UHASSELT
KNOWLEDGE IN ACTION

Maastricht University

Faculteit Wetenschappen
School voor Informatietechnologie

master in de informatica

Masterthesis

Networking and Security - Optimisations for ABR over QUIC using cross-layer metrics sharing

Arno Verstraete

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

PROMOTOR :

Prof. dr. Peter QUAX

2021
2022

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

Networking and Security - Optimisations for ABR over QUIC using cross-layer metrics sharing

Arno Verstraete

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

PROMOTOR :

Prof. dr. Peter QUAX

Hasselt University

Masterproef voorgedragen tot het behalen van de graad van master in de informatica

Optimisations for ABR over QUIC using cross-layer metrics sharing

Author:

Arno Verstraete

Promotor:

Prof. dr. Peter Quax

Mentors:

Joris Herbots,
dr. Maarten Wijnants

2021-2022

Contents

Acknowledgements

Firstly, I would like to thank Prof. dr. Peter Quax for giving me this opportunity and providing me with valuable feedback and support during the more difficult periods of this thesis. I want to express my gratitude to my mentor Joris Herbots for guiding me through this entire process and for introducing me to many of the technologies discussed in this thesis. Also, allowing me to work at UHasselt EDM during the summer with him has helped accelerate my research immensely. I learned so much more by being surrounded by experts and peers. I also want to thank dr. Maarten Wijnants for being my mentor during Joris's absence. I want to thank him in particular for helping me find the right direction for this thesis.
I would also like to acknowledge Mike Vandersanden and Olaf Van Bylen. Their different perspectives on my topics sparked interesting conversations during our stay at UHasselt EDM, from which I have learned a lot. Lastly, I thank my family and friends for their continuous support and encouragement throughout my studies.

Abstract

Adaptive Bitrate Streaming (ABR) is a technology that allows video to be streamed to a client in multiple bitrates by dividing a video into small segments and offering multiple versions of each segment. DASH is an ABR technology that uses HTTP to download segments of a video stream. Based on metrics gathered during media playback (e.g., goodput, round-trip-times), the ABR algorithm decides when to download segments and also in what kind of quality. It essentially attempts to download the highest possible bitrate that the network throughput allows. The network layer is a "black box" for the ABR algorithm. It requests a segment and has little to no feedback during the download of that segment. Because of this, ABR algorithms usually only make decisions every time a new segment arrives. There is information about the transport layer that could be useful for the ABR algorithm, like packet loss, packet arrival rate, RTT etc. This thesis investigates whether existing ABR algorithms can benefit from cross-layer metric sharing. We test if the transport layer and application layer can exchange information with each other and if this sharing could improve the performance of ABR algorithms.

It is difficult to make changes to TCP, the protocol used by HTTP/2 and older versions of HTTP. QUIC is a transport layer protocol that offers the same features as TCP and is used by HTTP/3. QUIC implementations currently exist in user-space. Because of this, we can make changes to the implementation more easily and benefit from cross-layer information sharing. We use a DASH framework called goDASH together with quic-go to implement cross-layer information sharing.

Firstly, we investigate whether replacing the metrics used by an existing ABR algorithm with metrics gathered from QUIC can improve the algorithm's performance. We do this by making a cross-layer version of the Average ABR algorithm. Average is a rate-based ABR algorithm that attempts to estimate throughput. Our cross-layer version is called AverageXL (Average Cross Layer). AverageXL performs worse than the original Average algorithm, but we believe that with some further optimisations, both algorithms can perform equally.

Secondly, we implement a new type of ABR algorithm that exploits the power of cross-layer information sharing by analysing QUIC packet information during the download of a segment to predict if stalling will occur. If stalling is detected, the ABR algorithm will abort the download of the segment and switch to a lower representation. We based this new algorithm on BBA, an existing buffer-based ABR algorithm. Our new algorithm, BBA-XL (BBA Cross Layer), performs better than BBA in a stress scenario that was specifically designed to point out the flaws of BBA. BBA-XL and BBA perform equally in a realistic scenario using a cellular dataset.

Cross-layer information sharing is possible when using QUIC. It can provide an ABR algorithm with additional metrics that allow for better-informed decision-making. We only used basic metrics gathered from QUIC for BBA-XL. There is a lot of untapped potential in cross-layer metrics sharing for ABR video streaming. Future work should investigate new kinds of ABR algorithms that use more metrics gathered from the transport layer. Future work should also look into sharing metrics from the application layer to the transport layer or metrics sharing in both ways.

Abbreviations

ABR Adaptive Bitrate Streaming.

AverageXL Average Cross-Layer.

BBA Buffer-Based Algorithm.

BBA-XL Buffer-Based Algorithm Cross- Layer.

CWND Congestion Window.

DASH Dynamic Adaptive Streaming over HTTP.

FIFO First In First Out.

HOL Head-Of-Line.

HTTP Hypertext Transfer Protocol.

HTTPS Hypertext Transfer Protocol Secure.

IETF Internet Engineering Task Force.

MAA Mean Average Algorithm.

MOS Mean Opinion Score.

MPD Media Presentation Description.

MPEG Moving Picture Experts Group.

