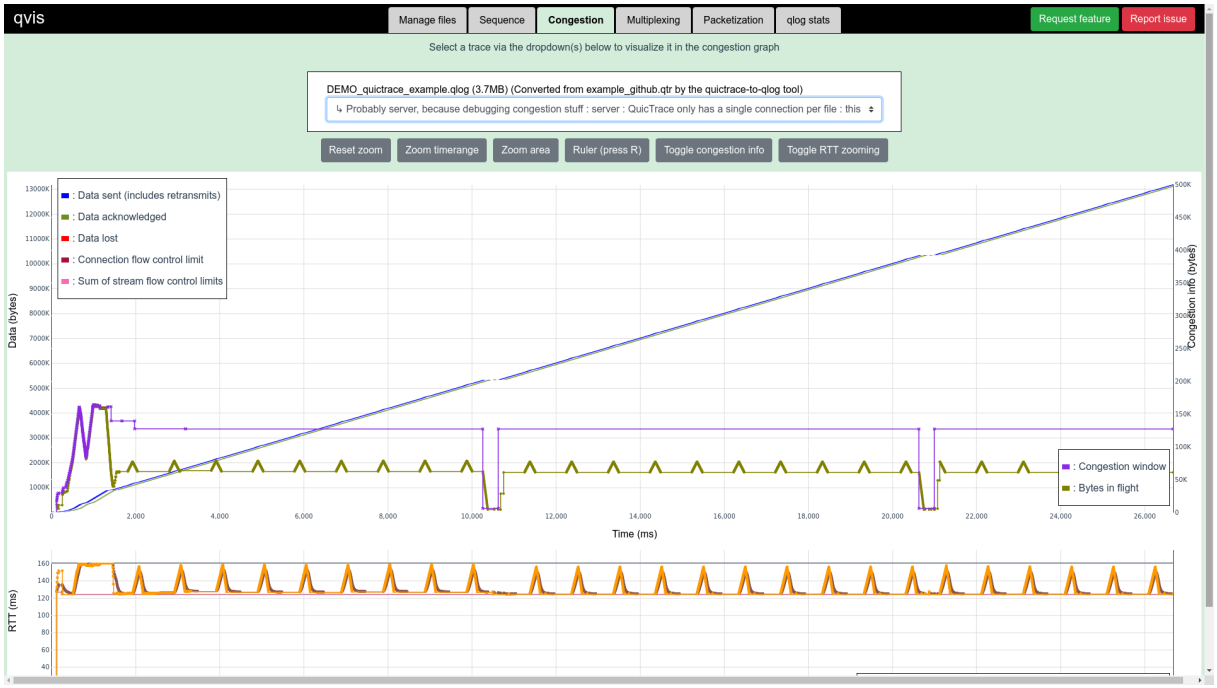


Figure 5.3: The qvis view showing congestion control metrics (top) and the RTT measurements (bottom).

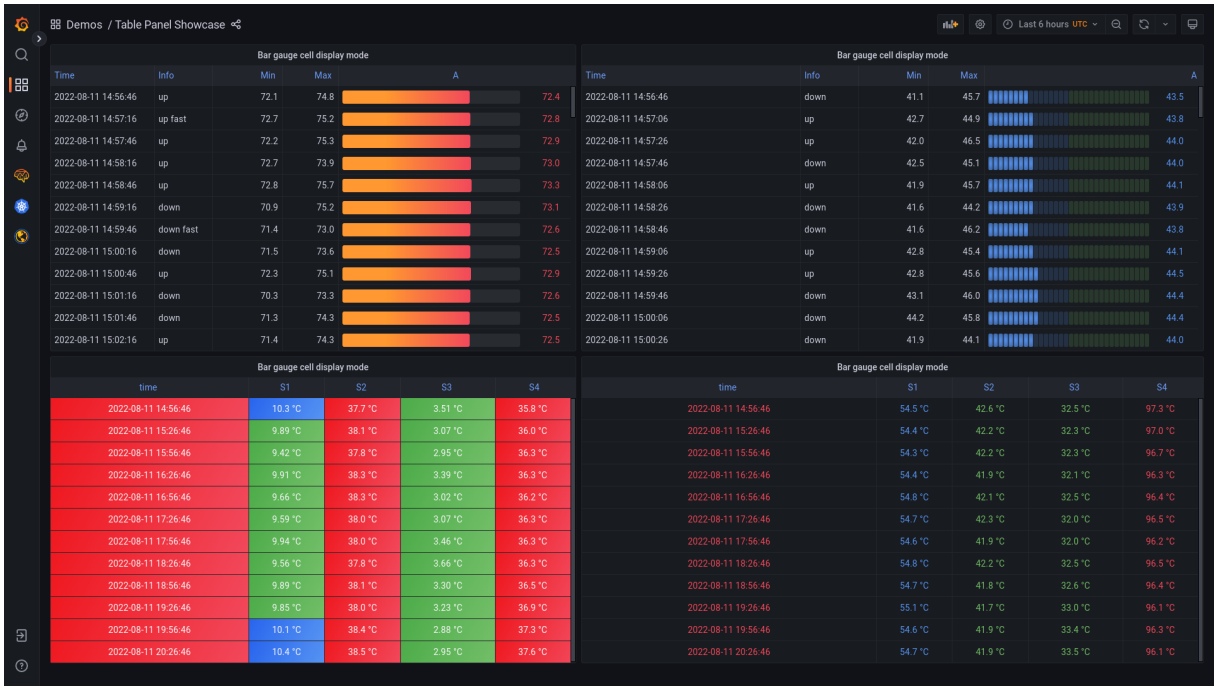


Figure 5.4: A Grafana dashboard showing the same temperature measurements in four different ways, all based on a table visualization. This dashboard is based on the table panel showcase demo.

Figure 5.5: A Grafana dashboard showing synthetic site data. Visualized are, among others, server requests, the amount of hits on Google, and the amount of logins. This dashboard is based on the Grafana dashboard demo.

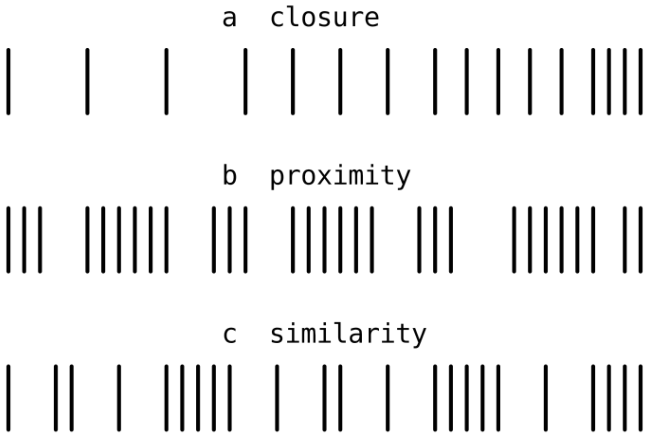


Figure 5.6: The Gestalt principles. (a) The closure principle states that humans tend to perceptually complete objects that are not complete. As the gaps grow smaller, the larger objects will be perceived. (b) The proximity principle states that objects close to each other are perceived as a group. Instead of single lines, seven objects made up of grouped lines will be perceived. (c) The similarity principle states that similar objects are grouped. Three types of objects can be observed: lines, double lines, big blocks [Elz+14].

Chapter 6

Analysis Framework

This chapter discusses the contributions made by the thesis. The most significant contribution was the addition of an analysis service to the Vegvisir framework, as detailed in Section 4.1. Aside from that, applications are modified to work better with the analysis service or to correct erroneous behavior.

Analysis is the act of examining something with the goal of understanding it better [Dic22]. With the gained knowledge, it is possible to act accordingly and improve the current situation. This thesis differentiates between two types of analysis. The first, real-time analysis, takes recent data, analyzes it, and based on known patterns decides the immediate action to take. During video streaming, the ABR algorithm performs this kind of analysis, it takes some measurements, and based on that data it decides what adaptation to choose. The algorithm has the knowledge of what the expected result will be of its decision, based on the measurements. Knowledge can also be gained over a longer period of time, not real-time, this is the second kind of analysis. With this kind of analysis, the goal is to detect patterns, and contrive improvements accordingly. This is the kind of analysis that would normally happen at an analysis service. An example is qvis, in this case, a QUIC implementation will be used in tests, and their qlog will then be analyzed at a later time. Performing a thorough search for anything that can be improved, instead of solely focussing on known patterns.

6.1 Analysis Service

The goal of an analysis service is to assist a human in the analysis process. It should be able to gather data, perform calculations on the data, and display it in a useful manner. Furthermore, for this thesis, the analysis service must be able to perform this in real-time. The goal is to make the root cause analysis process easier in this way, because while an anomaly may be noticeable, the actual cause may be well hidden. This means the analysis service should help identify problems, differentiate between times of normal and abnormal behavior, and help distinguish between the root cause and correlated but uninfluential factors [Wik22]. The technologies mentioned in the previous chapters are evaluated and combined into a coherent service, that attempts to meet these requirements.

Not only during testing, but also in real-world applications, applications generate a plethora of logs. While streaming content, the Netflix application sends data to a server, presumably to measure QoE [Adh+12]. Telenet, another ISP, has an IP return data path for their set-top boxes. This path can be used for interaction between the set-top box and a content provider, independent of the internet connection supplied to the user, for example, to send reports back [Tel21]. Based on these real-world use cases, it can be concluded that the goal of the analysis

service, to receive these reports in the form of logs and then facilitate analysis, is not entirely theoretical.

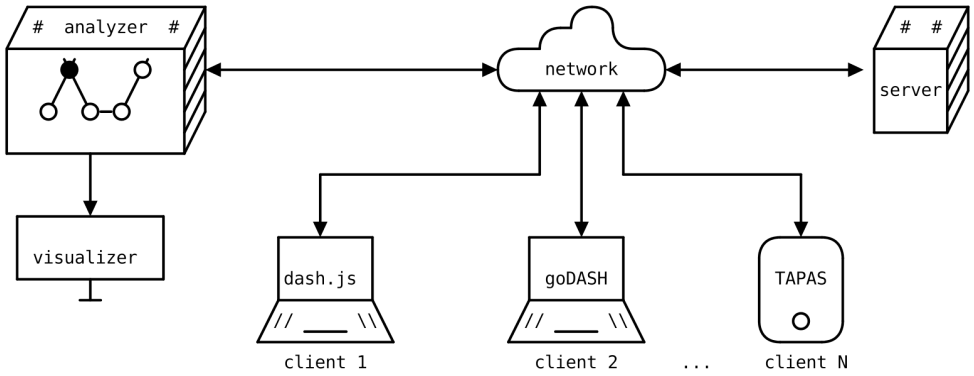


Figure 6.1: Architecture of the analysis framework, which is based on the Vegvisir framework. The analysis service, consisting of an analyzer and a visualizer, are on the left, connected to the same network as the simulation framework.

The analysis service consists of two parts, as shown in Figure 6.1 on the left. The *analyzer* handles data ingress and serves as a data source for other applications. It must be connected to the testing network in order to gather data and allow other applications to access it. It is assumed that clients will always send logs to the analysis service. Because the provider may not have access to the CDN logs, if a CDN is used, network and server data are regarded as optional. The analyzer is a GraphQL endpoint implemented in a custom application, that is able to perform some basic actions on the data, such as changing the representation of fields and aggregating data. The *visualizer* queries the data, and allows visualizing the data in various ways to aid analysis. This part is based on Grafana. In the background, the framework starts the analysis service. This way, the service can collect data from various test runs and will remain operational after the tests are completed.

6.1.1 Ingress

The analyzer collects the ingress data, which is generated by the entities participating in the simulation. Section 5.2 discusses qlog, a high-level logging format that is supported by many QUIC implementations. It is the preferred format for ingress data due to its granularity, flexibility, support for both transport layer and application-specific events, and adoption by the majority of QUIC implementations. There is no network simulator that supports this format; however, because the intermediate network is considered a black box, the logs from this entity are not taken into account during the analysis process. The generated logs are event-based, the analyzer supports applications to stream either single events or batches of events. It also supports different file formats for qlog, JSON, ND-JSON, and JSON-SEQ.

An additional attribute is added to the qlog schema to aid analysis. The new field analysis ID allows aggregating multiple traces from different layers within an application, or from different entities. During a test, multiple qlog files with very similar IDs will be generated. Only the suffix can differ, representing the role of an entity, specifically client or server, or distinguishing two entities with the same role, such as by adding a unique identifier. For example, if "`test-id`" is the base ID, "`test-id_server`" indicates the server logs, "`test-id_0`" indicates a client, where the added number identifies the different clients, "`test-id_1`" indicates a second client etc.

Furthermore, the ABR qlog schema, which is discussed in Section 5.2, is expanded. Listing 6 shows a selection of the modifications. It was extended with a "metrics updated" event, which was inspired by the QUIC event of the same name. The QUIC event is part of the recovery

category, which describes congestion control behavior; however, in the ABR qlog, it is classified as generic. The event includes RTT measurements, which can be taken at various layers, such as the transport and application layer. The new event, which is logged by the streaming client, consists solely of RTT measurements, which are performed in the same manner as in the QUIC implementation, the other congestion control metrics are removed. Another event was added to represent a decision made by the ABR algorithm, and the basis for that decision; currently, this event is focused on throughput-based algorithms. Aside from that, existing events were updated to support more representations of the same data, primarily because these representations were more convenient.

```
type EventMetricsUpdated {
  min_rtt: Float
  smoothed_rtt: Float
  latest_rtt: Float
  rtt_variance: Float
}

type EventABRDecision {
  segment: Int!
  algorithm: String!
  throughput: Int!
  delivery_time: Int!
  delivered_bits: Int!
}

type EventABRSwitch {
  from_id: String
  from_bitrate: Int
  from_width: Int
  from_height: Int
  to_id: String!
  to_bitrate: Int
  to_width: Int
  to_height: Int
  media_type: MediaType!
}
```

Listing 6: New and improved ABR qlog events. The metrics updated event is based on an event defined by the QUIC qlog schema, but only carries the RTT data. The ABR decision event is a novel event, used to represent both the decision by an ABR algorithm, and the reason for making that decision. The ABR switch event, which appears if the ABR algorithm changes its target adaptation, got extended with more representations of the new target adaptation set.

6.1.2 Querying

Section 5.2 discusses how the qlog is based on the JSON format and how JSON can be queried, and GraphQL appears to be the best option for this analysis service. Partially because both define the data structures using a schema, the idea being that the conversion into the GraphQL schema would enable multiple applications to query qlog files. Since both are influenced by JSON, converting the qlog schema to a GraphQL schema is simple. However, issues arise when it comes to the types of data. The qlog specification has less stringent guidelines because it is a

high level logging format that allows the implementation some leeway in how it represents certain fields. Fields representing timestamps are an example of this. Previous qlog versions permitted any representation, such as strings, as long as another field declared what the value represented. According to the most recent version of the qlog specification, these should be represented in milliseconds, which is a far more convenient representation to work with. GraphQL is more stringent about types; the exact type must be declared at all times. The type of field was thus chosen based on both the qlog specification and what was most convenient for the visualization of a field; for example, numbers are preferred over strings since numbers can be interpreted more easily without additional parsing. Furthermore, GraphQL does not support certain types that are used within qlog, such as nested unions, which had to be flattened for the GraphQL schema. To query data from the analyzer, the visualizer uses the Grafana GraphQL Data Source plugin[1].

Both mutations and queries are declared in the GraphQL schema. The mutations are used by the simulated entities to send the generated data to the data analysis service. Firstly, a qlog instance with a unique ID must be created. Second, the implementation can use the ID to add a trace to the qlog instance; this trace is also identified by a unique ID. Finally, using the trace ID, events can be streamed into the "events" field of the trace. Each mutation has a corresponding input type, that is used to pass data to the mutation. The events should be sent as a JSON string because it is impossible to declare a single type for this field due to technical limitations; GraphQL requires an exact type for the input, not a union of all event types. The queries are used by the visualization service to query the aggregated data. All known qlog instances and analysis IDs can be queried for. Both of these can be used to query traces, and the traces can be used to query events, annotations, and high level controls that Grafana can use. The types declared by the main GraphQL schema are used in the queries.

A GraphQL type that implements the Grafana annotation interface is shown in Listing 7. Annotations are a feature in Grafana that allows a global mark to be set at a specific timestamp. It requires a timestamp and, optionally, the end time if a range is to be marked. Furthermore, it requires a title and, if desired, a description. Finally, annotations are classified using tags. Since annotations can be used to represent non-numerical data, not every event will have a corresponding annotation.

```
type Annotation {
  time: Float!
  end_time: Float
  title: String!
  description: String
  tags: String!
}
```

Listing 7: A GraphQL type that implements the Grafana annotation interface.

6.1.3 Visualization

Grafana is used to visualize event-based data generated by clients and servers. It is necessary for the data source being queried to provide an absolute time value, such as a UNIX timestamp. However, in a qlog, this is not always the preferred time representation. The analyzer corrects this by recalculating the timestamp into an appropriate representation. Another issue is the

[1]The GraphQL Data Source plugin can be found on GitHub: `https://github.com/fifemon/graphql-datasource`.

desire of Grafana to display real-time data. This is especially useful when streaming data into the visualization and performing real-time analysis. However, if a closer look at older data is desired, it is time-consuming to determine when the data was recorded. A query that returns the time range where the events appear as a relative link was added. By clicking the link, the dashboard will show the correct time range containing all events. The dashboards can display both numerical and textual data by querying the data source and selecting an appropriate visualization technique. Annotations can be used to show non-numerical data. Finally, the visualizations can be exported using the Grafana Image Renderer plugin[2].

The analysis service provisions the GraphQL data source and a number of pre-configured dashboards for Grafana. Figure 6.2 and 6.3 show the oldest dashboards, which are used to explore the visualization possibilities of Grafana with the qlog data for both QUIC and ABR. Based on these explorations, a new dashboard was created which compares the QUIC and ABR traces from a streaming session. This combination dashboard is shown in Figure 6.4. Two more dashboards were developed for the evaluation phase, which is described in Chapter 7. Figure 6.5 demonstrates the dashboard for viewing the QUIC traces of both a client and a server. The final dashboard, shown in Figure 6.6, can be used to compare two ABR traces. When comparing data from two sessions in Grafana, the timestamps will not match if the tests are performed at different times. This is solved by calculating the difference between the time of the initialization events, and then using that difference to shift the events of the second trace so that both traces are shown on top of one another.

[2]The Grafana Image Renderer plugin can be found on GitHub: `https://github.com/grafana/grafana-image-renderer`.

Figure 6.2: A Grafana dashboard visualizing QUIC qlog events. The trace metadata is listed in the table at the top. The panels below show, from top-left to bottom-right, the RTT measurements, aggregate data of the RTT measurements, the occurrences of specific events, the maximum amount of data sent, and the data in flight.