QoE Quality of Experience.

RFC Request For Comments.

SSIM Structural Similarity Index Measure.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

UDP User Datagram Protocol.

VBR Variable Bitrate Encoding.

Nederlandse samenvatting

Videostreaming is vandaag verantwoordelijk voor het grootste deel van het internetgebruik. Volgens een onderzoek van Sandvine 2022 was 53,72 procent van al het internetverkeer in 2021 videostreaming. Videostreamingplatformen zoals Netflix[1] en YouTube[2] maken gebruik van Adaptive Bitrate Streaming (ABR) technologie. ABR biedt de video aan in verschillende versies van verschillende kwaliteiten. Elke versie noemen we een representatie. Een representatie wordt gedefinieerd door de bitrate, resolutie, codec, audiokanalen, taal etc. Elke representatie wordt in segmenten aangeboden. Zo bestaat elke representatie van een video van 60 seconden bijvoorbeeld uit 30 segmenten die elk twee seconden lang zijn. Het aanbieden van veel verschillende versies zorgt ervoor dat de software die aan de kant van de client de video streamt (de ABR client), kan kiezen welke versie het beste is om te downloaden in de huidige netwerksituatie. Zo kan een smartphone die met een mobiel netwerk verbonden is kiezen voor een lagere kwaliteit en een computer die met kabelinternet verbonden is voor een hogere kwaliteit. Het indelen van de video in segmenten zorgt er ook voor dat een ABR client tijdens het streamen van kwaliteit kan veranderen. Wanneer de internetverbinding plots heel slecht wordt bijvoorbeeld, kan de client overschakelen naar een lagere kwaliteit om ervoor te zorgen dat de video niet gaat vastlopen.

Dynamic Adaptive Streaming over HTTP (DASH) is een voorbeeld van een ABR-technologie. DASH maakt gebruik van HTTP-berichten om segmenten op te vragen. HTTP/2 en lager gebruikt TCP in de transportlaag. TCP is een "zwarte doos" voor DASH (de applicatielaag) en andersom. Beide lagen weten niet van elkaar wat ze doen. Nochtans gebeurt er veel op de transportlaag dat interessant kan zijn voor de ABR-algoritmes die kiezen van welke representatie een segment gedownload moet worden. Een ABR-algoritme krijgt enkel feedback over de netwerksituatie wanneer een segment klaar is met downloaden. Hij kan dan bijvoorbeeld berekenen hoe lang het heeft geduurd om dat segment door te sturen om zo een inschatting te maken van de doorvoersnelheid op het netwerk. De transportlaag daarentegen heeft deze informatie voor elk IP-pakket dat gestuurd wordt. Het weet ook wanneer er pakketverlies is, de Round Trip Time (RTT) van elk IP pakket etc. Als we deze metrieken zouden kunnen communiceren over de lagen heen (cross-layer metrics sharing), dan zouden we ABR-algoritmes kunnen maken die veel beter geïnformeerd zijn [Herbots e.a. 2020].

In deze thesis maken we gebruik van QUIC als transportlaagprotocol om metrieken van QUIC naar een DASH client te sturen. We gebruiken deze metrieken dan om een bestaand ABR-algortime aan te passen, en om een nieuw soort ABR-algoritme te ontwikkelen.

Onderzoeksvragen

In deze thesis trachten we de volgende onderzoeksvragen te beantwoorden:

1. Kunnen we de gecommuniceerde metrieken van de transportlaag gebruiken in de applicatielaag om een bestaand ABR-algoritme te verbeteren?

2. Kunnen we een nieuw soort ABR-algoritme ontwikkelen door gebruik te maken van de metrieken uit de transportlaag in het ABR-algoritme? Kan dit algoritme voor streams van een hogere kwaliteit zorgen dan bestaande algoritmes?

De hypotheses voor de onderzoeksvragen:

1. Het verschil in de metrieken die gemeten worden in de transportlaag gaan niet groot genoeg zijn om het ABR-algoritme een andere beslissing te laten maken en zal dus niet voor een verbetering in kwaliteit zorgen.

[1] https://www.netflix.com
[2] https://www.youtube.com

2. Het zou mogelijk moeten zijn om een ABR-algoritme te ontwikkelen dat gebruikmaakt van de metrieken uit de transportlaag om een gelijkaardige of een betere kwaliteit te kunnen hebben dan bestaande ABR-algoritmes.

QUIC

QUIC is een nieuw transportlaagprotocol dat tracht dezelfde kenmerken te hebben als TCP. Het kan pakketten betrouwbaar en in de juiste volgorde van de verzender naar de ontvanger sturen. QUIC is gebouwd bovenop UDP. De reden daarvoor is zodat de middleboxes (al de apparaten die tussen de zender en ontvanger liggen op het netwerk) niet een nieuw transportlaagprotocol moeten ondersteunen [Robin Marx 2021]. Het is zeer moeilijk om nieuwe functies te introduceren in TCP omdat de middleboxes nieuwe versies van TCP vaak niet ondersteunen.