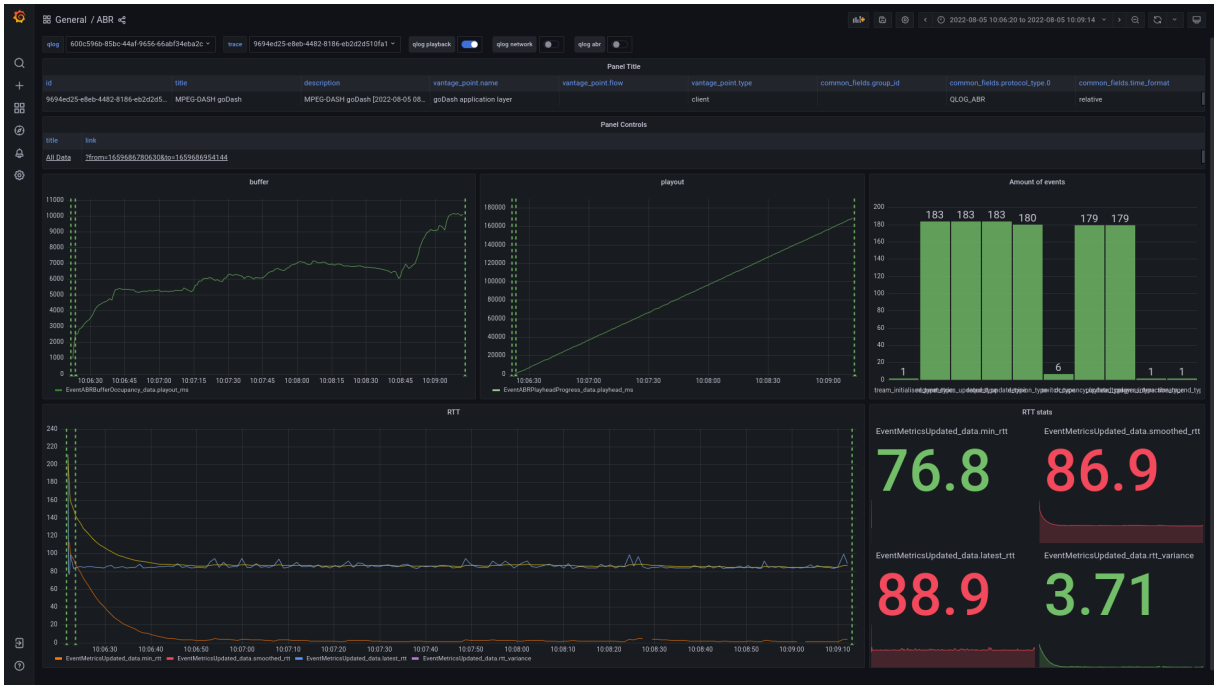


Figure 6.3: A Grafana dashboard visualizing ABR qlog events. The trace metadata is listed in the table at the top. The panels below show, from top-left to bottom-right, the buffer occupancy, the playout time, the occurrences of specific events, the RTT measurements, and aggregate data of the RTT measurements. Furthermore, annotations, the green vertical lines, show the stream initialization, when the stream playback can start, and the end of the stream.

Figure 6.4: A Grafana dashboard visualizing QUIC and ABR qlog events of the same stream session. The metadata of the traces is listed in the tables at the top. The panels below show, from left to right, aggregate data of the ABR RTT measurements first, and then for the QUIC trace, with the playout time below that, on the right, it shows the QUIC RTT measurements op top and the ABR RTT measurements below that.

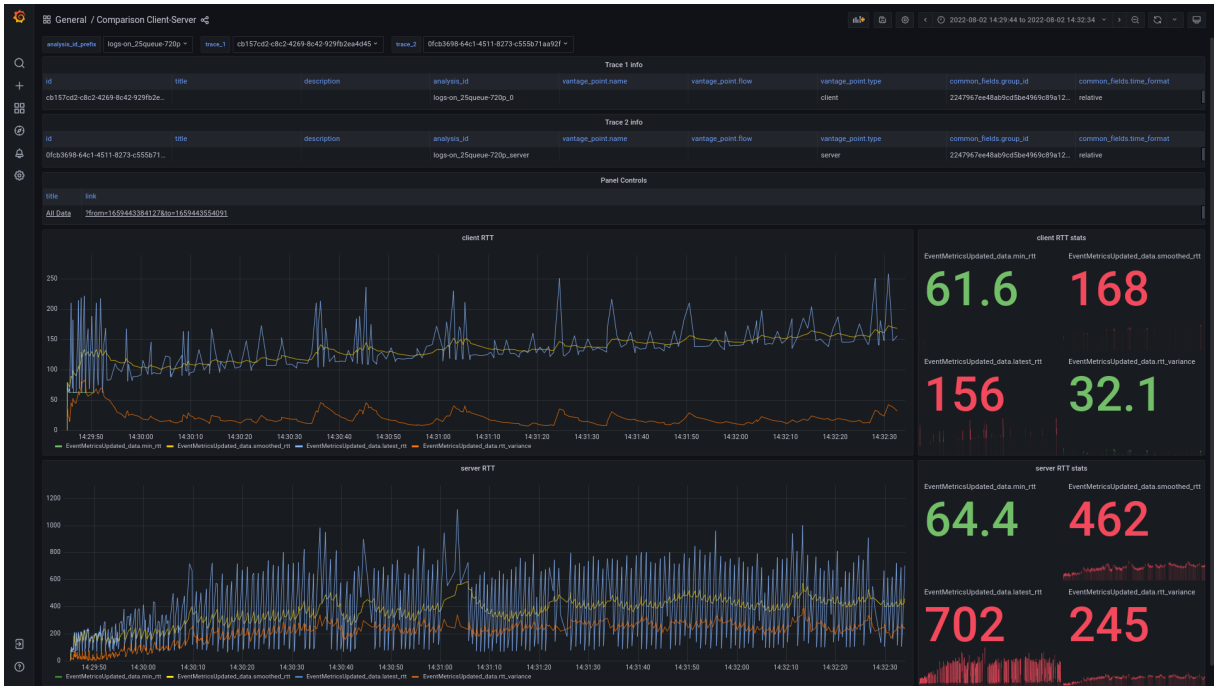


Figure 6.5: A Grafana dashboard visualizing QUIC qlog events from both the client and server. The metadata of the traces is listed in the tables at the top. The panels below compare the RTT measurements of both traces, both as a graph and their aggregates.

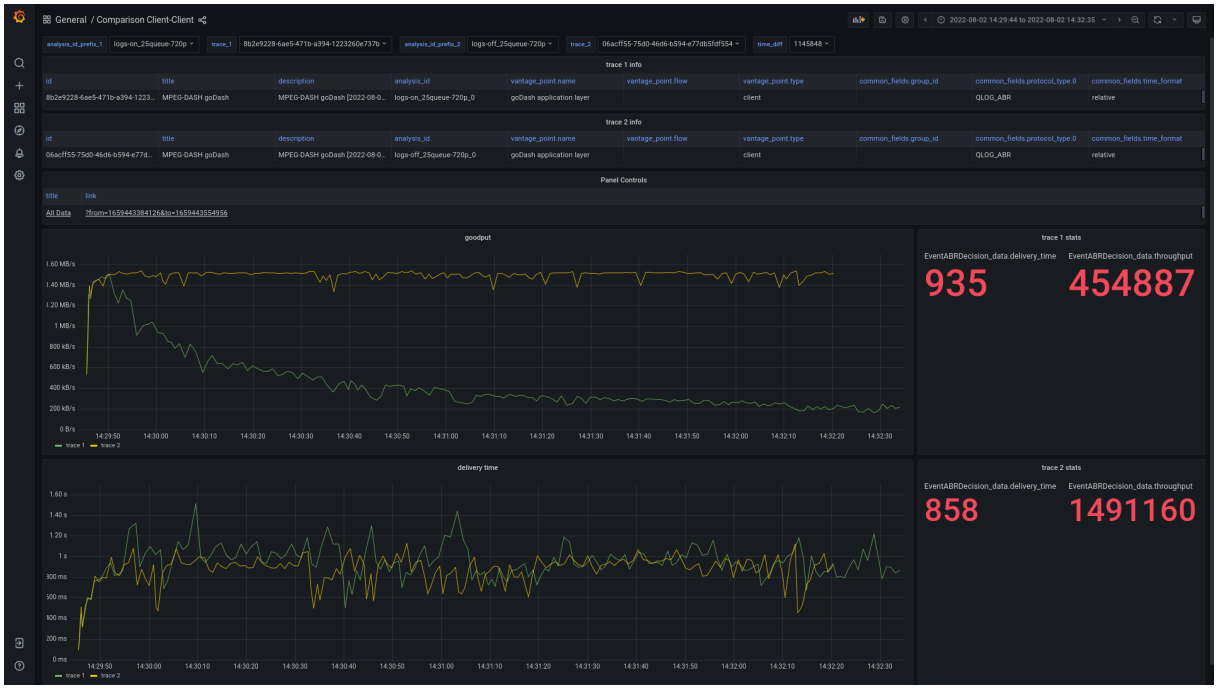


Figure 6.6: A Grafana dashboard visualizing ABR qlog events of two separate sessions. The metadata of the traces is listed in the tables at the top. The panels below compare the goodput and the delivery times. It also includes aggregate data of those metrics.

6.2 Simulation Subjects

Only the goDASH client meets all the requirements discussed in Section 4.2, with a few minor modifications. The modified client can log data using qlog at both the transport and ABR layer. The transport layer uses quic-go, but instead of writing qlog events to file, it allows them to be streamed to the analysis service. Streaming can be executed in a few manners, either individually or in batches. The logging routine, that is a separate routine from the main streaming routine, receives events and processes them. Individual events are immediately processed and sent to the analysis service. If batched sending is enabled, events will be stored until the batch size is reached, and then all events are processed at once. Batching events results in less network request being made, but also results in the logs arriving later at the analysis service. This qlog implementation is ported to the ABR layer, where it uses the exact same functionality, but with a different set of events. The container used to run the clients is adapted to support running multiple clients at the same time with the same settings. The dash.js player provided by the research department is an alternative, but it does not provide as extensive transport layer logs as goDASH, it is more difficult to automate, and possibly impossible to have multiple clients running at the same time, due to how a browser executes JavaScript on a webpage that is not in focus.

During a test, any HTTP/3 compliant server can be used as a server. When the server logs are required during analysis, a quic-go server is developed that implements the streaming functionality. Aside from that, no server implementations are changed.

Because measuring the impact of specific parameters is more convenient with a theoretical model, the network simulator ns-3, discussed in Section 4.4, is used. The realistic models are not verifiable, and inconsistent over time, which is a feature of being realistic. If the network does not change unexpectedly, it will be clear how certain changes in the testing environment affect the behavior of a client. Several versions of the network simulator are created, for example, with different qdisc classes.

6.3 Preliminary Testing

Many smaller preliminary tests were carried out during the development of the analysis service to ensure that it works properly. Several oddities were discovered during these tests. While certain errors were found within the adopted implementations, others were introduced by the service itself. Grafana offers a lot of freedom, which is useful when trying to find the best visualization for a specific data point. However, freedom can be a source of errors. With so many parameters to tune, it can take some time to ensure that each parameter improves the visualization while not hiding any important data. This section discusses a selection of interesting anomalies that were discovered.

One of the first preliminary tests appeared to exhibit bufferbloat characteristics, as detailed in Chapter 2. This was further investigated, since there was no obvious cause for this behavior. It was discovered that the UDP receive buffer size was too small, which was causing this. Because QUIC runs on top of UDP, the UDP stack influences QUIC performance; the UDP receive buffer size can result in limited throughput. Some QUIC implementations, including quic-go, which was used in these tests, attempt to increase the size of this buffer but are unable to do so because they exceed the kernel buffer size limit. During tests, the value of the UDP receive buffer in the kernel should be set to approximately 2.5 MB to ensure a large enough buffer [See21].

Another issue discovered was a disparity in RTT measurements between the transport and application layers. The transport layer calculates the RTT for each packet at a granular level. It is measured when an acknowledgement of a packet arrives. However, at the application layer,

the unit of data is video segments. This can also be used to measure RTT, but at a much more coarse level. The RTT in this context refers to the time it takes between executing the function call for a segment request and receiving the response object. Furthermore, because the segment request is not a blocking function call in the implementation of the client, the data may not be completely available at the client when the function returns the response. The transport layer streams the data into the response object in the background, however, this asynchronous behavior will become blocking, and wait until all packets have arrived, if the data is being explicitly read from the response object. Because the RTT values describe various types of data units, it is unclear what a comparison of these values represents. Attempting to compare these values resulted in incomprehensible graphs, as illustrated in Figure 6.7. The solution was to ignore RTT measurements at the application layer and instead add a new event to the ABR qlog that included delivery time, which is the time it takes to receive a complete segment.

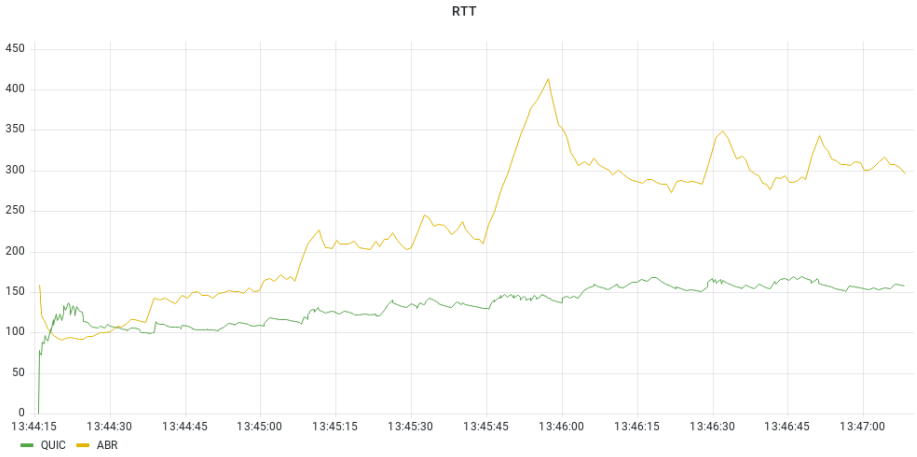


Figure 6.7: A graph showing the smoothed RTT measurement from the QUIC and ABR qlog of the same streaming session. Anomalies occur in the beginning, where the ABR measurements are much higher than expected, then the ABR measurements are lower than the QUIC measurements, which should not be possible, and finally, throughout the session, the ABR measurements show unexpected spikes.

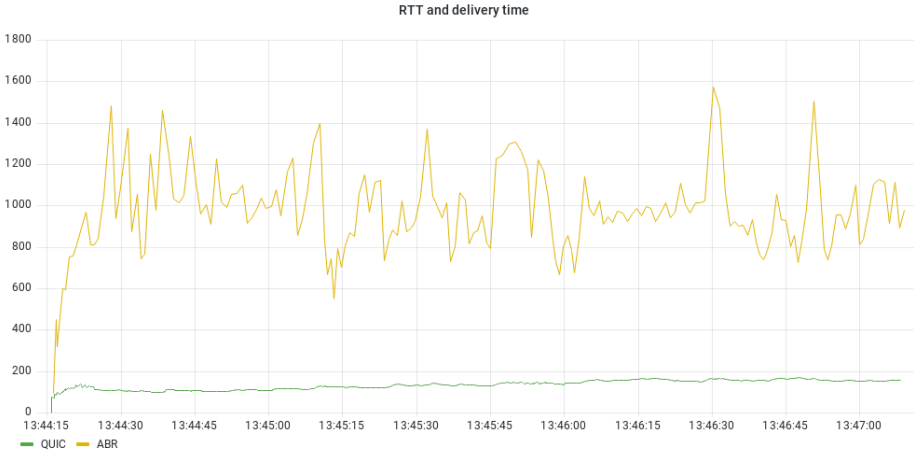


Figure 6.8: A graph comparing QUIC RTT measurements to ABR delivery time measurements. The delivery time is higher than the RTT, as expected, however, the values are too high, caused by application overhead when calculating QoE values.

Another anomaly was discovered while investigating the RTT measurements. With no obvious cause in sight, the time between requesting consecutive segments was always a few hundred milliseconds longer than expected, shown in Figure 6.8. The function calculating QoE values

caught the eye while tracking down the actions performed between requests. The ITU P.1203 implementation used by goDASH is a Python script that goDASH executes to as an external process. It starts a new asynchronous routine for each calculation, but it makes a blocking call while waiting for the calculations to finish. The delay between requests was caused by this blocking call, which disappeared after turning off the QoE calculations. The data needed to calculate QoE, however, is still available, so this can be done after the fact.

Chapter 7

Evaluation

This chapter will describe the outcomes of numerous tests that were run and analyzed using the analysis framework described in Chapter 6. The tests will compare the streaming performance of a HAS streaming client in different scenarios. Section 7.1 expounds the set of parameters that can vary between tests, as well as the most important metrics gathered during the tests. Following that, numerous tests in Section 7.2 will evaluate the chosen client. Finally, Section 7.3 assesses the analysis service itself, specifically how it facilitated the analysis process.

7.1 Methodology

The evaluation is carried out by running tests using the framework described in Chapter 6, which also explains why the client, network simulator, and server were chosen. Different parameters will be modified between tests, but the metrics collected will remain the same. The results of various tests will be compared, and their analysis will be performed in Section 7.2. Unlike preliminary tests, this evaluation phase solely uses the analysis service to analyze the gathered data. The analysis service is also used to generate all graphs. Since these use absolute time, the x-axis shows time in the "hour:minute:second" format, which refers to the time when the test was performed.

7.1.1 Parameters

In order to evaluate the performance of multiple streaming clients using the same network concurrently, tests can be executed with different amounts of concurrent clients. The amount of clients will always be a power of two; one, two, four, eight, and so on. The results of these tests are discussed in Section 7.2.4.

The test video is based on Big Buck Bunny [Ble13], with an aspect ratio of 16:9. It consists of the first 180 seconds of the source content, segmented into one second segments, described by a manifest that uses the live profile, set up for VOD. It has twenty adaptations, ranging from an average bit-rate of approximately 4.7 Mbps and a resolution of 1080p, to an average bit-rate of approximately 47 Kbps and a resolution of 360p.

The HAS streaming client uses an ABR algorithm to ensure a high QoE for the user when streaming, as explained in Chapter 3. By limiting the bandwidth at the network simulator, ns-3 in this case, the client should measure a throughput not higher than the bandwidth limit, and the ABR algorithm should decide on lower bit-rate adaptations. Table 7.1 lists four chosen adaptations, all with a different average bit-rate, as a target for the ABR algorithm. The shaper will be set up with this average bit-rate value as the bandwidth limit. Using this method of

shaping the network, loss will be introduced naturally whenever the client exceeds the bandwidth. Latency will be set at the arbitrary chosen value of 30 ms, without any jitter. Section 7.2.1 details these results.

resolution	average bit-rate
1080p	4.7 Mbps
720p	1.7 Mbps
480p	621 Kbps
360p	425 Kbps

Table 7.1: The average bit-rate of different adaptation of the test video.

One of the parameters of the network simulator is the size of the queue for buffered packets. A queue of length zero will have every packet be subject to the network shaper, every packet that goes over the predefined limit will be dropped. A larger queue will have packets buffered and sent out by the scheduler, any packet that goes over the limit has a chance to be saved by being enqueued and sent whenever possible [Mik10; ns-22]. Since a connection goes over multiple middleboxes, that all have buffering capabilities, it is decided that tests are run with a queue length of 25 packets. Another parameter is the qdisc class, which changes the behavior of the queue of the network simulator. Section 7.2.6 discusses the results to see if there is any impact by changing the queue behavior. As stated in Section 6.3, the UDP receive size buffer will be set to approximately 2.5 MB, to ensure the QUIC implementations are not limited by this factor.

The streaming client, goDASH, will run using the same parameters most of the time. A simple ABR algorithm is chosen, named *conventional*, which measures the goodput and chooses the closest adaptation without exceeding the measurement. Other algorithms are available, such as the *average* algorithm, that will calculate the average goodput value for the complete streaming session. The initial buffer size is set to two seconds and the maximum buffer size is set to thirty seconds, as these are the values used by dash.js. Furthermore, the client is set to stream the full duration seconds of the test video. Section 7.2.2 details the results of comparing the *conventional* algorithm to the *average* algorithm. Different methods of streaming logs and how they affect the performance of an application are discussed in Section 7.2.3. The streaming of logs can be turned on, turned off, or logs can be sent in batches of different sizes.

To test the impact of background traffic, Section 7.2.5 discusses the results of tests performed with and without iPerf3 running in the background. The iPerf3 client and server send data as fast as possible, over a TCP connection, with cubic as their congestion control.

A summary of all tests and which parameters are used can be found in Table 7.2.

Impact of	# clients	log streaming method	ABR	background traffic	queue behavior	bandwidth	latency/jitter/loss
Bandwidth (7.2.1)	1	none	conventional	none	FIFO	425 Kbps, 621 Kbps, 1.7 Mbps, 4.7 Mbps	30ms/0ms/0%
ABR (7.2.2)	——”——	——”——	conventional, average	——”——	——”——	——”——	——”——
Log Streaming (7.2.3)	——”——	none, individual, batched (10), batched (100)	conventional	——”——	——”——	——”——	——”——
Multiple Clients (7.2.4)	1,2,4,8	batched (100)	——”——	——”——	——”——	——”——	——”——
Background Traffic (7.2.5)	——”——	——”——	——”——	TCP cubic	——”——	——”——	——”——
Queue Behavior (7.2.6)	——”——	——”——	——”——	none	FIFO, CoDel	——”——	——”——

Table 7.2: A summary of the tests and their parameters. The number of clients decides how many are run concurrently during the test. A client will be set up with the log streaming client and ABR parameters. Background traffic decides which connections are run in the background during a test, that try to congest the network. The network simulator is set up with the queue behavior, bandwidth, latency, jitter, and loss parameters.

7.1.2 Metrics

During the tests, qlog data is generated by both the client and server. Both the client and server export QUIC qlog, on top of that the client also exports ABR qlog. Metrics are gathered from the transport layer and the higher-level application layer. For comparing the performance between clients, the goodput measure and delivery time showed most potential, therefore these are put in the spotlight for most tests.

Furthermore, buffering events can be used to show when and how often buffering occurs. Another event shows which adaptation is chosen for the current event, and by referring to the network event that captures the request of a segment, the size of the segment can be inferred.

7.2 Tests and Results

This section discusses test results and compares them to one another. The goal is both to gain knowledge about how the client and server are impacted by different parameters.

7.2.1 Impact of Bandwidth

This section will discuss tests to determine the impact of different network bandwidths on the performance of the streaming client. A single client is used, with the conventional ABR algorithm, and no events are streamed to the analysis service. There will be no other applications competing for the bandwidth. The network simulator will be running with the FIFO qdisc class, with 30 ms of latency, no jitter, and no loss, whilst the bandwidth is set to different targets for every test: 4.7 Mbps, 1.7 Mbps, 621 Kbps, and 425 Kbps.

The hypothesis is that the goodput measurement will approach the bandwidth. There should not be any buffering, as the delivery time remains under the one-second mark, because this is the length of a segment.

Table 7.3 shows the results of this test. It should be noted that the average delivery time never does not exceed the one-second mark, meaning all segments can be delivered in time, and the buffer can start to fill. Besides that, the average goodput measurements also do not exceed the bandwidth. Staying under these limits is the goal of the ABR algorithm. None of the tests show any buffering, the buffer never runs dry. When inspecting the actual data sent, however, in Figure 7.1, the discrepancy between the goodput and the actual data sent on the wire becomes obvious. The average bit-rate advertised by the manifest is not representative of all segments. While this is not causing too many problems in this case, this could become problematic with multiple clients streaming more data than expected, especially at lower bandwidths.

bandwidth	delivery time	goodput
4.7 Mbps	887 ms	4,106.751 Kbps
1.7 Mbps	858 ms	1,491.160 Kbps
621 Kbps	938 ms	546.850 Kbps
425 Kbps	921 ms	376.452 Kbps

Table 7.3: Table containing average delivery time and goodput values of tests where a single streaming application is limited by different bandwidth values.

For this test, the server logs were consulted to ensure that no anomalies can be found for this simple test. Figure 7.2 shows the RTT measurements and the congestion window used by the server. The congestion window resembles the expected pattern from a New Reno implementation. It might seem like there is an unusual amount of jitter in the RTT measurements, however,

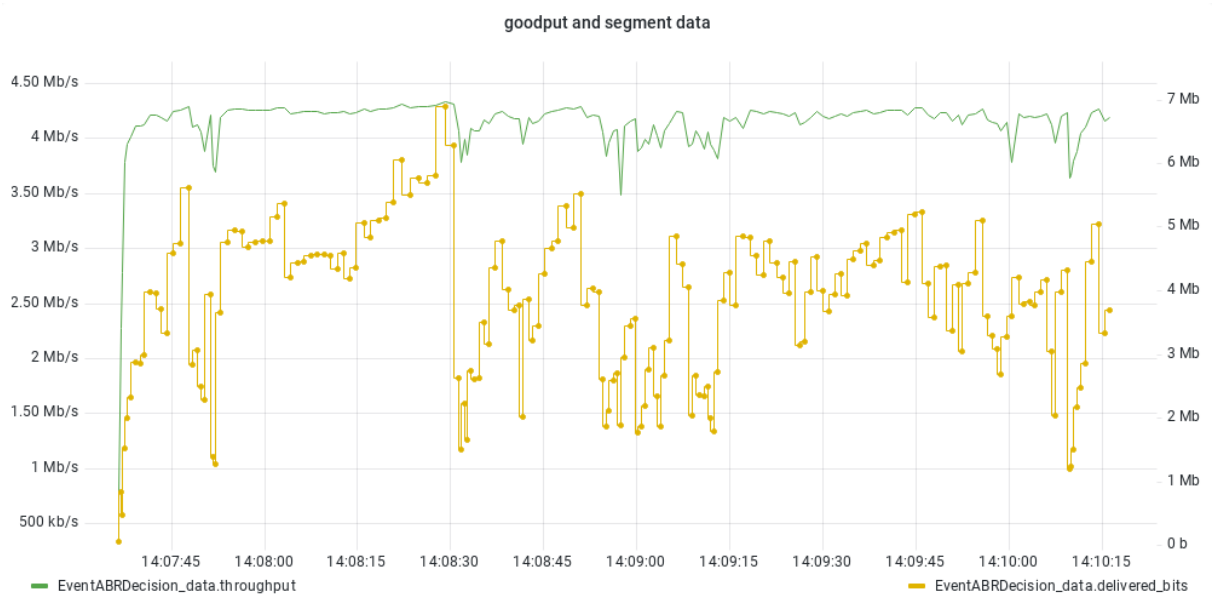
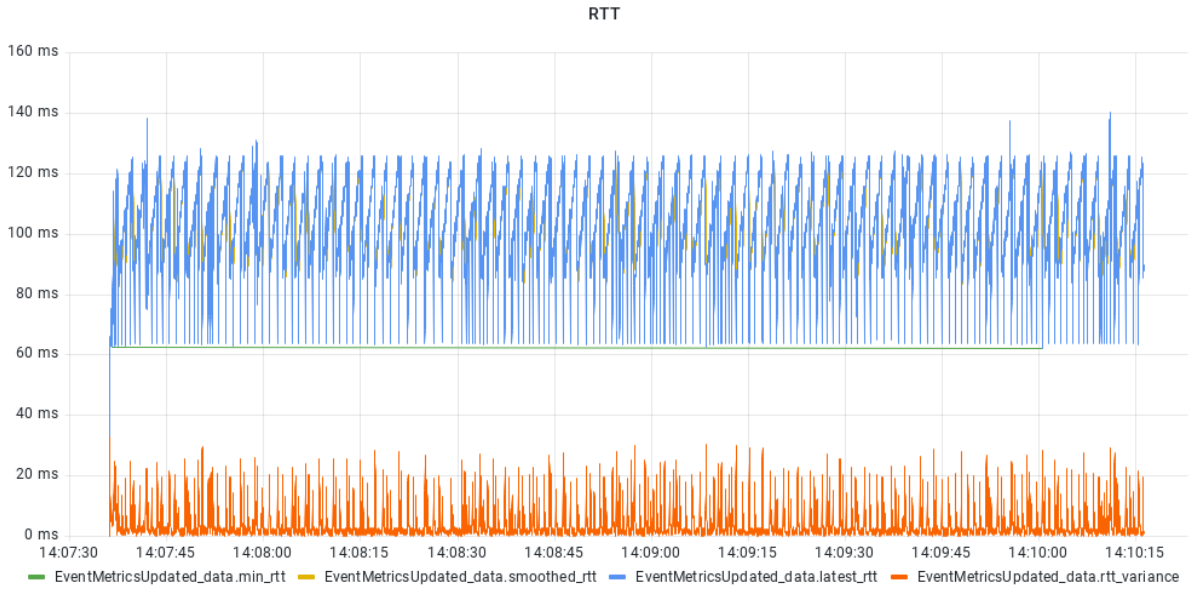
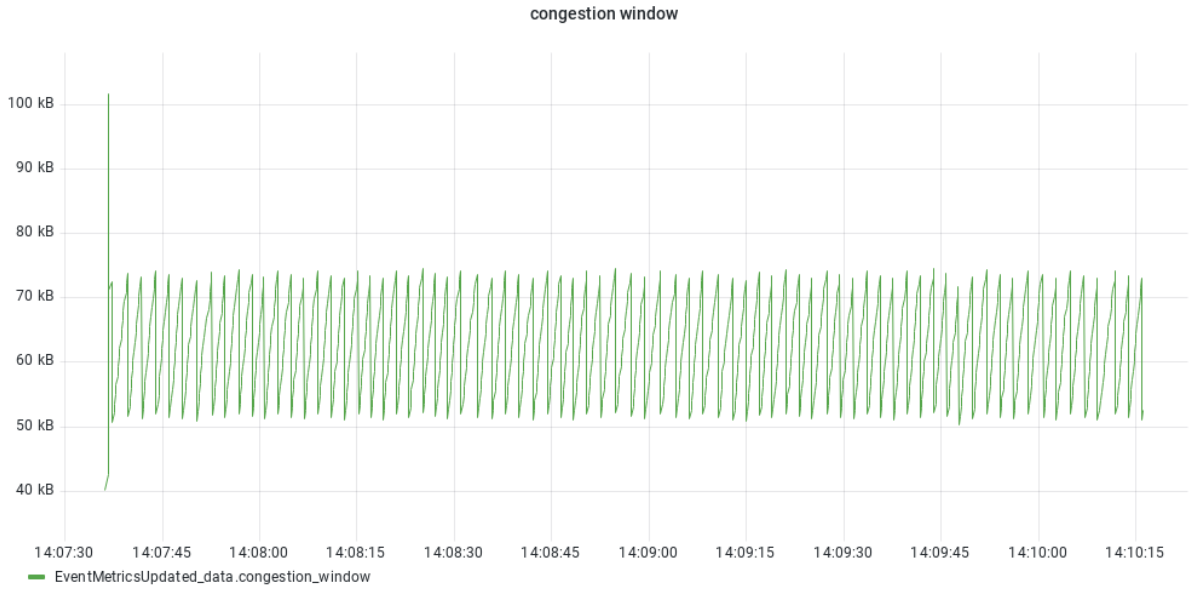


Figure 7.1: A comparison between the measured goodput and the actual data sent to the client, during the test with a bandwidth of 4.7 Mbps. The green line represents the measured goodput, whilst the yellow line represents the amount of data transferred to the client that resulted in the goodput measurement.

this seems to be caused by the interaction with the goDASH client, this RTT pattern is always present at the server.



(a) RTT, lower is better



(b) congestion window

Figure 7.2: A selection of transport layer protocols, collected at the server during the test with a bandwidth of 4.7 Mbps. (a) The RTT measurements, blue shows the latest measurement, yellow shows the smoothed RTT calculated using the latest measurements, red shows the variance between the smoothed and the latest measurement, green is the minimum RTT value. The minimum remains at zero until a first measurement is made. (b) The congestion window.

7.2.2 Impact of ABR

This section will discuss tests to determine the impact of different ABR algorithms on the performance of the streaming client. A single client is used, with the conventional or average ABR algorithm, and no events are streamed to the analysis service. There will be no other applications competing for the bandwidth. The network simulator will be running with the FIFO qdisc class, with 30 ms of latency, no jitter, and no loss, whilst the bandwidth is set to different targets for every test: 4.7 Mbps, 1.7 Mbps, 621 Kbps, and 425 Kbps.

The hypothesis is that both algorithms will have very similar performance, because of their similar behavior. Where the conventional algorithm reacts immediately to a throughput change, the average algorithm will make the same changes, but less intense. Both algorithms will approach the bandwidth with their goodput measurements. There should not be any buffering, as the delivery time remains under the one-second mark, because this is the length of a segment.

	conventional		average	
bandwidth	delivery time	goodput	delivery time	goodput
4.7 Mbps	887 ms	4,106.751 Kbps	878 ms	4,106.662 Kbps
1.7 Mbps	858 ms	1,491.160 Kbps	860 ms	1,492.742 Kbps
621 Kbps	938 ms	546.850 Kbps	**784 ms**	**537.385 Kbps**
425 Kbps	921 ms	376.452 Kbps	922 ms	375.782 Kbps

Table 7.4: Table containing the comparison between average delivery times and goodput measurements between a streaming client using the conventional and the average algorithm.

The comparison between the conventional and the average algorithm is shown in Table 7.4. The results are as expected, there is almost no difference between delivery times and goodput measurements. Figure 7.3 shows the goodput and delivery time measurements with a bandwidth of 4.7 Mbps. Both of the traces appear to have similar behavior. There is, however, an anomaly during the test with a bandwidth of 621 Kbps, the delivery time is much lower than expected with the average algorithm. Furthermore, with the average algorithm, the test ends approximately 23 seconds earlier because of this. Figure 7.4 shows the goodput and delivery time measurements. Visually, the delivery time is lower, and the goodput seems to be lower at most points as well. By inspecting the logs even further, it is obvious that with the average algorithm, an adaptation with a lower bit-rate is chosen consistently. Whereas the conventional algorithm usually goes for the adaptation with an average bit-rate of 537 Kbps, the average algorithm opts for the adaptation with an average bit-rate of 424 Kbps. When inspecting the actual data sent to the clients, shown in Figure 7.5, it is obvious that the average algorithm chooses lower bit-rate adaptations more often than the conventional algorithm. This only happens a few times during the test with the bandwidth at 4.7 Mbps, but frequently with the test at 621 Kbps.

The algorithms can not exceed the goodput measurement. While measurements can approach 600 Kbps, they can also be as low as 500 Kbps. Averaging out these measurements, as can be seen by the average goodput in Table 7.4, results in a value just around 537 Kbps. The algorithm will choose the lower adaptation because it thinks the 537 Kbps adaptation will congest the network.

(a) goodput, higher is better

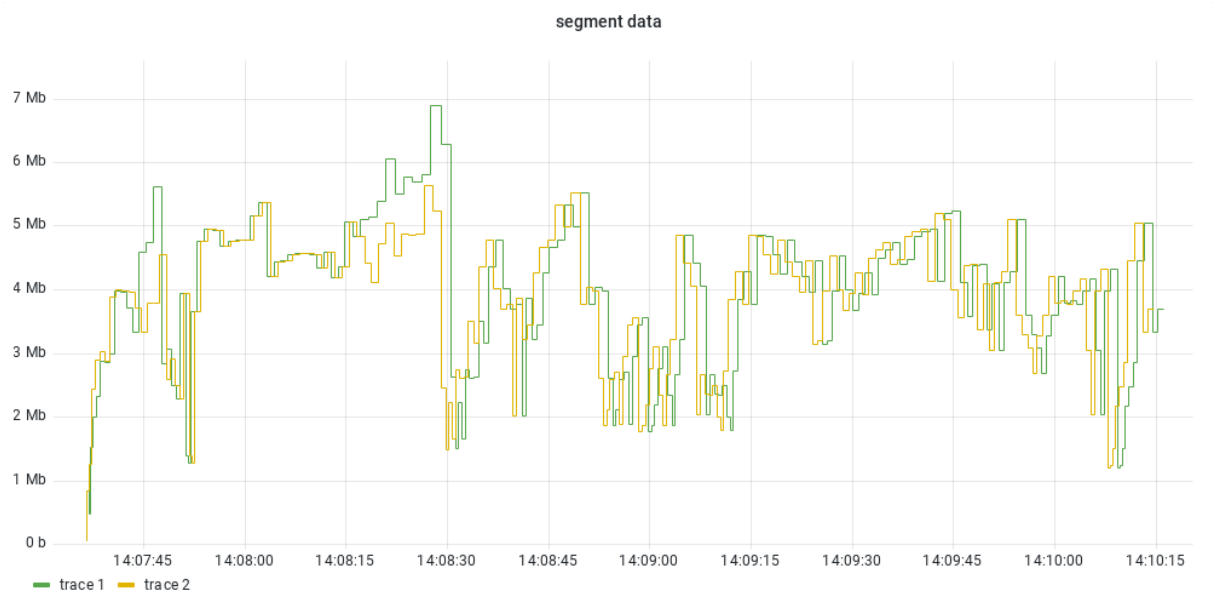
(b) delivery time, lower is better

Figure 7.3: Goodput and delivery time compared between tests with different ABR algorithms, conventional (trace 1) and average (trace 2), with a bandwidth of 4.7 Mbps.