QUIC-implementaties bestaan momenteel allemaal in user-space. Om deze reden kunnen we gemakkelijker aanpassingen maken aan de implementaties en het delen van metrieken uit QUIC naar de applicatielaag mogelijk maken. Voor onze implementatie maken we gebruik van quic-go [Clemente 2022], een QUIC-implementatie geschreven in Go. We hebben deze implementatie aangepast zodat qlog berichten over een kanaal naar de bovenliggende laag gestuurd kunnen worden. Deze qlog berichten zijn ontwikkeld om software die QUIC gebruikt gemakkelijker te kunnen debuggen [Robin Marx, Lamotte e.a. 2018]. Hierdoor wordt veel interessante informatie bijgehouden over de verzending en aankomsten van pakketten, het verlies van pakketten etc. Die informatie is bruikbaar voor het maken van een ABR-algoritme dat metrieken uit de transportlaag gebruikt.

ABR streamen met DASH

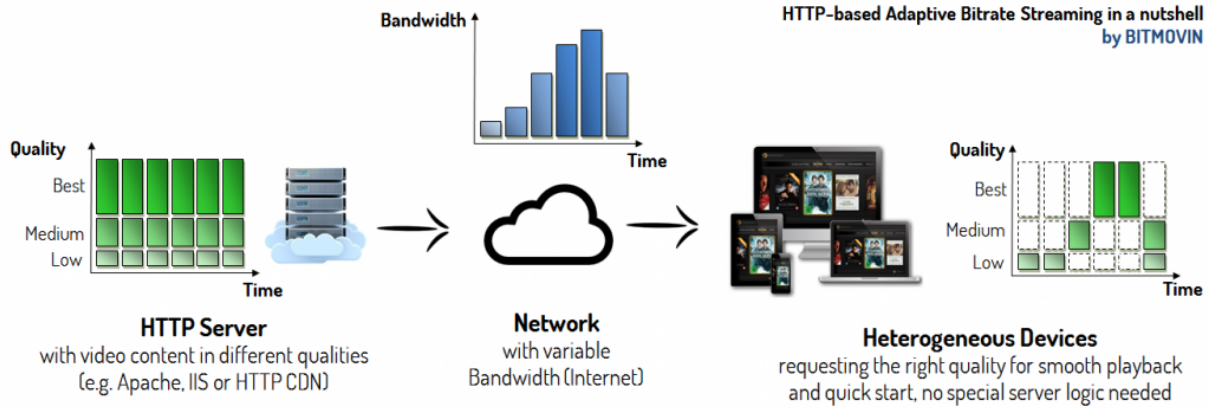
Figuur 1 toont de structuur van een DASH stream. Een DASH stream begint bij het downloaden van de Media Presentation Description (MPD) van de video. Dat is een XML-bestand waarin staat wat voor representaties er beschikbaar zijn voor deze video. De client kiest hier dan een representatie uit om te beginnen downloaden. De client heeft een videobuffer. Dit is een buffer waarin gedownloade segmenten terechtkomen. De videobuffer zorgt ervoor dat het ABR-algoritme wat tijd heeft om te reageren op veranderingen in het netwerk. De videospeler gaat telkens een segment afhalen van de videobuffer wanneer het vorige segment afgespeeld is. Als de videobuffer leeg is wanneer de speler een segment uit de buffer wil halen gaat de video vastlopen omdat het segment niet op tijd klaar was met downloaden. Een goed ABR-algoritme zorgt ervoor dat segmenten van een zo hoog mogelijke kwaliteit worden gedownload zonder de videobuffer ooit leeg te laten lopen.

Voor de implementatie hebben we een DASH client nodig waarmee het makkelijk is om metingen te doen, die QUIC ondersteunt en die open-source is zodat we de code kunnen aanpassen. We hebben onze implementatie met het goDASH [Raca, Manifacier en Quinlan 2020; O'Sullivan, Raca en Quinlan 2020] framework gemaakt. Dit is een headless DASH client die in Go geschreven is en quic-go onderliggend gebruikt. Een headless client is een client dat de videosegmenten downloadt maar niet echt afspeelt. Het framework doet eigenlijk alsof het een echte DASH client is en gedraagt zich zoals een echte client zich zou gedragen. Er waren een aantal problemen met de oorspronkelijke implementatie van goDASH. Mike Vandersanden heeft deze problemen opgelijst en ook veel problemen verholpen in zijn eigen versie van goDASH[3]. Dit is de versie waar we onze implementatie op baseren.

Kwaliteit van een videostream meten

Om een nieuw ABR-algoritme te kunnen vergelijken met een bestaand ABR-algoritme hebben we een manier nodig om streams objectief met elkaar te vergelijken. Wanneer is een stream van een hogere kwaliteit dan een andere stream? We zouden de gemiddelde bitrate kunnen nemen van beide streams en deze vergelijken. We houden dan enkel rekening met de kwaliteit van de segmenten zelf. Wanneer echter een stream vaak wisselt tussen kwaliteiten kan dit opvallen voor de kijker en de waargenomen kwaliteit van de stream eigenlijk verlagen [Rodriguez e.a. 2014]. We moeten dus ook kijken naar wanneer en hoe vaak er gewisseld wordt tussen kwaliteiten. De grootste ergernis voor een gebruiker bij het streamen van video is wanneer de video vastloopt. We zouden kunnen kijken naar hoe vol de videobuffer is. We zouden ook kunnen tellen hoe lang een stream gemiddeld vastloopt en dit vergelijken.