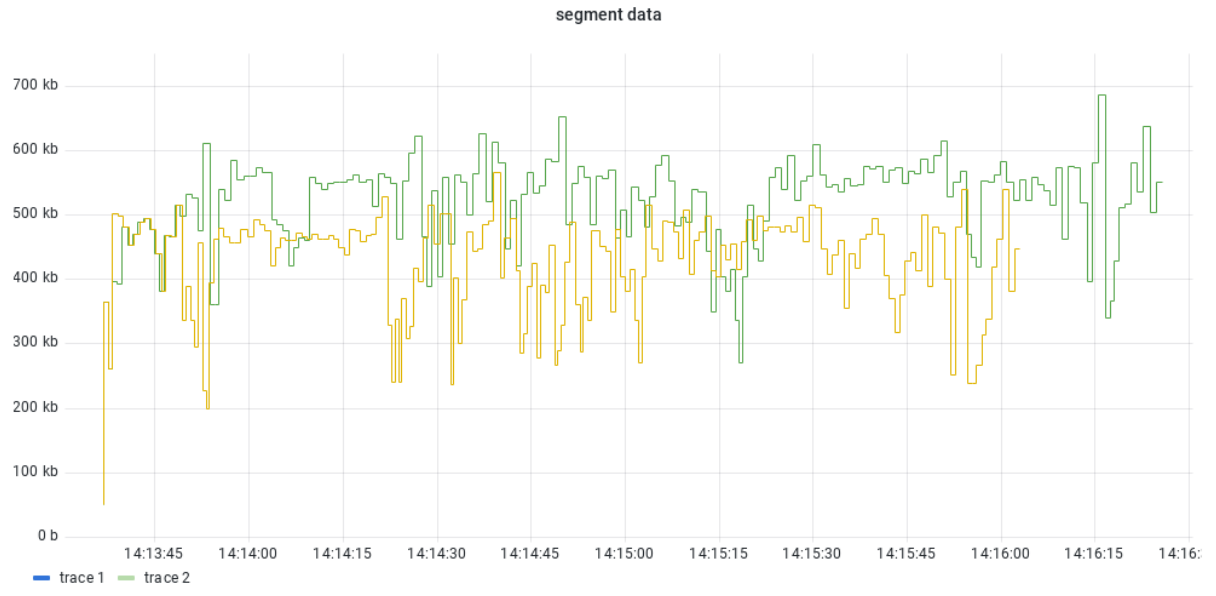
(a) goodput, higher is better

(b) delivery time, lower is better

Figure 7.4: Goodput and delivery time compared between tests with different ABR algorithms, conventional (trace 1) and average (trace 2), with a bandwidth of 621 Kbps. Trace 2 ends approximately 23 seconds earlier than trace 1.



(a) bandwidth at 4.7 Mbps



(b) bandwidth at 621 Kbps

Figure 7.5: Comparison between the segment data of the tests with the conventional (trace 1) and the average (trace 2) ABR algorithm.

7.2.3 Impact of Log Streaming

This section will discuss tests to determine the impact on the performance of the streaming client of different methods of streaming events to the analysis service. A single client is used, with the conventional ABR algorithm, and events are streamed to the analysis service in four ways: none are streamed, all are streamed individually, batches of events are streamed, either in groups of 10 or 100. There will be no other applications competing for the bandwidth. The network simulator will be running with the FIFO qdisc class, with 30 ms of latency, no jitter, and no loss, whilst the bandwidth is set to different targets for every test: 4.7 Mbps, 1.7 Mbps, 621 Kbps, and 425 Kbps.

As a reminder, event logging is done by a routine that is separate from the main streaming loop. This routine receives events, and writes them to the local log file. Then, based on the parameters, the event is either ignored if streaming is turned off, or it is passed onto the piece of the application that provides this streaming functionality. Whenever events are streamed individually, the events are sent out one-by-one, otherwise they are temporarily stored as the batch is being prepared until the batch size has been reached.

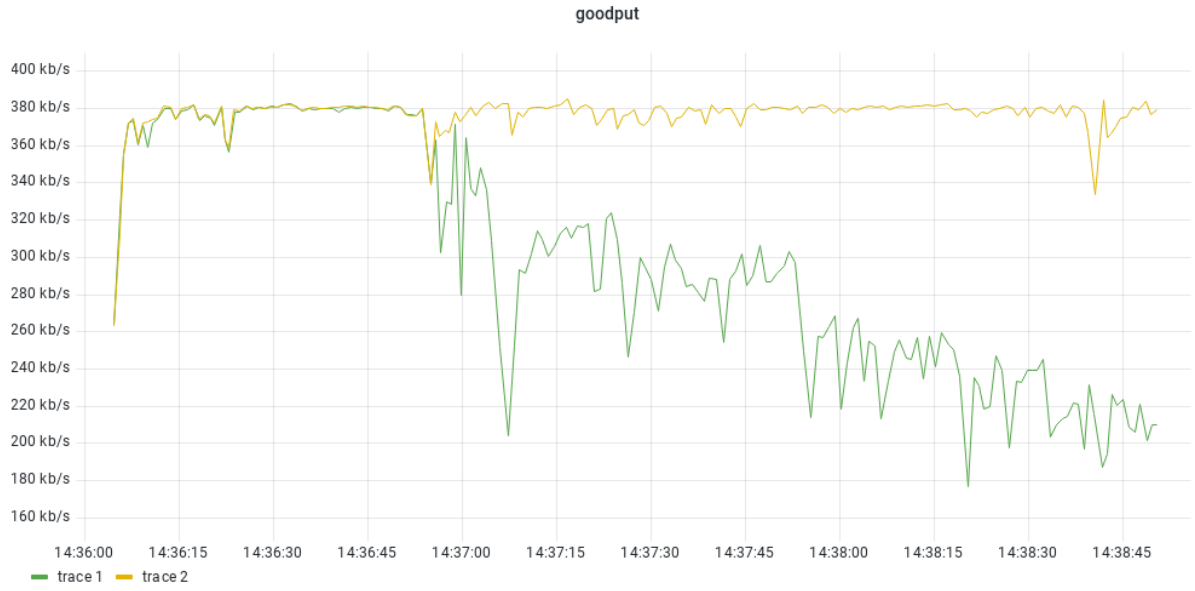
The hypothesis is that the amount of requests will impact the available throughput for the application, with more requests meaning a more negative impact. Meaning that not streaming logs will have the best performance, streaming individual events will have the worst performance. The batched streaming method will have a lesser negative impact, but will be noticeable. It should not be unexpected to see delivery times exceeding the one-second mark, as the many events could result in very high latency on the network, therefore, buffering is possible.

Table 7.5 shows the results of the tests with logging turned off and events streamed individually. In both cases, the ABR algorithm is able to make sure that the delivery time does not exceed the segment length of one second. It is obvious, however, that the goodput is severely and negatively impacted by streaming the logs. In the case of the test with the most bandwidth, the goodput measurement is approximately a ninth of the same test with streaming of logs turned off. Figure 7.6 shows the goodput and delivery time of the tests with the lowest bandwidth, whereas Figure 7.7 shows the same metrics for the highest bandwidth test. The delivery times for both traces in each test do not seem too out of the ordinary. Trace 1 shows in both cases more frequent peaks than trace 2, however, their average stays below the critical segment length. The goodput measurements, however, clearly show when congestion occurs. A lower goodput starts being measured and a lower average bit-rate adaptation is chosen for the following segments. The lower bit-rate allows for the delivery time to stay level. When looking at the buffering events, as is done with Figure 7.8, it can be seen that with a higher bandwidth buffering events occur, but this does not occur with a lower bandwidth. This is most likely caused by the larger segments getting impacted more by the congestion.

	logs on		logs off	
bandwidth	delivery time	goodput	delivery time	goodput
4.7 Mbps	974 ms	462.336 Kbps	887 ms	4,106.751 Kbps
1.7 Mbps	935 ms	454.887 Kbps	858 ms	1,491.160 Kbps
621 Kbps	930 ms	362.609 Kbps	938 ms	546.850 Kbps
425 Kbps	922 ms	299.257 Kbps	921 ms	376.452 Kbps

Table 7.5: Table containing average delivery time and throughput values of tests where a single streaming application is limited by different bandwidth values, and streaming of logs is turned on or off.

Table 7.6 shows the results where events are batched together before sending the batches to



(a) 425 Kbps, goodput

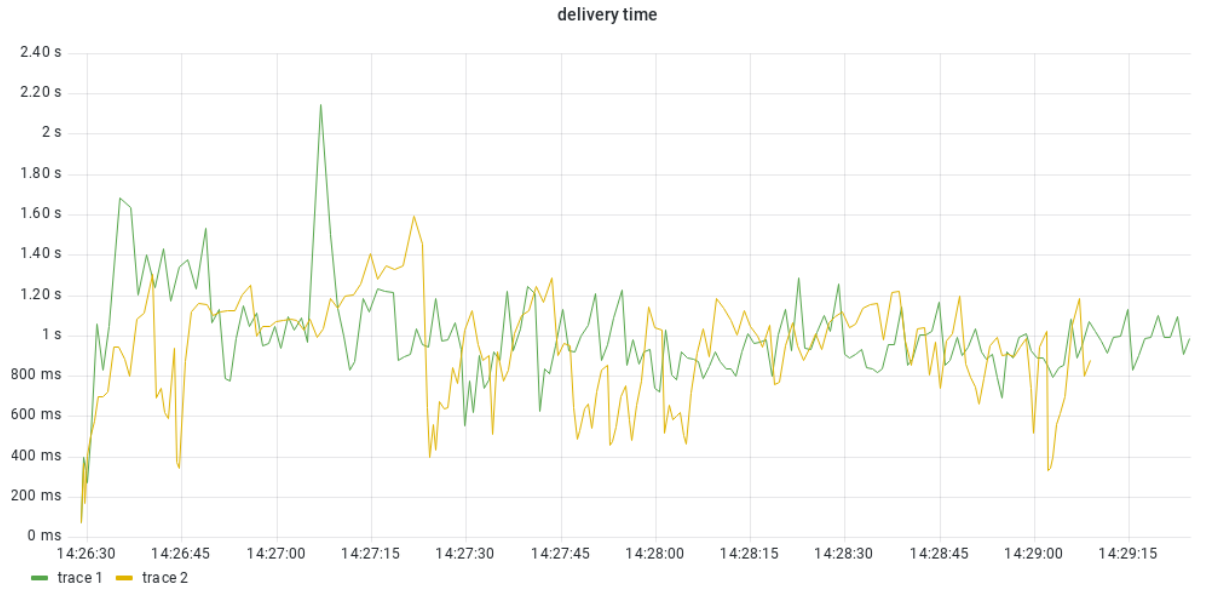
(b) 425 Kbps, delivery time

Figure 7.6: Goodput and delivery time compared between tests with streaming of logs on (trace 1) and off (trace 2), with a maximum bandwidth of 425 Kbps.

	batched logs (10)		batched logs (100)	
bandwidth	delivery time	goodput	delivery time	goodput
4.7 Mbps	924 ms	1,624.363 Kbps	928 ms	3,035.264 Kbps
1.7 Mbps	901 ms	1,261.480 Kbps	861 ms	1,485.830 Kbps
621 Kbps	947 ms	545.524 Kbps	931 ms	545.923 Kbps
425 Kbps	921 ms	375.712 Kbps	921 ms	376.570 Kbps

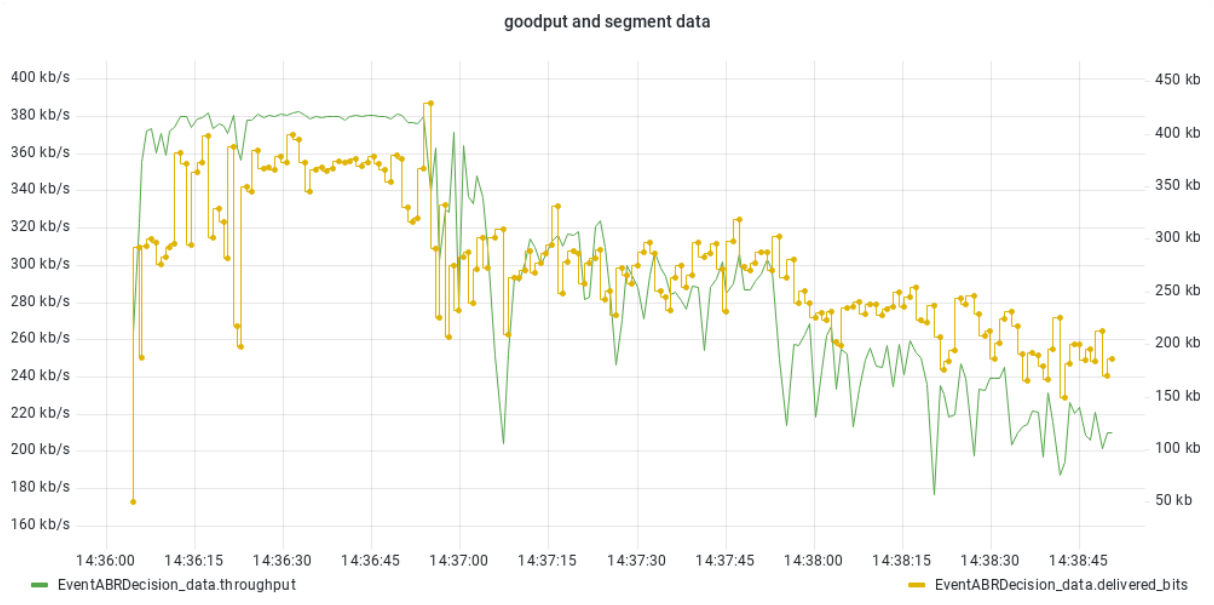
Table 7.6: Table containing average delivery time and throughput values of tests where a single streaming application is limited by different bandwidth values, and events are streamed in batches of ten and one hundred events.

(a) goodput, higher is better

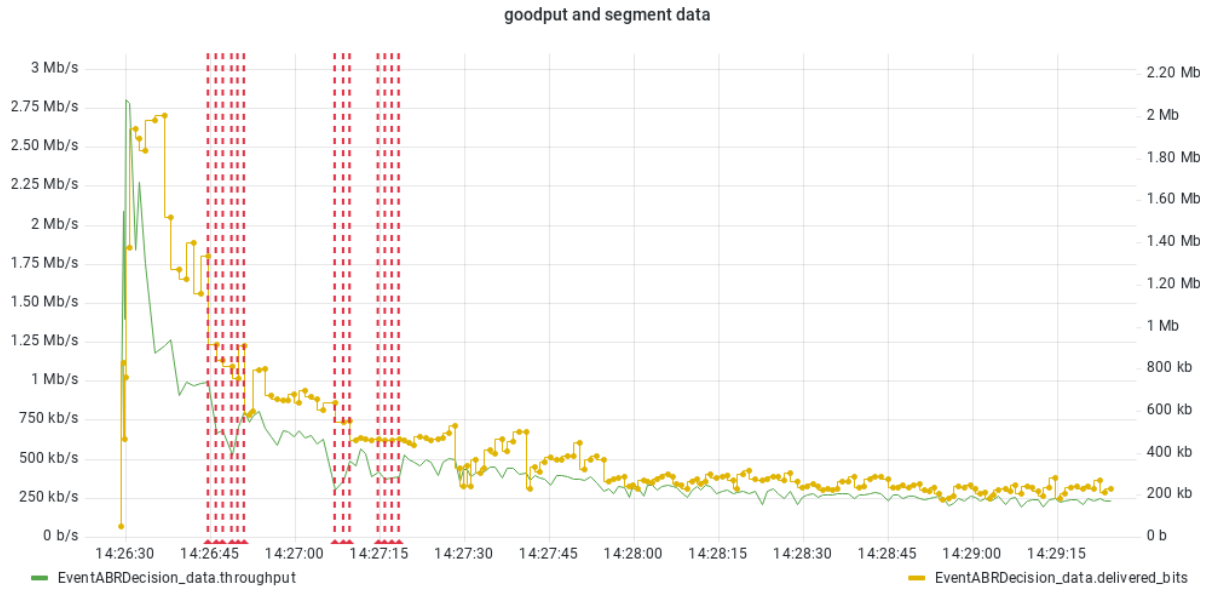


(b) delivery time, lower is better

Figure 7.7: Goodput and delivery time compared between tests with streaming of logs on (trace 1) and off (trace 2), with a maximum bandwidth of 4.7 Mbps.



(a) bandwidth at 425 Kbps



(b) bandwidth at 4.7 Mbps

Figure 7.8: The goodput measurement and the actual data sent to the client. The red vertical lines indicate buffering events.

the analysis service. While the delivery time stays stagnant, the goodput rises as the size of the batches rise. There is, however, a diminishing return, since the goodput can not exceed the bandwidth limit. With a batch size of ten, the client is able to reach an equal goodput to not streaming the logs for the tests up to 621 Kbps, and the 1.7 Mbps test reaches approximately 85% of the target goodput. With a larger batch size of one hundred, the 1.7 Mbps test also reaches their target goodput, the tests with a lower maximum bandwidth see no change in goodput. This means that those tests delay sending their events, with no actual benefit for increasing the goodput.

7.2.4 Impact of Multiple Clients

This section will discuss tests to determine the impact of multiple streaming clients concurrently on their performance. A single client is used as a reference, then tests are run with multiple concurrent clients: 2, 4, and 8 clients. Every client uses the same parameters: the conventional ABR algorithm, and events are streamed to the analysis service in batches of 100 events. There will be no other applications competing for the bandwidth. The network simulator will be running with the FIFO qdisc class, with 30 ms of latency, no jitter, and no loss, whilst the bandwidth is set to different targets for every test: 4.7 Mbps, 1.7 Mbps, 621 Kbps, and 425 Kbps.

The hypothesis is that the amount of clients will impact the performance, as the bandwidth has to be shared, a lower goodput will be measured, and a lower bit-rate adaptation will be chosen. Clients will take their fair share of the bandwidth, meaning the bandwidth will approximately be divided evenly between all clients. There should be no buffering events, since the congestion control should be able to notice the other clients being active on the network, and ensure a lower experienced throughput.