[3] https://github.com/JorisHerbots/godash-qlogabr/blob/master/PROBLEMS.md



Figuur 1: De structuur van een DASH stream, door Mueller 2019.

Er bestaat een standaard, ITU P.1203, die rekening houdt met alle elementen die we net benoemd hebben en die deze gebruikt om een score te geven aan een videostream tussen 1 en 5. Een 1 staat voor een lage waargenomen kwaliteit en een 5 voor een hoge. ITU P.1203 is zo ontwikkeld dat de score die gegenereerd wordt heel gelijkaardig is aan de score die een echte groep mensen zou geven aan die stream [Robitza e.a. 2018].

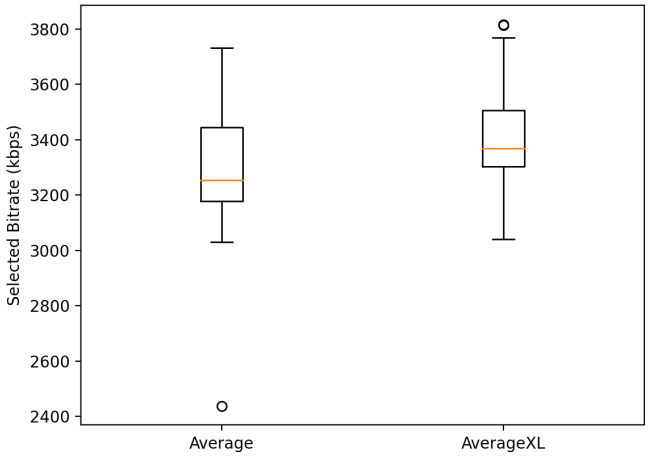
AverageXL

Om de eerste onderzoeksvraag te beantwoorden hebben we een nieuwe versie gemaakt van Average, een bestaand ABR-algoritme dat de gemiddelde doorvoersnelheid van de gedownloade segmenten berekent. Aan de hand van dat gemiddelde gaat Average een segment downloaden dat eenzelfde bitrate heeft als de gemiddelde doorvoersnelheid, of er net onder. Average is geen complex ABR-algoritme. Het werkt goed in situaties waar de doorvoersnelheid nooit verandert, maar als dit wel drastisch verandert gaat Average oude netwerkinformatie gebruiken en soms een slechte inschatting maken. We hebben Average gekozen voor het eerste cross-layer algoritme omdat het een relatief simpel algoritme is en dus perfect kan dienen als "proof of concept" om aan te tonen dat het delen van metrieken over de lagen heen überhaupt kan werken. Onze hypothese voor onderzoeksvraag 1 luidde dat de metrieken een bestaand algoritme vervangen door de metrieken uit de transportlaag geen invloed gaat hebben op het gedrag van dat algoritme. Stel, we meten de gemiddelde doorvoersnelheid op transportlaagniveau. We meten wanneer pakketten vertrekken en aankomen. Aan de hand van de grootte van die pakketten kunnen we de doorvoersnelheid van dat pakket meten. Als we het gemiddelde nemen van al die metingen hebben we de gemiddelde snelheid over de hele videostream. Deze berekening zou geen ander resultaat mogen geven dan de gemiddelde snelheid berekenen op applicatieniveau. Als de berekening ook (bijna) hetzelfde is, gaat het Average algoritme ook geen andere beslissing maken.

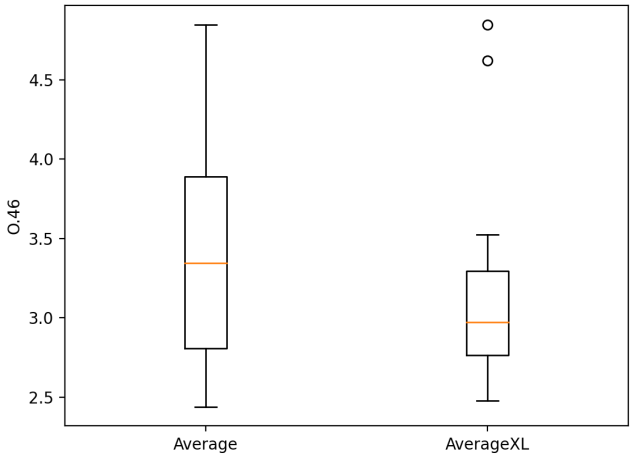
Onze hypothese was echter niet volledig juist. Onze eigen implementatie, die metrieken van de transportlaag deelt met het ABR-algoritme en daarmee de gemiddelde doorvoersnelheid berekent, heet AverageXL (Average Cross Layer). Figuur 2 toont de resultaten van 20 tests die we met beide algoritmen hebben uitgevoerd op een dataset van Akamai van een mobiel netwerk. AverageXL presteert slechter dan Average volgens deze resultaten. AverageXL kiest voor hogere representaties dan Average, maar de video loopt vaker vast (stall time) en veroorzaakt daarom een lagere ITU P.1203 score. Dit komt mogelijks omdat de meting van de doorvoersnelheid in de transportlaag niet helemaal hetzelfde is als in de applicatielaag. De transportlaag meet namelijk ook de HTTP headers mee in de pakketgroottes. Hierdoor lijkt de doorvoersnelheid in de transportlaag hoger te zijn dan in de applicatielaag. We zouden deze fout kunnen compenseren door in te schatten wat het verschil is en dit in rekening te brengen bij de berekening van de doorvoersnelheid.

BBA-XL

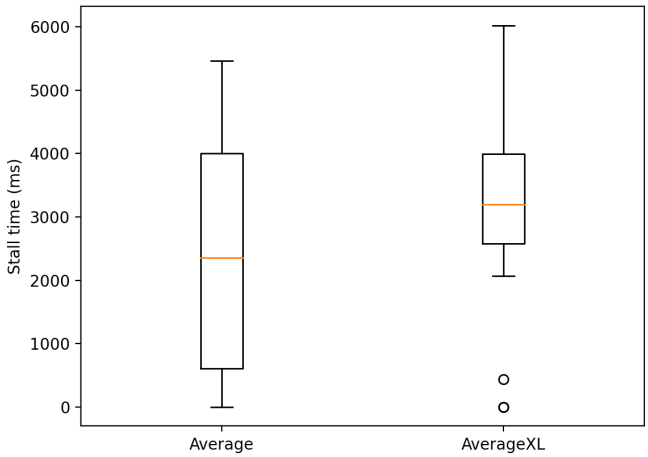
Het voordeel van toegang hebben tot de metrieken van de transportlaag in een ABR-algoritme is dat we veel meer granulariteit hebben in data. Een normaal ABR-algoritme krijgt enkel informatie over de netwerksituatie wanneer een segment is toegekomen. In vergelijking, de transportlaag krijgt informatie



(a) Boxplots van de gekozen bitrates van Average and AverageXL. De bitrate is in kbps en geeft de bitrate van de gekozen representatie op dat moment weer.

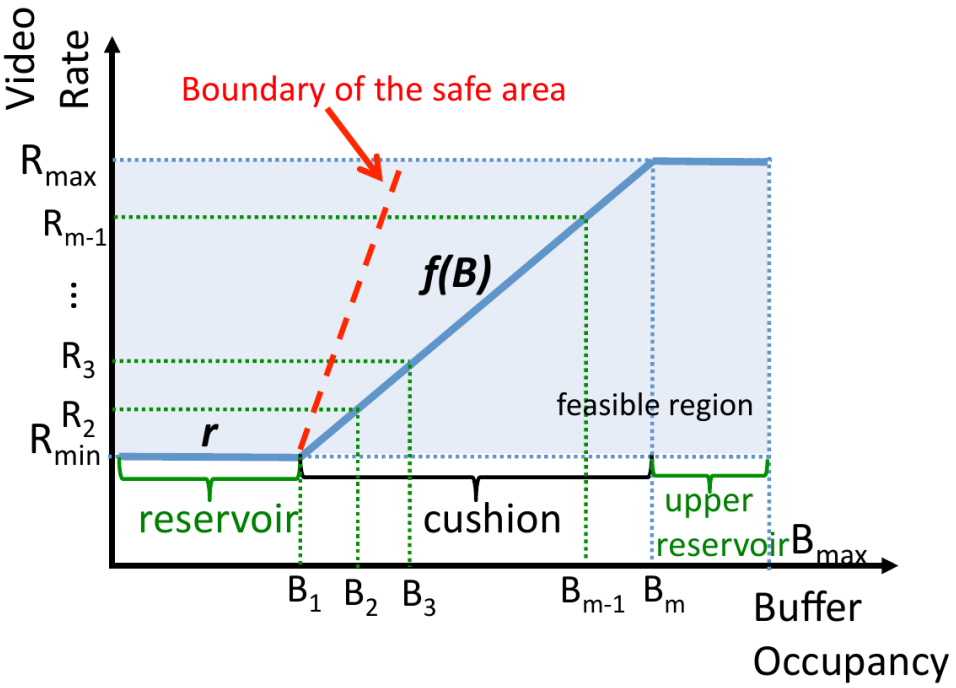


(b) Boxplots van de ITU P.1203 O.46 score van Average en AverageXL. De score geeft weer hoe echte gebruikers de stream zouden beoordelen met een score tussen 1 en 5. Een hogere score betekent een hogere ervaren kwaliteit.



(c) Boxplots van de totale tijd dat een stream is vastgelopen voor Average en AverageXL. Hoe minder seconden de stream is vastgelopen hoe beter.

Figuur 2: Een vergelijking van Average en AverageXL. De metingen zijn gebeurd met een netwerksimulate die een dataset volgt van Akamai van een mobiel netwerk. Voor beide algoritmen hebben we 20 tests uitgevoerd.