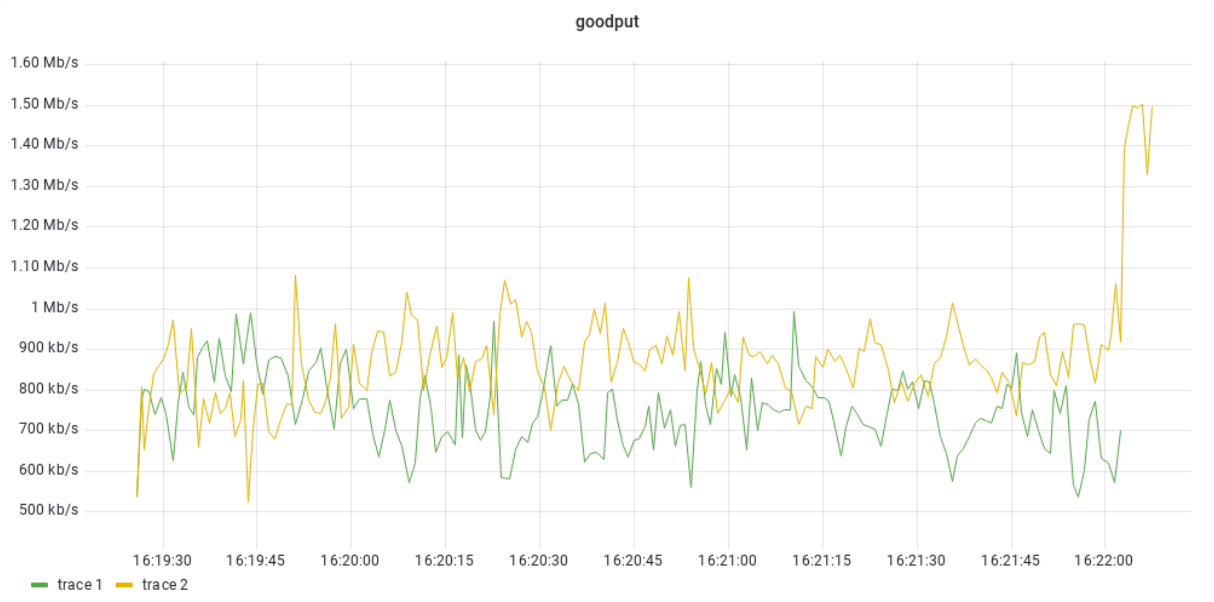
The first test is run with two clients, their delivery times and goodput measurements are shown in Table 7.7. The clients seem to share the bandwidth quite fairly, however, one of the clients is always able to achieve a higher average goodput, this can result in a higher bit-rate adaptation for that client. Shown in Figure 7.10 are the adaptation choices made by the two clients, of the test with a bandwidth of 4.7 Mbps. One of the clients is able to choose a much higher bit-rate adaptation, however, towards the end both clients approach an equilibrium.

Figure 7.9 depicts the goodput and delivery time of the test with a bandwidth of 1.7 Mbps. This test shows the greatest difference between the end time of the clients, it results in one of the client being able to take a much larger share of the bandwidth for the latest segments, up to 1.5 Mbps.

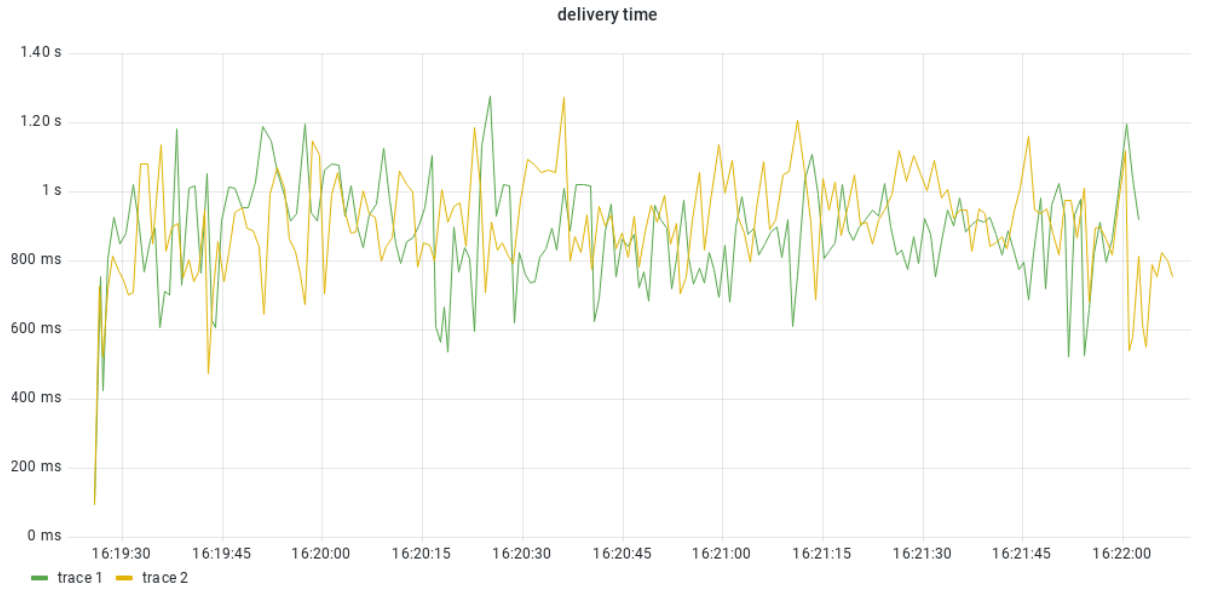
	client 1		client 2		
bandwidth	delivery time	goodput	delivery time	goodput	difference
4.7 Mbps	860 ms	2,106.363 Kbps	855 ms	1,859.652 Kbps	1 s
1.7 Mbps	869 ms	752.363 Kbps	897 ms	879.075 Kbps	5 s
621 Kbps	917 ms	299.949 Kbps	919 ms	300.838 Kbps	1 s
425 Kbps	955 ms	217.032 Kbps	943 ms	201.003 Kbps	2 s

Table 7.7: Table containing the comparison between average delivery times and goodput measurements between two streaming clients from the same streaming session, and the absolute difference in how long it took for the clients to finish.

Table 7.8 compares the aggregated measurements of the two clients to the measurements of a single client. While the delivery times do not seem out of the ordinary, there is a higher overall goodput. Individually, the two clients measure a lower goodput than the single client, and have



(a) goodput, higher is better



(b) delivery time, lower is better

Figure 7.9: Goodput and delivery time compared between tests with two clients, client 1 (trace 1) and client 2 (trace 2), and a bandwidth of 1.7 Mbps.

a lower QoE. This comparison, however, shows that the bandwidth is more optimally used by the two clients.

	1 client		2 clients	
bandwidth	delivery time	goodput	delivery time	goodput
4.7 Mbps	928 ms	**3,035.264 Kbps**	858 ms	**3,966.015 Kbps**
1.7 Mbps	861 ms	**1,485.830 Kbps**	883 ms	**1,631.438 Kbps**
621 Kbps	931 ms	**545.923 Kbps**	918 ms	**600.787 Kbps**
425 Kbps	921 ms	**376.570 Kbps**	949 ms	**418.035 Kbps**

Table 7.8: Table containing the comparison between average delivery times and goodput measurements between a single client and two clients sharing the same bandwidth. For the two clients, the data is aggregated, delivery times are averaged and goodput values are summed up.

With four clients, the same pattern continues. The measurements stay quite level over time, and there is some difference between the achieved goodput values of the clients. Table 7.9 depicts the delivery times and goodput measurements for the four-client test with a bandwidth of 4.7 Mbps, which shows no anomalies.

	delivery time	goodput
client 1	906 ms	1,128.000 Kbps
client 2	916 ms	1,331.165 Kbps
client 3	891 ms	1,044.998 Kbps
client 4	889 ms	1,055.754 Kbps

Table 7.9: Table containing average delivery time and throughput values of four clients who ran concurrently during a streaming session with a bandwidth of 4.7 Mbps.

However, if we compare the aggregated data points of the four clients to the single client, depicted in Table 7.10, the goodput measurements stand out. During certain tests, the average goodput measurement exceeds the bandwidth. Furthermore, the clients suffer from buffering during the session. By measuring the goodput when other clients are less active, a higher goodput is measured. The conventional algorithm immediately chooses a higher bit-rate adaptation, but the network will struggle with delivering it as the other clients become more active. This pattern continues as the amount of clients grows. The average delivery time of a client does not exceed the one-second mark, while the average goodput is roughly equal to the bandwidth divided by all clients, however the sum of the average goodput is higher than the bandwidth. With a greater amount of clients, a greater amount of congestion is noticed, resulting in more buffering, and a lower QoE.

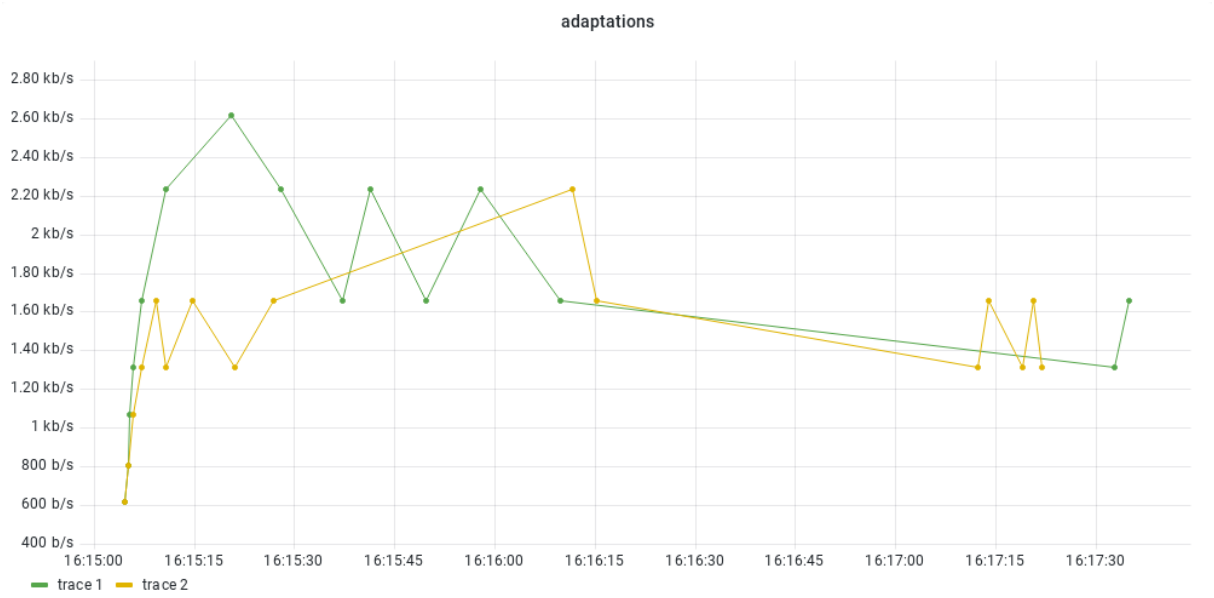


Figure 7.10: The bit-rate choices from two clients that streamed concurrently during the test with a bandwidth of 4.7 Mbps.

	1 client		4 clients	
bandwidth	delivery time	goodput	delivery time	goodput
4.7 Mbps	928 ms	3,035.264 Kbps	901 ms	4,559.917 Kbps
1.7 Mbps	861 ms	1,485.830 Kbps	946 ms	**1,746.120 Kbps**
621 Kbps	931 ms	545.923 Kbps	942 ms	**649.032 Kbps**
425 Kbps	921 ms	376.570 Kbps	970 ms	**486.734 Kbps**

Table 7.10: Table containing the comparison between average delivery times and goodput measurements between a single client and four clients sharing the same bandwidth. For the four clients, the data is aggregated, delivery times are averaged and goodput values are summed up.

7.2.5 Impact of Background Traffic

This section will discuss tests to determine the impact of background traffic on the performance of multiple clients streaming concurrently. A single client is used as a reference, then tests are run with multiple concurrent clients: 2, 4, and 8 clients. Every client uses the same parameters: the conventional ABR algorithm, and events are streamed to the analysis service in batches of 100 events. The background traffic is generated using iPerf3, which sets up a TCP connection with a cubic congestion controller, and continuously sends data over this connection with the aim to saturate the network. The network simulator will be running with the FIFO qdisc class, with 30 ms of latency, no jitter, and no loss, whilst the bandwidth is set to different targets for every test: 4.7 Mbps, 1.7 Mbps, 621 Kbps, and 425 Kbps.

The hypothesis is that the background traffic will reduce the available throughput. However, this is also the case with multiple clients. This test simply adds another subject to the network to compete for the bandwidth, but it should play fair, just like the streaming clients. This time, however, buffering is not written out.

The measurements of a single client, both with and without background traffic, are depicted in Table 7.11. As expected, the goodput measurement is lower with background traffic. There is no buffering taking place at any point within the streaming session, as the lower goodput measurement ensures the delivery time does not get too high. When inspecting the graph visually, shown in Figure 7.11, it is obvious that jitter is present. Based on these results and the results in Section 7.2.4, however, with more clients the QoE will rapidly drop as there is less bandwidth to divide and more unexpected behavior on the network.

	1 client		1 client, background traffic	
bandwidth	delivery time	goodput	delivery time	goodput
4.7 Mbps	928 ms	3,035.264 Kbps	898 ms	2,452.228 Kbps
1.7 Mbps	861 ms	1,485.830 Kbps	909 ms	660.137 Kbps
621 Kbps	931 ms	545.923 Kbps	885 ms	157.975 Kbps
425 Kbps	921 ms	376.570 Kbps	778 ms	69.166 Kbps

Table 7.11: Table containing the comparison between average delivery times and goodput measurements between streaming with and without background traffic.

Figure 7.12 shows the goodput measurements and delivery times of two out of four clients that streamed concurrently. While showing similar behavior, there is obvious jitter present and one client experiences much more buffering than the other. As with the tests in Section 7.2.4, the bandwidth is more optimally used by multiple clients, as can be seen in Table 7.12. This time, however, the bandwidth is not exceeded, whilst buffering still occurs. Background traffic takes up their share of the bandwidth that is not represented in the table.

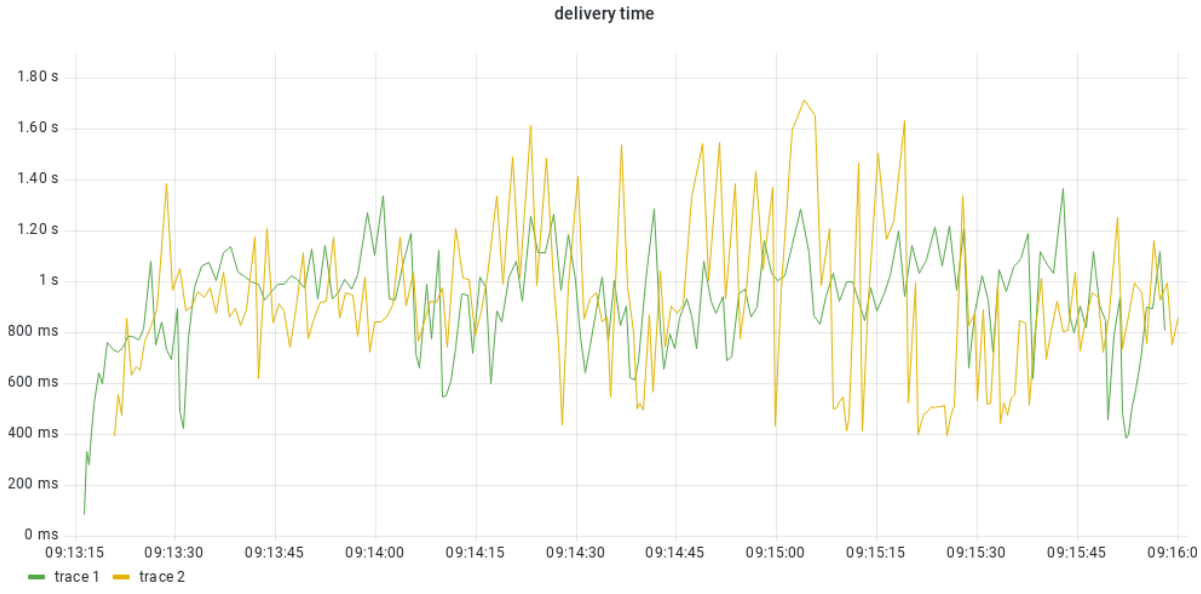
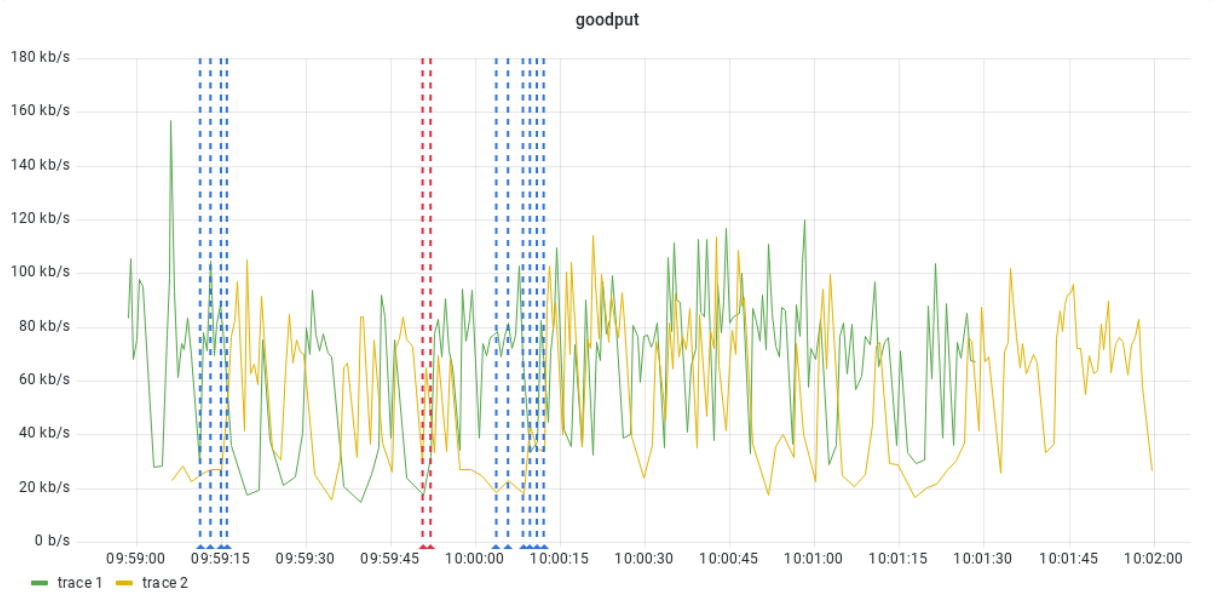


Figure 7.11: The delivery times compared between tests with a single client and with background traffic. A test is run with a bandwidth of 4.7 Mbps (trace 1), the other with a bandwidth of 621 Kbps (trace 2). A lower delivery time is better.

	4 clients		4 clients, background traffic	
bandwidth	delivery time	goodput	delivery time	goodput
4.7 Mbps	901 ms	4,559.917 Kbps	903 ms	**3,998.263 Kbps**
1.7 Mbps	946 ms	1,746.120 Kbps	927 ms	**1.480,323 Kbps**
621 Kbps	942 ms	649.032 Kbps	937 ms	**358.226 Kbps**
425 Kbps	970 ms	486.734 Kbps	889 ms	**268.194 Kbps**

Table 7.12: Table containing the comparison between average delivery times and goodput measurements between a single client and four clients sharing the same bandwidth. For the four clients, the data is aggregated, delivery times are averaged and goodput values are summed up.