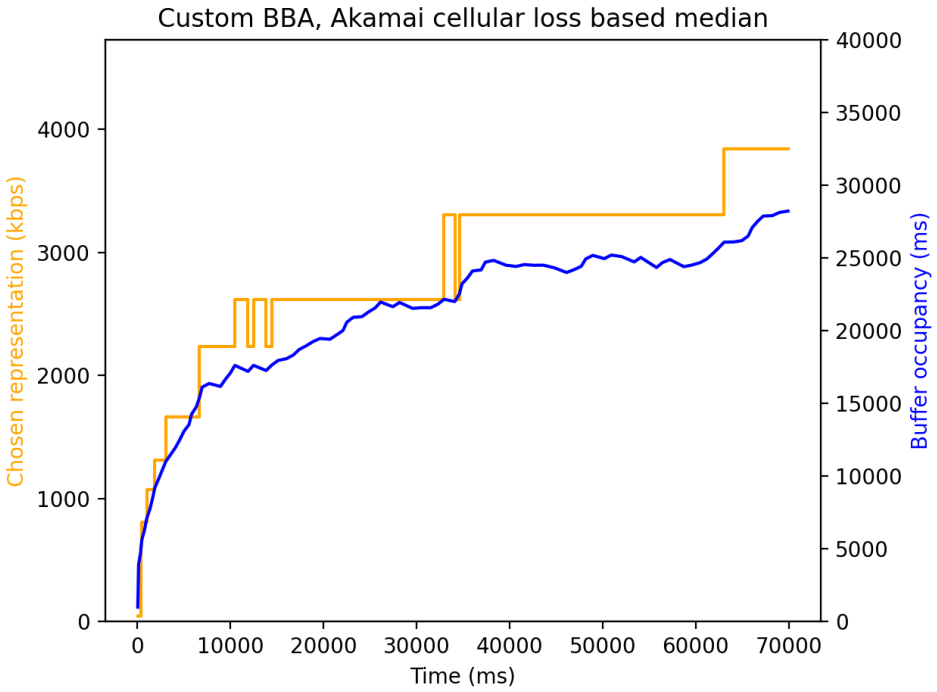


Figuur 3: Werking van BBA-1, door Huang e.a. 2014a. Buffer occupancy is het bufferniveau, het aantal seconden aan video dat nog in de videobuffer zit. Video rate is de bitrate van de beschikbare representaties.

per pakket. In onze tests met een stream met een segmentgrootte van één seconde betekende dit dat het ABR-algoritme ongeveer één maal per seconde een pakket aankreeg, en dus één maal per seconde informatie over het netwerk kreeg. De transportlaag deelt dat segment van één seconde op in enkele honderden pakketten, en kreeg dus informatie over het netwerk enkele honderden keren per seconde. De granulariteit is hier dus veel hoger.

Voor de tweede implementatie van een cross-layer ABR-algoritme willen we gebruikmaken van deze extra granulariteit. In plaats van een ABR-algoritme enkel beslissingen te laten maken wanneer een pakket aankomt, willen we functies toevoegen die ook beslissingen maken tijdens het downloaden van een pakket, wat een klassiek ABR-algoritme niet kan. Als basis voor het nieuwe algoritme gebruiken we BBA [Huang e.a. 2014a]. Hiervoor hebben we een eigen implementatie van BBA-1 gemaakt. Er bestaat een BBA-2-implementatie voor goDASH, maar deze gedraagt zich niet zoals we verwachten dat BBA-2 zich gedraagt. We hebben in de plaats daarvan een eigen implementatie geschreven van BBA-1 die wel werkt zoals we verwachten. BBA staat voor Buffer-Based Algoritm. BBA is anders dan rate-based algoritmes zoals het eerder besproken Average, omdat BBA niet tracht de huidige doorloopsnelheid in te schatten. BBA baseert zich op het bufferniveau (hoeveel seconden aan video er nog in de videbuffer zitten). Hoe hoger het bufferniveau, hoe hoger de representatie zal zijn dat het algoritme kiest. Figuur 3 toont hoe het BBA-1 algoritme beslissingen maakt. Telkens als er een segment gedownload is, gaat het algoritme kijken naar het bufferniveau. Als het bufferniveau lager is dan het onderste reservoir, gaat het algoritme kiezen voor de laagst mogelijke representatie. Als het bufferniveau boven het bovenste reservoir is, dan kiest het algoritme voor de hoogste representatie. Tussen de twee reservoirs is de cushion. Als het bufferniveau hierin ligt, mappen we bufferniveaus op bitrates van representaties. We sorteren de representaties volgens hun bitrate van laag naar hoog. De laagste bitrate is R_{min} en de hoogste is R_{max}. Het laagste bufferniveau van de cushion is B_1 en het hoogste is B_m. De representaties tussen R_{min} en R_{max} mappen we op bufferniveaus tussen B_1 en B_{max}. Figuur 4 toont hoe BBA zich gedraagt in een mobiel netwerk. Het bufferniveau kruipt langzaam omhoog, en de gekozen representatie volgt telkens.

Om onderzoeksvraag 2 te beantwoorden maken we gebruik van de extra granulariteit van de informatie uit de transportlaag, en de mogelijkheid om informatie te krijgen tijdens het downloaden van een segment. BBA-XL (BBA Cross Layer) is een nieuw ABR-algoritme dat we gebaseerd hebben op BBA-1. BBA-XL heeft de mogelijkheid om de download van een segment vroegtijdig af te breken als het algoritme denkt dat het segment niet klaar zal zijn met downloaden voordat de videobuffer leeg is. Wanneer de videobuffer leeg is, loopt het afspelen van de stream vast omdat we moeten wachten tot het volgende segment gedownload



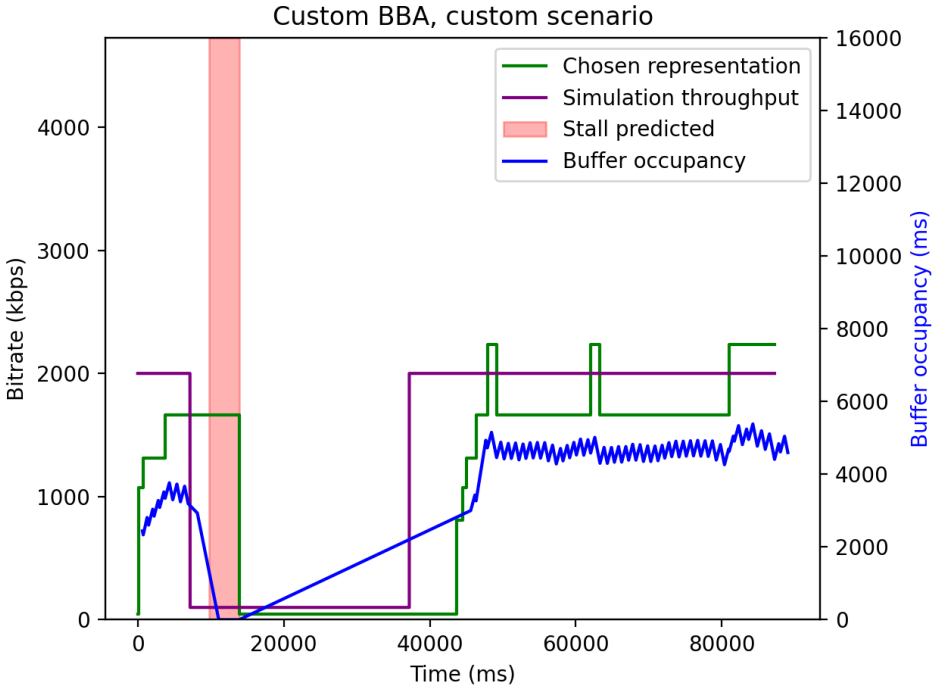
Figuur 4: Bufferniveau (buffer occupancy) en de gekozen representatie (chosen representation) van de eigen BBA-implementatie tijdens een videostream met een dataset van Akamai van een mobiel netwerk.

is. We willen met het vroegtijdig afbreken van segmenten ervoor zorgen dat dit minder gebeurt, om zo een stream van hogere ervaren kwaliteit te hebben. BBA-XL schat de huidige doorvoersnelheid van het netwerk in door een *prediction window* op te bouwen en de gemiddelde doorvoersnelheid van de window te berekenen. De prediction window kan je zien als een FIFO queue. Wanneer we een segment beginnen downloaden is de queue leeg. Telkens als er een pakket op de transportlaag toekomt voegen we dit toe aan de queue. De queue heeft een maximumgrootte. Wanneer de queue vol is verwijderen we het laatste pakket wanneer we een nieuw pakket toevoegen. Zo blijft de inschatting zeer recent. BBA-XL gaat enkel de doorvoersnelheid inschatten wanneer de prediction window gevuld is, om te voorkomen dat we een berekening maken die niet representatief is omdat er maar een paar pakketten in de window zitten. Aan de hand van de queue berekent BBA-XL of het pakket dat momenteel gedownload wordt klaar gaat zijn met downloaden voordat de buffer leeg is. Als dat het geval is dan doet BBA-XL niets anders dan dat BBA zou doen. Als dat niet het geval is, dan gaat BBA-XL het segment afbreken en onmiddellijk datzelfde segment opnieuw downloaden in de laagst beschikbare kwaliteit.