(a) goodput, higher is better; red vertical lines show buffering events of trace 1, blue vertical lines show buffering events of trace 2

(b) delivery time, lower is better

Figure 7.12: The goodput and delivery time of two out of four streaming clients that streamed concurrently with a bandwidth of 425 Kbps.

7.2.6 Impact of Queue Behavior

This section will discuss tests to determine the impact of the network simulator queue behavior on the performance of multiple clients streaming concurrently. A single client is used as a reference, then tests are run with multiple concurrent clients: 2, 4, and 8 clients. Every client uses the same parameters: the conventional ABR algorithm, and events are streamed to the analysis service in batches of 100 events. The background traffic is generated using iPerf3, which sets up a TCP connection with a cubic congestion controller, and continuously sends data over this connection with the aim to saturate the network. The network simulator will be running with the FIFO or CoDel qdisc class, with 30 ms of latency, no jitter, and no loss, whilst the bandwidth is set to different targets for every test: 4.7 Mbps, 1.7 Mbps, 621 Kbps, and 425 Kbps.

The hypothesis is that the CoDel qdisc class should handle a congested network better, ensuring that traffic is let through more fairly, since the FIFO queue is prone to bufferbloat [The22a]. Resulting in less buffering and jitter, but similar latencies and goodput.

	FIFO		CoDel	
	delivery time	goodput	delivery time	goodput
logs off	938 ms	546.859 Kbps	961 ms	547.744 Kbps
logs batched (10)	948 ms	545.408 Kbps	940 ms	544.342 Kbps
logs batched (100)	932 ms	545.985 Kbps	945 ms	546.918 Kbps
logs	930 ms	362.609 Kbps	962 ms	351.513 Kbps

Table 7.13: Table containing average delivery time and throughput values of tests with a bandwidth of 621 Kbps, comparing the FIFO qdisc class to the CoDel qdisc class.

An initial test compares the difference in performance between using the FIFO and CoDel qdisc classes, with all methods of log streaming. When comparing these results, which are summarized in Table 7.13, no significant difference is found. The behavior of the packet queue seems to behave in similar ways when only a single client is saturating the network.

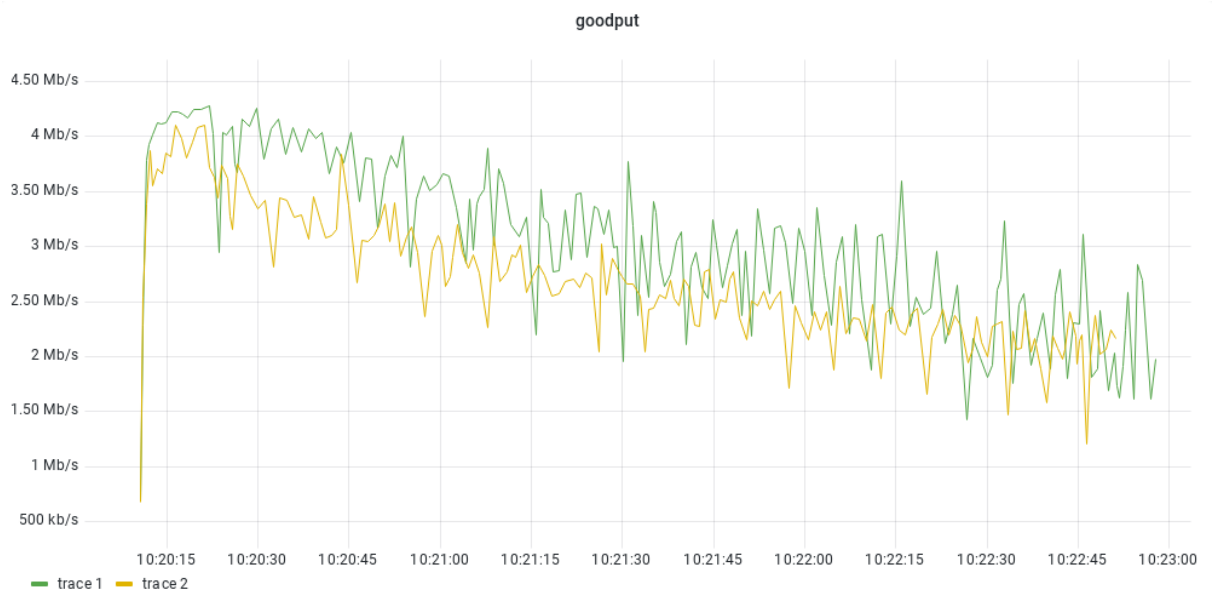
Figures 7.13 and 7.14 show results with a bandwidth of 4.7 Mbps, with individual event logging and logging in batches of 100, respectively. Again, there are a few seconds of difference between the end of the tests. The goodput with CoDel seems to be more jittery when logging individual events, but the goodput in both scenarios show similar averages. With batched logging, however, the FIFO qdisc class is able to reach higher goodput overall, over 3.04 Mbps compared to 2.69 Mbps with CoDel. The same can be observed when logs are not streamed to the analysis service, with the FIFO qdisc class reaching 4.10 Mbps, whereas with CoDel an average goodput of 3.91 Mbps is achieved. Figure 7.15 zoom in to the start of the test where events are streamed individually, with a CoDel qdisc class, and a bandwidth of 4.7 Mbps. As the client starts to stream content, events are generated and start taking up bandwidth, resulting in lower a goodput. With a decreasing share of the bandwidth for the client, buffering events start to occur, until an adaptation is chosen where the bit-rate of the content is low enough.

The test with two concurrently streaming clients shows the impact of CoDel on the scheduling of the network. Table 7.14 compares the aggregate data of the test with two concurrently streaming clients. With FIFO, there is a higher overall goodput measured. In Figure 7.16, however, the results of the test with a bandwidth of 1.7 Mbps show that goodput measurements are very similar, hinting at that the network is shared fairly when using CoDel. The sizes of the segments that are streamed to the clients are equal, and both clients choose an adaptation once, and never have to switch. This pattern can be found in every test with two clients when using the CoDel qdisc class, except for some client making a few minor adjustments in adaptation

(a) goodput, higher is better; red vertical lines show buffering events

(b) delivery time, lower is better

Figure 7.13: The goodput and delivery time of tests with the FIFO qdisc class (trace 1) and the CoDel qdisc class (trace 2), with events being streamed individually, and a bandwidth of 4.7 Mbps. The test with the CoDel qdisc class took longer to finish, the average delivery time was higher. This test also shows more jittery goodput behavior.



(a) goodput, higher is better; red vertical lines show buffering events

(b) delivery time, lower is better

Figure 7.14: The goodput and delivery time of tests with the FIFO qdisc class (trace 1) and the CoDel qdisc class (trace 2), with events being streamed in batches of 100, and a bandwidth of 4.7 Mbps. The test with the CoDel qdisc class took less time to finish, the average delivery time was lower.

choice at the start, but an equilibrium is quickly reached.

	2 clients, FIFO		2 clients, CoDel	
bandwidth	delivery time	goodput	delivery time	goodput
4.7 Mbps	858 ms	3,966.015 Kbps	854 ms	3,604.210 Kbps
1.7 Mbps	883 ms	1,631.438 Kbps	798 ms	1,536.532 Kbps
621 Kbps	918 ms	600.787 Kbps	922 ms	590.154 Kbps
425 Kbps	949 ms	418.035 Kbps	918 ms	394.359 Kbps

Table 7.14: Table containing the comparison between average delivery times and goodput measurements between two concurrently streaming clients, once with the FIFO qdisc class, and once with the CoDel qdisc class. For the two clients, the data is aggregated, delivery times are averaged and goodput values are summed up.

When performing this test with a greater amount of clients, however, the slight differences can start to grow larger. An equilibrium will be reached, but with a less fair distribution of the available bandwidth. Figure 7.17 shows the goodput compared between a few different clients from the test with eight clients, with a bandwidth of 621 Kbps and using the CoDel qdisc class. Client 0 and client 1 show very similar behavior, resulting in an average goodput of over 100 Kbps. However, with the bandwidth only being 621 Kbps, it is not sustainable for all clients to receive data at this rate. That is why all other clients have lower goodput measurements, as can be seen by comparing client 0 with client 2. No buffering occurs during this test, as the clients are still playing reasonably fair with each other. Most other tests with multiple clients, however, show a more fair distribution of the bandwidth. The clients act as expected, and avoid congesting the network

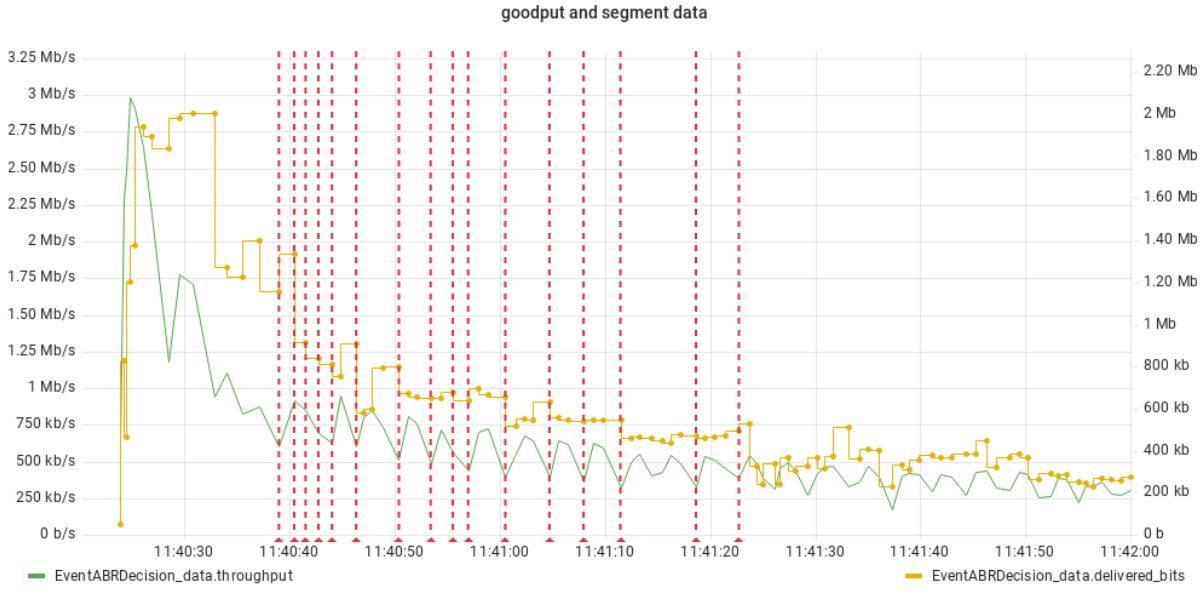
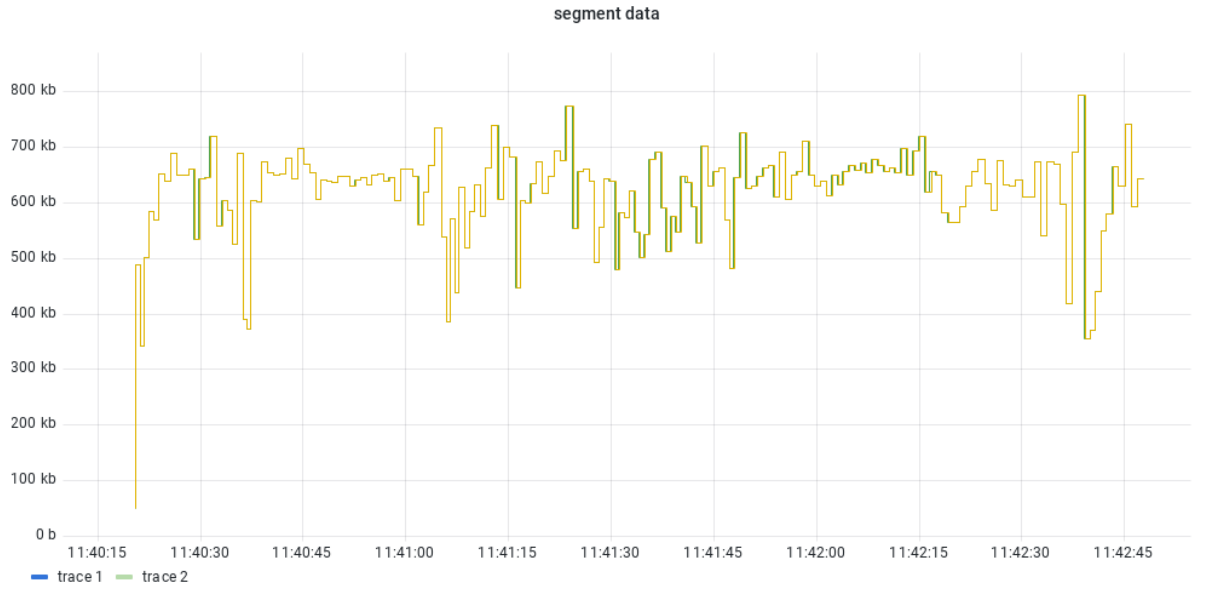


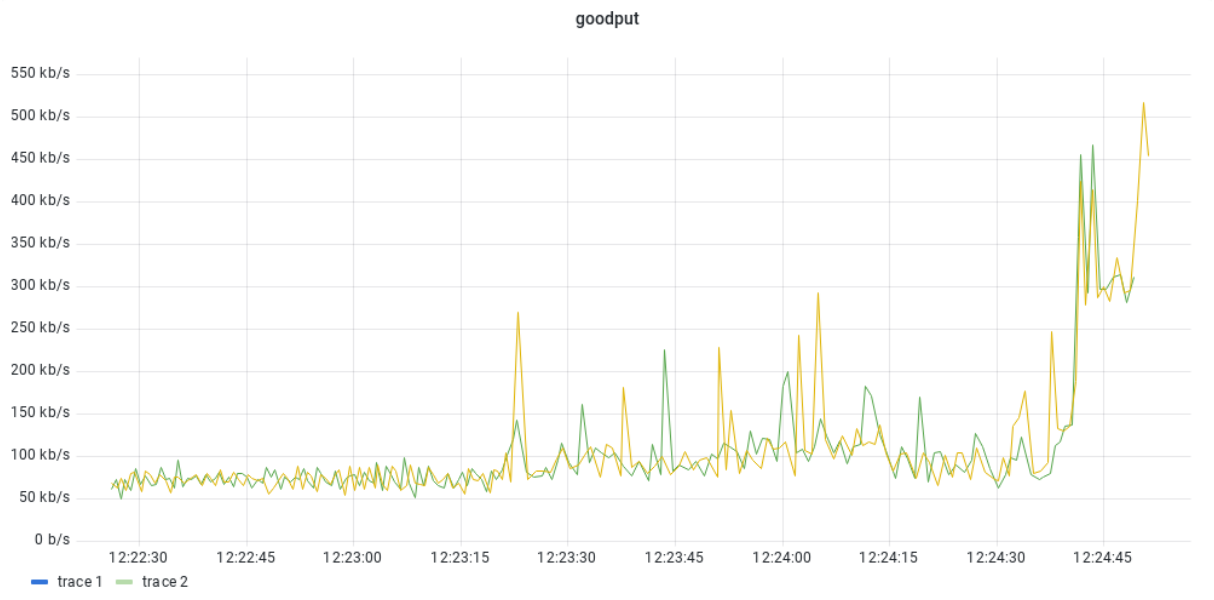
Figure 7.15: Partial results of the test with the CoDel qdisc class, and a bandwidth of 4.7 Mbps, with the clients streaming events individually. The green line shows the goodput measurement. The height of the yellow bars show the amount of bits per segment, while the length shows the time it took for the segment to arrive, these are used to calculate the goodput. The red vertical lines indicate a buffering event.

(a) goodput, higher is better

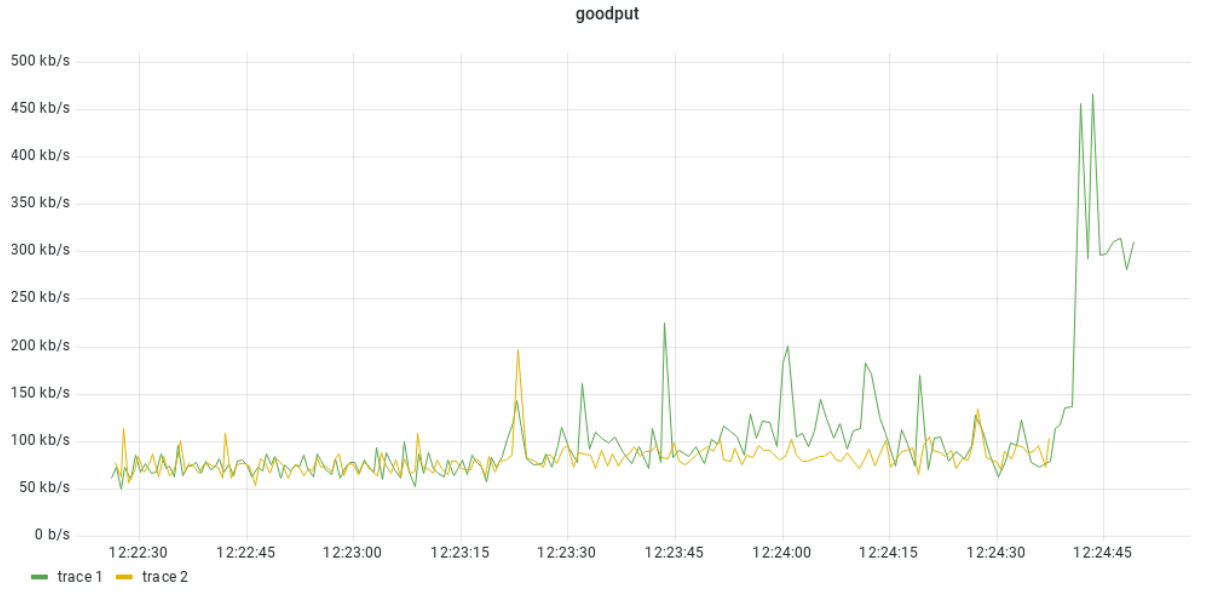


(b) segment data

Figure 7.16: Goodput measurements and segment data sent to two concurrently streaming clients. The segment data shows the size of the segments in the y-axis, the width of the bar shows the time needed to deliver the segment.

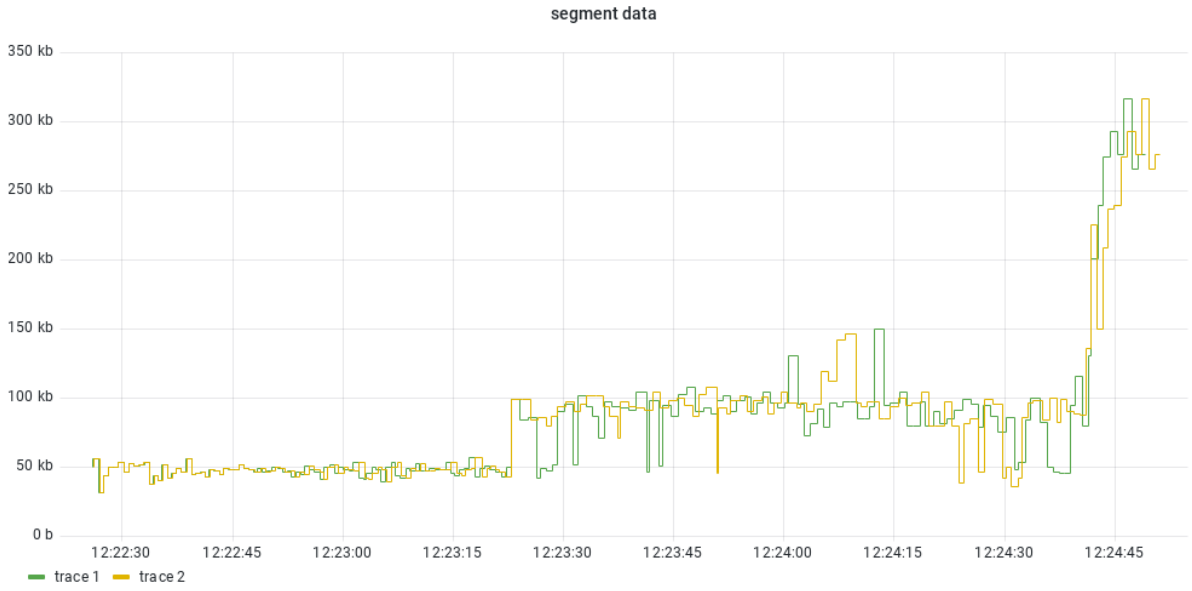


(a) goodput, compared between client 0 and client 1, higher is better

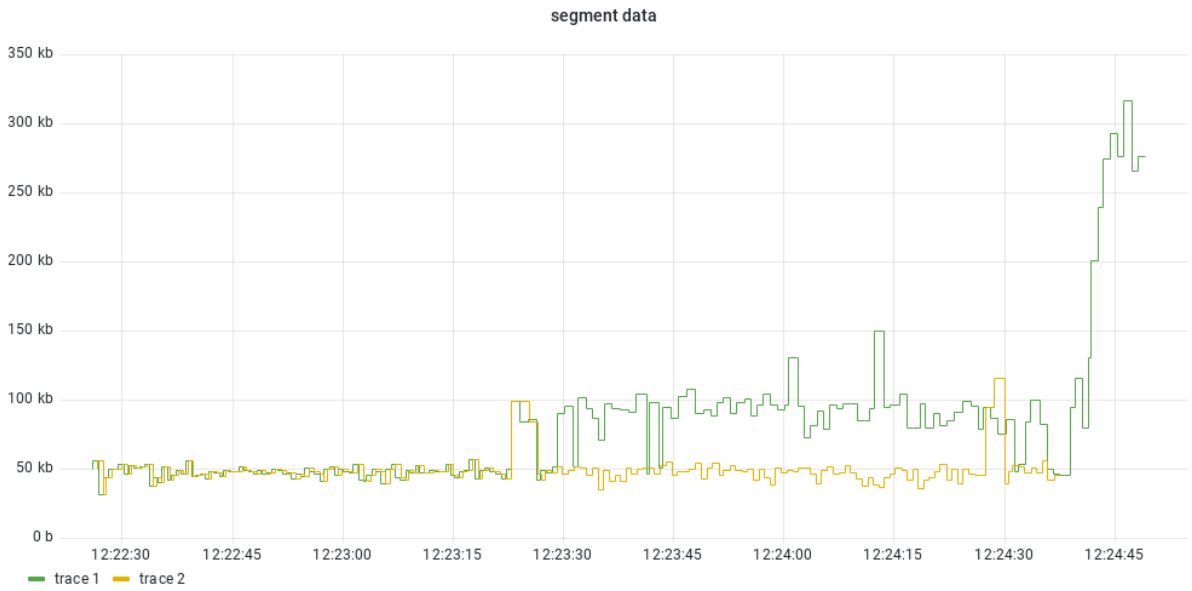


(b) goodput, compared between client 0 and client 2, higher is better

Figure 7.17: Goodput measurements compared between clients during the test with eight concurrently streaming clients, with a bandwidth of 621 Kbps, using the CoDel qdisc class. (a) Some clients measure equal goodput values, (b) others measure lower values.



(a) segment data, compared between client 0 and client 1



(b) segment data, compared between client 0 and client 2

Figure 7.18: The segment sizes compared between clients during the test with eight concurrently streaming clients, with a bandwidth of 621 Kbps, using the CoDel qdisc class. The segment data shows the size of the segments in the y-axis, the width of the bar shows the time needed to deliver the segment. (a) Some clients some the same behavior, resulted from getting the same bandwidth share, (b) other clients show different behavior, since they measure different goodput values, therefore choosing different adaptations.

7.3 Analysis Service

Throughout the evaluations, the analysis service was a valuable asset, that does come with some flaws. Since absolute timestamps are used, it is not trivial to find the correct time range at all times. With little being preprepared and being an unguided experience, creating a concise dashboard that shows the most important data points in the correct way is a challenge every time. With a lower number of clients to compare, it was not difficult to create a dashboard to facilitate this. When the number of clients increased, however, it became inconvenient to create a dashboard to compare all clients. Grafana lacks the ability to dynamically change the dashboard according to the number of clients.

Furthermore, while the GraphQL plugin for Grafana works, the user interface is not really user-friendly. The queries are not checked at the frontend, which is done by many GraphQL front-ends. In order to write correct queries, it was most convenient to use an external front-end to write the queries, before importing them into Grafana. Besides that, the plugin does not support all data types and can not access all data within the query result. Queries might need to be rewritten in order to work correctly with the plugin. Besides that, the way the GraphQL queries work with interfaces, abstract types, is suboptimal for the qlog events. Common attributes are queried on all events, and then it can be specified which attributes to extract from specific subtypes. This results in a trace returning the common data for every event, which in the case of transport layer logs can be tens of thousands of events, even those who are not needed, and can result in Grafana needing some noticeable processing time. The lag experienced is not only caused by large traces, but also by the inefficient querying method by Grafana. Every panel on a dashboard executes its own query, even if panels use equivalent data.

Chapter 8

Conclusion

The thesis confirms that it is possible to perform real-time analysis of an adaptive video stream when using next-generation protocols without negatively impacting the performance of the application. However, this is reliant on the network having the required bandwidth to support log transfer without interfering with the video stream. Section 7.2.3 explains how different methods of streaming logs affect streaming performance. The fewer network requests that must be made and the less overhead that is avoided, the more likely a streaming client will be able to reach its target bit-rate for the current network environment. The method of streaming logs should be tuned towards the network for both the best streaming performance, but also making sure this data stream stays as real-time as possible.

Several tests on streaming clients can be performed using the framework and analysis service. Influences that multiple concurrent clients exert on one another can be inspected by overlaying the data points from multiple clients and making a comparison. Section 7.2.4 shows how this method is used on tests with numerous clients. The differences in behavior of a streaming client can be observed if internal streaming parameters are changed; for example, the ABR algorithm, as is done in Section 7.2.2. Furthermore, outside forces, such as the network environment, also changes the behavior of a streaming client. Tests are run with different network settings in Section 7.2.6, and show how the clients and network react. Another outside force is background traffic, which can also have an influence on the streaming experience, as shown in Section 7.2.5. While the analysis service cannot analyze these outside forces/external factors, the network is treated as a black box, it is still possible to analyze the behavior of the streaming client and infer the behavior of outside forces.

While it was possible to achieve the real-time aspect of the thesis, throughout the evaluation phase the importance of this aspect became questionable. The realization that there are two types of analysis, as stated in Chapter 6, uncovered a misalignment between what this kind of analysis service is best suited for, and the goal of the thesis. It is possible to detect certain events, such as congestion, but the analysis service is not able to act on those events. The client detects these events as well, however, it does act immediately; for example, congestion is noticed by the transport protocol and the congestion window is reduced, and the ABR algorithm might notice a reduced goodput and choose a lower bit-rate adaptation. Knowledge can still be gained by using the analysis service, but the real-time aspect does not benefit this process. Following an evaluation, it was discovered that comparisons provide the most insight into client behavior. A good comparison necessitates the availability of corresponding data points; this is most easily accomplished when both subjects to be compared have all data points available.

Another problem is the velocity at which logs arrive at the analysis service, and the speed at which they can be processed. The human analyzer is a bottleneck, as there are numerous data

points continuously streaming into the analysis service, while the analyzer can only observe a selection of those data points at a time. It is possible to construct a dashboard with panels highlighting the most important data points, but it is unlikely that a single human analyzer can keep an eye on all of them. During the analysis process, multiple dashboards would be used, which all highlighted a different set of data points from different layers. At times, a single panel of the dashboard would be focused, in order to zoom in or change the parameters to improve the visualization.

The visualizations, while straight-forward to create and modify, are difficult to perfect. A lot of experience is required in order to know what data to visualize in what manner, which turned out to be more challenging than expected. The many visualizations available in the dashboards, however, show great potential, and there is still much room for improvement.

8.1 Future Work

Intrusion Detection Systems (IDS), in the field of cybersecurity, have the goal of analyzing network traffic and events, detecting anomalies, and performing an action based on what was detected. This is a process that is similar to the analysis service process. There is active research being conducted to integrate Machine Learning techniques (ML) within IDS solutions in order to detect anomalies more effectively. The ability of ML to generalize methods means that previously unknown anomalies can be detected by training an ML model on known anomalous behavior [LL19]. Anomalies can be detected in a real-time stream of temporal data, regardless of the type of data. The stream will be analyzed in real-time, and if the behavior of the data appears to change at any point in time, this will be classified as an anomaly [Ahm+17].

The integration of ML techniques would be an improvement over the current analysis service. The analysis service would relieve the burden of analyzing every single data point by detecting anomalies in the data and marking the most interesting areas to inspect. This will also make real-time data more feasible, as data can be marked as interesting during ingress. The human analyzer has fewer data points to observe, or it can be automated. If the analysis process is automated, it will be possible to react to events in the same way that an ABR algorithm does. Changes in CDN routing may be possible to reduce latency.

At the moment, the analysis focuses on a single layer, the streaming application. While there are dashboards that allow for comparisons between different layers, these did not prove to gain as many valuable insights as expected. Instead, the other layers served as sanity checks to make sure everything was working as expected. Chapter 2 discusses the cross-layer aspect of protocols, this analysis service can be used to facilitate analyzing an application that uses this principle. The numerous data points can be correlated to gain insights into how the cross-layer principle affects the application.

The analysis can also be layered by aggregating data from various nodes. Only logs from the client and, optionally, the server are currently aggregated. While the network is regarded as a black box, it could be argued that a large corporation, such as Google, which controls the client, server, and CDN, has access to data from the middleboxes. The different connections between the different middleboxes can also be analyzed and are potentially valuable sources of knowledge. In practice, there is no single connection between the client and the server, but there is a connection between the server and the CDN, as well as between CDN nodes, and between a CDN edge server and the client. A poor streaming experience might be caused by only one of these connections.

8.2 Reflection

I, Mike Vandersanden, have had various educational experiences throughout writing this thesis. A number of technologies and principles discussed in the thesis have been addressed in courses or been explored in my bachelor thesis. Having the opportunity to gain more in-depth knowledge on these and other topics, such as video streaming and network protocols, is invaluable. At times, it is challenging to take the acquired knowledge and form a coherent and clear text that my peers would comprehend. With the help of my mentors and some friends, however, I think the thesis is able to accomplish this. Being both quite comprehensive, and still understandable for people that lack domain knowledge. The practical aspect of the thesis turned out to be different from expected, more testing and analyzing, less actual implementation. It also included more of an exploration of different technologies, and turned into a proof-of-concept of their collaboration. In the end, while not being what I would have imagined, the end result satisfies me, it is wonderful to see all things learned coming together.

Throughout the implementation phase, the decision was made to not base my work upon existing solutions, specifically using qvis for visualization of qlog. It is difficult to prove whether this was the right call. After spending some time with members of the research department, however, and seeing how their research was developing on related subjects, this seemed like the right call at that time. What sounded like small improvements that they tried to implement in their own visualizations took a substantial amount of time. Recreating their simple visualizations was no challenge with Grafana. Later, during the evaluation phase, the downsides of Grafana started to show, but I do not think another solution would have been considerably better.

Based on my experiences with writing a thesis for my bachelor degree, I tried some new methods of trying to improve my thesis writing, and most importantly, not postpone the writing. I kept a list of all papers and other sources that seemed relevant and showed potential. The list was categorized and had a little summary for every source. Furthermore, throughout the implementation and evaluation phase, I would try to take notes and include them in the thesis document. This resulted in a lot of content already being in the document when starting the writing phase, the sentences just had to be written. During the evaluation phase, however, this tactic was not used optimally. Various problems kept postponing what I thought to be the actual evaluation phase, instead of realizing that these problems could be turned into parts of that phase. Without keeping track of what problems exactly came up, and the data gathered during that time, some things got lost. This was eventually turned into Section 6.3, the preliminary testing. I encountered an example of Miller's Law: Exceptions prove the rule – and wreck the budget. It took longer than expected to ensure a certain level of quality, but I failed to adapt to the depleted time budget.

If I were to start over, I would take better advantage of the facilities offered by my mentor, specifically the ability to get a desk in their research room. During the summer, the "QUIC @ EDM – summer edition" initiative allowed all researches, students, and student workers, to collaborate on their work. This invaluable experience made it possible to finish the thesis in time, at a higher level of quality I would have been able to achieve on my own.

Finally, throughout this experience, I learned that it is important to listen to others, but not follow their words blindly. While reading numerous publications and conversing with people, my own ideas would get challenged, and at times this resulted in rejection of these ideas. The further I got through the year, however, I realized that my ideas are as valuable as those of others, especially as long as there is proof or expertise saying otherwise. Towards the end, I would start listening to people and taking their ideas with a grain of salt, as I should have from the start.

Bibliography

[14] *Information technology — Dynamic adaptive streaming over HTTP (DASH)*. Standard. Geneva, CH: International Organization for Standardization, May 2014.

[20] *Information technology — Multimedia application format (MPEG-A) — Part 19: Common media application format (CMAF) for segmented media*. Standard. Geneva, CH: International Organization for Standardization, Mar. 2020.

[AB18] Sevket Arisu and Ali C. Begen. "Quickly Starting Media Streams Using QUIC". In: *Proceedings of the 23rd Packet Video Workshop*. PV '18. Amsterdam, Netherlands: Association for Computing Machinery, 2018, pp. 1–6. ISBN: 9781450357739. DOI: 10.1145/3210424.3210426.

[Adh+12] Vijay Kumar Adhikari et al. "Unreeling netflix: Understanding and improving multi-CDN movie delivery". In: *2012 Proceedings IEEE INFOCOM*. 2012, pp. 1620–1628. DOI: 10.1109/INFCOM.2012.6195531.

[Ahm+17] Subutai Ahmad et al. "Unsupervised real-time anomaly detection for streaming data". In: *Neurocomputing* 262 (Nov. 2017), pp. 134–147. DOI: 10.1016/j.neucom.2017.04.070.

[App21] Apple. *Enabling Low-Latency HTTP Live Streaming (HLS)*. 2021. URL: https://developer.apple.com/documentation/http_live_streaming/enabling_low-latency_http_live_streaming_hls (visited on 25/7/2022).

[App22] Apple. *About the Common Media Application Format with HTTP Live Streaming*. 2022. URL: https://developer.apple.com/documentation/http_live_streaming/about_the_common_media_application_format_with_http_live_streaming (visited on 15/2/2022).

[Ban22] Nick Banks. *QUIC at Microsoft - Nick Banks - EPIQ 2021 Keynote 1*. 2022. URL: https://www.youtube.com/watch?v=W8I3bjYn4_0 (visited on 6/8/2022).

[BEW95] R.A. Becker, S.G. Eick, and A.R. Wilks. "Visualizing network data". In: *IEEE Transactions on Visualization and Computer Graphics* 1.1 (1995), pp. 16–28. DOI: 10.1109/2945.468391.

[Bis22] Mike Bishop. *HTTP/3*. RFC 9114. June 2022. DOI: 10.17487/RFC9114.

[Bit19] Bitmovin. *Adaptive Bitrate Streaming*. 2019. URL: https://bitmovin.com/adaptive-streaming/ (visited on 25/7/2022).

[Bit20] Bitmovin. *Video Compression Basics: What is video transcoding, why is it important, and how does it apply to my everyday life?* 2020. URL: https://bitmovin.com/what-is-transcoding/ (visited on 22/7/2022).

[Bit21] Bitmovin. *Video Developer Report*. 2021. URL: https://go.bitmovin.com/video-developer-report.

[Bit22] Bitmovin. *The Definitive Guide to Container File Formats [2022]*. 2022. URL: https://bitmovin.com/container-formats-fun-1/ (visited on 23/7/2022).

[Ble13] Blender. *Big Buck Bunny*. 2013. URL: https://peach.blender.org/ (visited on 4/8/2022).

[Bom22] David Bombal. *The Internet just changed*. 2022. URL: https://www.youtube.com/watch?v=cdb7M37o9sU (visited on 3/8/2022).

[BPT15] Mike Belshe, Roberto Peon, and Martin Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. May 2015. DOI: 10.17487/RFC7540.

[Bra89a] Robert T. Braden. *Requirements for Internet Hosts - Application and Support*. RFC 1123. Oct. 1989. DOI: 10.17487/RFC1123.

[Bra89b] Robert T. Braden. *Requirements for Internet Hosts - Communication Layers*. RFC 1122. Oct. 1989. DOI: 10.17487/RFC1122.

[BRV20] Pierre Bourhis, Juan L. Reutter, and Domagoj Vrgoč. "JSON: Data model and query languages". In: *Information Systems* 89 (2020), p. 101478. ISSN: 0306-4379. DOI: https://doi.org/10.1016/j.is.2019.101478.

[BRZ17] Divyashri Bhat, Amr Rizk, and Michael Zink. "Not so QUIC: A Performance Study of DASH over QUIC". In: *Proceedings of the 27th Workshop on Network and Operating Systems Support for Digital Audio and Video*. NOSSDAV'17. Taipei, Taiwan: Association for Computing Machinery, 2017, pp. 13–18. ISBN: 9781450350037. DOI: 10.1145/3083165.3083175.

[Búc17] Javier Búcar. *D-ITG (Distributed Internet Traffic Generator)*. 2017. URL: https://github.com/jbucar/ditg (visited on 10/8/2022).

[Clo22a] Cloudflare. *Cloudflare Radar*. 2022. URL: https://radar.cloudflare.com/ (visited on 3/8/2022).

[Clo22b] Cloudflare. *Mininet: An Instant Virtual Network on your Laptop (or other PC)*. 2022. URL: http://mininet.org/ (visited on 10/8/2022).

[Clo22c] Cloudflare. *What is a CDN? — How do CDNs work?* 2022. URL: https://www.cloudflare.com/en-gb/learning/cdn/what-is-a-cdn/ (visited on 10/8/2022).

[Cur22a] Luke Curley. *Video Distribution Progression @ Twitch*. 2022. URL: https://docs.google.com/document/d/1OTnJunbpSJchdj8XI3GU9Fo-RUUFBqLO1AhlaKk5Alo/edit (visited on 25/7/2022).

[Cur22b] Luke Curley. *Warp - Segmented Live Media Transport*. Internet-Draft draft-lcurley-warp-01. Work in Progress. Internet Engineering Task Force, July 2022. 15 pp. URL: https://datatracker.ietf.org/doc/draft-lcurley-warp/01/.

[CYC74] Vinton Cerf, Dalal Yogen, and Sunshine Carl. *Specification of Internet Transmission Control Program*. RFC 675. Dec. 1974. DOI: 10.17487/RFC0675.

[dac22a] dacast. *HLS vs. MPEG-DASH: A Live Streaming Protocol Comparison for 2022*. 2022. URL: https://www.dacast.com/blog/mpeg-dash-vs-hls-what-you-should-know/ (visited on 25/7/2022).

[dac22b] dacast. *What is RTMP? The Real-Time Messaging Protocol: What you Need to Know in 2022*. 2022. URL: https://www.dacast.com/blog/rtmp-real-time-messaging-protocol/ (visited on 26/7/2022).

[Das22a] Dash Industry Forum. *DASH-IF DASH Live Source Simulator*. 2022. URL: https://github.com/Dash-Industry-Forum/dash-live-source-simulator (visited on 10/8/2022).

[Das22b] Dash Industry Forum. *dash.js*. 2022. URL: https://github.com/Dash-Industry-Forum/dash.js/ (visited on 10/8/2022).

[De +14] Luca De Cicco et al. "TAPAS: A Tool for RApid Prototyping of Adaptive Streaming Algorithms". In: *Proceedings of the 2014 Workshop on Design, Quality and Deployment of Adaptive Video Streaming*. VideoNext '14. Sydney, Australia: Association for Computing Machinery, 2014, pp. 1–6. ISBN: 9781450332811. DOI: 10.1145/2676652.2676654.

[Dic21a] Cambridge Dictionary. *LIVESTREAM — meaning in the Cambridge English Dictionary*. 2021. URL: https://dictionary.cambridge.org/dictionary/english/livestream (visited on 19/10/2021).

[Dic21b] Cambridge Dictionary. *STREAMING — meaning in the Cambridge English Dictionary*. 2021. URL: https://dictionary.cambridge.org/dictionary/english/streaming (visited on 14/10/2021).

[Dic21c] Cambridge Dictionary. *VIDEO-ON-DEMAND — meaning in the Cambridge English Dictionary*. 2021. URL: https://dictionary.cambridge.org/dictionary/english/video-on-demand (visited on 14/10/2021).

[Dic22] Cambridge Dictionary. *ANALYSIS — meaning in the Cambridge English Dictionary*. 2022. URL: https://dictionary.cambridge.org/dictionary/english/analysis (visited on 13/8/2022).

[Dri22] Michael Driscoll. *The Illustrated QUIC Connection*. 2022. URL: https://quic.xargs.org/ (visited on 8/8/2022).

[Dug+22] Jon Dugan et al. *iPerf - The ultimate speed test tool for TCP, UDP and SCTP*. 2022. URL: https://iperf.fr/ (visited on 11/8/2022).

[Ecm17] Ecma International. *ECMA-404: The JSON data interchange syntax*. ECMA ECMA-404. Ecma International, Dec. 2017. URL: https://www.ecma-international.org/publications-and-standards/standards/ecma-404/.

[Elz+14] Stef van den Elzen et al. "Dynamic Network Visualization withExtended Massive Sequence Views". In: *IEEE Transactions on Visualization and Computer Graphics* 20.8 (2014), pp. 1087–1099. DOI: 10.1109/TVCG.2013.263.

[FF13] Daniela Florescu and Ghislain Fourny. "JSONiq: The History of a Query Language". In: *IEEE Internet Computing* 17.5 (2013), pp. 86–90. DOI: 10.1109/MIC.2013.97.

[Fie+97] Roy T. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2068. Jan. 1997. DOI: 10.17487/RFC2068.

[FJT02] Thomas Funkhouser, Jean-Marc Jot, and Nicolas Tsingos. ""Sounds Good to Me!" Computational Sound for Graphics, Virtual Reality, and Interactive Systems". In: *SIGGRAPH 2002 Course Notes* (2002). URL: https://www.cs.princeton.edu/~funk/course02.pdf.

[FNR22a] Roy T. Fielding, Mark Nottingham, and Julian Reschke. *HTTP Caching*. RFC 9111. June 2022. DOI: 10.17487/RFC9111.

[FNR22b] Roy T. Fielding, Mark Nottingham, and Julian Reschke. *HTTP Semantics*. RFC 9110. June 2022. DOI: 10.17487/RFC9110.

[FNR22c] Roy T. Fielding, Mark Nottingham, and Julian Reschke. *HTTP/1.1*. RFC 9112. June 2022. DOI: 10.17487/RFC9112.

[Fou] The GraphQL Foundation. *GraphQL — A query language for your API*. URL: https://graphql.org/ (visited on 19/10/2021).

[Fre+96] Ron Frederick et al. *RTP: A Transport Protocol for Real-Time Applications*. RFC 1889. Jan. 1996. DOI: 10.17487/RFC1889.

[Get11] Jim Gettys. "Bufferbloat: Dark buffers in the internet". In: *IEEE Internet Computing* 15.3 (2011), pp. 96–96.

[Goe+17] Utkarsh Goel et al. "Domain-Sharding for Faster HTTP/2 in Lossy Cellular Networks". In: *CoRR* abs/1707.05836 (2017). arXiv: 1707.05836. URL: http://arxiv.org/abs/1707.05836.

[Hem11] Stephen Hemminger. *NetEm - Network Emulator*. 2011. URL: https://man.archlinux.org/man/tc-netem.8.en (visited on 10/8/2022).

[Her+20] Joris Herbots et al. "Cross-Layer Metrics Sharing for QUICker Video Streaming". In: *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '20. Barcelona, Spain: Association for Computing Machinery, 2020, pp. 542–543. ISBN: 9781450379489. DOI: 10.1145/3386367.3431901.

[Hoo+16] Jeroen van der Hooft et al. "HTTP/2-Based Adaptive Streaming of HEVC Video Over 4G/LTE Networks". In: *IEEE Communications Letters* 20.11 (2016), pp. 2177–2180. DOI: 10.1109/LCOMM.2016.2601087.

[Hoo21] Hootsuite. *Digital Trends Q4 Update*. Statshot Q4. Hootsuite, Oct. 2021.

[HSA17] Md. Faisal Murad Hossain, Mahasweta Sarkar, and Syed Hassan Ahmed. "Quality of Experience for video streaming: A contemporary survey". In: *2017 13th International Wireless Communications and Mobile Computing Conference (IWCMC)*. 2017, pp. 80–84. DOI: 10.1109/IWCMC.2017.7986266.

[IET22] IETF. *IETF Mail Archive: MoQ*. 2022. URL: https://mailarchive.ietf.org/arch/browse/moq/ (visited on 6/8/2022).

[Int17] International Telecommunication Union. *Parametric bitstream-based quality assessment of progressive download and adaptive audiovisual streaming services over reliable transport*. Recommendation P.1203. International Telecommunication Union, Oct. 2017. 22 pp. URL: https://www.itu.int/rec/T-REC-P.1203.

[Int22] Twitch Interactive. *Company — Twitch.tv*. 2022. URL: https://www.twitch.tv/p/en/company/ (visited on 25/7/2022).

[IT21] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. May 2021. DOI: 10.17487/RFC9000.

[ITE12] ITEC. *Datasets — ITEC – Dynamic Adaptive Streaming over HTTP*. 2012. URL: https://dash.itec.aau.at/dash-dataset/ (visited on 10/8/2022).

[Kal+17] Mark Kalman et al. *Introducing LHLS Media Streaming*. 2017. URL: https://medium.com/@periscopecode/introducing-lhls-media-streaming-eb6212948bef (visited on 25/7/2022).

[KBF22] Charles 'Buck' Krasic, Mike Bishop, and Alan Frindell. *QPACK: Field Compression for HTTP/3*. RFC 9204. June 2022. DOI: 10.17487/RFC9204.

[KR17] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach, 7th Edition*. Pearson, 2017.

[Kra16] Vlad Krasnov. *HPACK: the silent killer (feature) of HTTP/2*. 2016. URL: https://blog.cloudflare.com/hpack-the-silent-killer-feature-of-http-2/ (visited on 4/8/2022).

[Lab] Grafana Labs. *Grafana: The open observability platform — Grafana Labs*. URL: https://grafana.com/ (visited on 19/10/2021).

[LL19] Hongyu Liu and Bo Lang. "Machine Learning and Deep Learning Methods for Intrusion Detection Systems: A Survey". In: *Applied Sciences* 9.20 (Oct. 2019), p. 4396. DOI: 10.3390/app9204396.

[Mar+18] Robin Marx et al. "Towards QUIC Debuggability". In: *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. EPIQ'18. Heraklion, Greece: Association for Computing Machinery, 2018, pp. 1–7. ISBN: 9781450360821. DOI: 10.1145/3284850.3284851.

[Mar+20a] Robin Marx et al. "Debugging QUIC and HTTP/3 with qlog and qvis". In: *Proceedings of the Applied Networking Research Workshop*. 2020, pp. 58–66.

[Mar+20b] Robin Marx et al. "Same Standards, Different Decisions: A Study of QUIC and HTTP/3 Implementation Diversity". In: *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. EPIQ '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 14–20. ISBN: 9781450380478. DOI: 10.1145/3405796.3405828.

[Mar21a] Robin Marx. *HTTP/3 From A To Z: Core Concepts (Part 1)*. Aug. 2021. URL: https://www.smashingmagazine.com/2021/08/http3-core-concepts-part1/ (visited on 13/10/2021).

[Mar21b] Robin Marx. *HTTP/3: Performance Improvements (Part 2)*. Aug. 2021. URL: https://www.smashingmagazine.com/2021/08/http3-performance-improvements-part2/ (visited on 13/10/2021).

[Mat20] Matroska. *What is Matroska?* 2020. URL: https://www.matroska.org/what_is_matroska.html (visited on 23/7/2022).

[Mc21] Mozilla and individual contributors. *Evolution of HTTP - HTTP — MDN*. Oct. 2021. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP (visited on 19/10/2021).

[MH21] Robin Marx and Joris Herbots. "Merge Those Metrics: Towards Holistic (Protocol) Logging". In: 2021. URL: https://www.iab.org/wp-content/IAB-uploads/2021/09/MergeThoseMetrics_Marx_Jul2021.pdf.

[Mik10] Mikrotik Wiki. *Manual:Queue Size*. 2010. URL: https://wiki.mikrotik.com/wiki/Manual:Queue_Size (visited on 9/8/2022).

[MNS22a] Robin Marx, Luca Niccolini, and Marten Seemann. *HTTP/3 and QPACK qlog event definitions*. Internet-Draft draft-ietf-quic-qlog-h3-events-01. Work in Progress. Internet Engineering Task Force, Mar. 2022. 25 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-quic-qlog-h3-events/01/.

[MNS22b] Robin Marx, Luca Niccolini, and Marten Seemann. *Main logging schema for qlog*. Internet-Draft draft-ietf-quic-qlog-main-schema-02. Work in Progress. Internet Engineering Task Force, Mar. 2022. 49 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-quic-qlog-main-schema/02/.

[MNS22c] Robin Marx, Luca Niccolini, and Marten Seemann. *QUIC event definitions for qlog*. Internet-Draft draft-ietf-quic-qlog-quic-events-01. Work in Progress. Internet Engineering Task Force, Mar. 2022. 48 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-quic-qlog-quic-events/01/.

[ndj22] ndjson. *ndjson*. 2022. URL: http://ndjson.org/ (visited on 11/8/2022).

[NFB96] Henrik Nielsen, Roy T. Fielding, and Tim Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.0*. RFC 1945. May 1996. DOI: 10.17487/RFC1945.

[ns-22] ns-3. *ns-3: ns3::PfifoFastQueueDisc Class Reference*. 2022. URL: https://www.nsnam.org/doxygen/classns3_1_1_pfifo_fast_queue_disc.html (visited on 9/8/2022).

[nsn22] nsnam. *ns-3*. 2022. URL: https://www.nsnam.org/ (visited on 10/8/2022).

[ORQ20] John O'Sullivan, Darijo Raca, and Jason J. Quinlan. "Godash 2.0 - The Next Evolution of HAS Evaluation". In: *2020 IEEE 21st International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*. 2020, pp. 185–187. DOI: 10.1109/WoWMoM49955.2020.00043.

[PKS22] Tommy Pauly, Eric Kinnear, and David Schinazi. *An Unreliable Datagram Extension to QUIC*. RFC 9221. Mar. 2022. DOI: 10.17487/RFC9221.

[PM17] Roger Pantos and William May. *HTTP Live Streaming*. RFC 8216. Aug. 2017. DOI: 10.17487/RFC8216.

[Pos80] Jon Postel. *User Datagram Protocol*. RFC 768. Aug. 1980. DOI: 10.17487/RFC0768.

[PR15] Roberto Peon and Herve Ruellan. *HPACK: Header Compression for HTTP/2*. RFC 7541. May 2015. DOI: 10.17487/RFC7541.

[Rac+18] Darijo Raca et al. "Beyond throughput: A 4G LTE dataset with channel and context metrics". In: *Proceedings of the 9th ACM multimedia systems conference*. 2018, pp. 460–465.

[Res18] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446.

[RKM20] Roberto Ramos-Chavez, Theo Karagkioules, and Rufael Mekuria. "A Scalable Load Generation Framework for Evaluation of Video Streaming Workflows in the Cloud". In: *Proceedings of the 11th ACM Multimedia Systems Conference*. MMSys '20.

Istanbul, Turkey: Association for Computing Machinery, 2020, pp. 255–260. ISBN: 9781450368452. DOI: 10.1145/3339825.3394930.

[RLS98] Anup Rao, Rob Lanphier, and Henning Schulzrinne. *Real Time Streaming Protocol (RTSP)*. RFC 2326. Apr. 1998. DOI: 10.17487/RFC2326.

[RM22] Eric Roman and Matt Menke. *NetLog: Chrome's network logging system*. 2022. URL: https://www.chromium.org/developers/design-documents/network-stack/netlog/ (visited on 11/8/2022).

[RMQ20] Darijo Raca, Maëlle Manifacier, and Jason J. Quinlan. "goDASH — GO Accelerated HAS Framework for Rapid Prototyping". In: *2020 Twelfth International Conference on Quality of Multimedia Experience (QoMEX)*. 2020, pp. 1–4. DOI: 10.1109/QoMEX48832.2020.9123103.

[Roc+21] Florentin Rochet et al. "TCPLS: Modern Transport Services with TCP and TLS". In: *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 45–59. ISBN: 9781450390989. DOI: 10.1145/3485983.3494865.

[Rüt+19] Jan Rüth et al. *Blitz-starting QUIC Connections*. 2019. arXiv: 1905.03144 [cs.NI].

[See21] Marten Seemann. *UDP Receive Buffer Size*. 2021. URL: https://github.com/lucas-clemente/quic-go/wiki/UDP-Receive-Buffer-Size (visited on 9/8/2022).

[Set21] De Rosal Igantius Moses Setiadi. "PSNR vs SSIM: imperceptibility quality assessment for image steganography". In: *Multimedia Tools and Applications* 80.6 (2021), pp. 8423–8444.

[Sha22] Keith Shaw. *The OSI model explained and how to easily remember its 7 layers*. 2022. URL: https://www.networkworld.com/article/3239677/the-osi-model-explained-and-how-to-easily-remember-its-7-layers.html (visited on 1/8/2022).

[Sta13] Stack Exchange. *What is the difference between "simulate" and "emulate"?* 2013. URL: https://english.stackexchange.com/questions/111787/what-is-the-difference-between-simulate-and-emulate (visited on 10/8/2022).

[Sta17] Stack Overflow. *Simulator or Emulator? What is the difference?* 2017. URL: https://stackoverflow.com/questions/1584617/simulator-or-emulator-what-is-the-difference (visited on 10/8/2022).

[TB22] Martin Thomson and Cory Benfield. *HTTP/2*. RFC 9113. June 2022. DOI: 10.17487/RFC9113.

[Tec21] Technopedia. *What is Encoding? — Definition from Technopedia*. 2021. URL: https://www.techopedia.com/definition/948/encoding (visited on 26/7/2022).

[Tel21] Telenet BV. *TLN WRO Specification type Dcument*. TLN WRO Specification type Document TLN_WRO_TA_I_S_PDAA V3.0. Specification and Certification AO STB. Telenet BV, Apr. 2021. 46 pp.

[THE20] THEOplayer. *Low Latency DASH (LL-DASH)*. 2020. URL: https://www.theoplayer.com/blog/low-latency-dash (visited on 25/7/2022).

[The22a] The Bufferbloat community. *Bufferbloat*. 2022. URL: https://www.bufferbloat.net/projects/ (visited on 6/8/2022).

[The22b] The Linux Documentation Project. *Components of Linux Traffic Control*. 2022. URL: https://tldp.org/HOWTO/Traffic-Control-HOWTO/components.html (visited on 15/8/2022).

[Thu22] Rob Thubron. *Number of teens using Facebook crashes as YouTube becomes platform of choice*. 2022. URL: https://www.techspot.com/news/95594-number-teens-using-facebook-crashes-youtube-becomes-platform.html (visited on 13/8/2022).

[Uni12] European Broadcasting Union. *TAKING DASH TO THE NEXT LEVEL AT IBC2012*. 2012. URL: https://www.ebu.ch/news/2012/09/taking-dash-to-the-next-level-at (visited on 10/2/2022).

[Wik22] Wikipedia. *Root cause analysis*. 2022. URL: https://en.wikipedia.org/wiki/Root_cause_analysis (visited on 11/8/2022).

[Wil15] Nicolás Williams. *JavaScript Object Notation (JSON) Text Sequences*. RFC 7464. Feb. 2015. DOI: 10.17487/RFC7464.

[Wir22] Wireshark. *Wireshark. Go Deep*. 2022. URL: https://www.wireshark.org/ (visited on 11/8/2022).