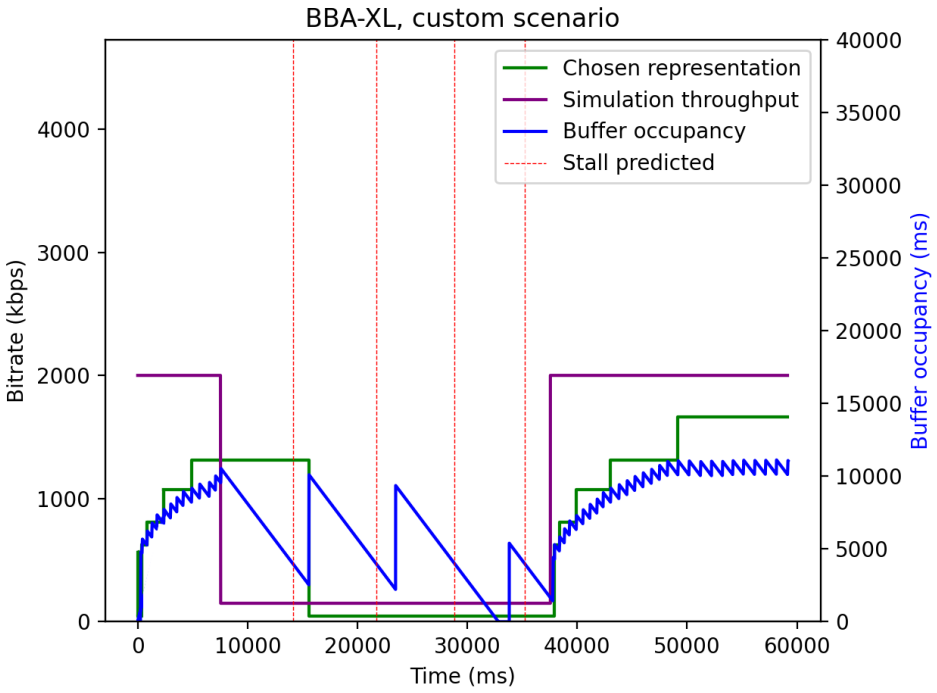
Figuur 5 toont hoe onze gewone BBA implementatie zich gedraagt in een scenario dat speciaal gemaakt is om een zwak punt van BBA weer te geven. De paarse lijn toont de doorvoersnelheid die de netwerksimulatie toelaat. Die begint relatief hoog met 2000kbps. Dat zorgt ervoor dat het algortime een buffer kan opbouwen (blauwe lijn) en naar een hogere representatie gaat schakelen (groene lijn). Plots daalt de doorvoersnelheid van de simulatie sterk. BBA was op dat moment een segment aan het downloaden van 1600kbps. De nieuwe doorvoersnelheid is zo traag dat dit segment niet op tijd gedownload wordt. Hierdoor daalt het bufferniveau (blauwe lijn) tot op nul en de video loopt vast. De video begint pas terug met afspelen rond de 40000ms omdat dan de buffer terug gevuld is.

Figuur 6 toont hetzelfde scenario als Figuur 5, maar deze keer wordt BBA-XL gebruikt met de mogelijkheid om segmenten af te breken. De rode lijnen hier geven weer wanneer BBA-XL beslist om een segment af te breken. Het bufferniveau wordt een aantal keren heel laag, maar het afbreken van het segment zorgt ervoor dat dit niet nul wordt. Na de derde afbraak komt het bufferniveau wel even op nul, waardoor de video nog steeds even vastloopt, maar dit is veel minder dan de gewone BBA implementatie. Figuur 7 toont de ITU P.1203 kwaliteitsscores voor BBA en BBA-XL. Deze houdt dus rekening met onderbroken video, gekozen representaties, kwaliteitswissels etc. Hier scoort BBA-XL beter dan BBA, voornamelijk omdat er minder onderbrekingen zijn.

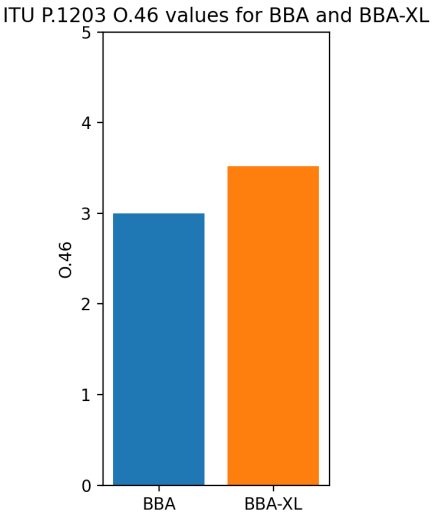
Het is goed dat BBA-XL in dit specifieke scenario beter presteert, maar het moet ook in algemene



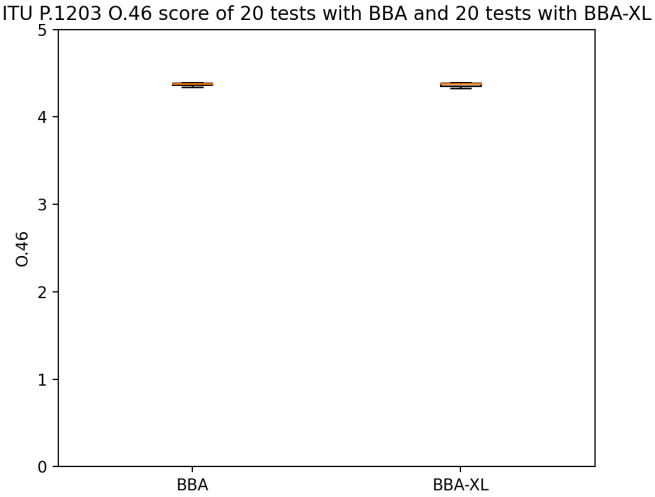
Figuur 5: Oorspronkelijk BBA-algoritme in een netwerkscenario speciaal gemaakt om een zwak punt van BBA te laten zien. De rode balk toont aan op welk moment BBA-XL het segment zou afbreken. De paarse lijn geeft de doorvoersnelheid weer dat het netwerkscenario toelaat. Deze blijft constant op 2000kbps tot het plots zakt. Dit gebeurt tijdens BBA een segment aan het downloaden is aan een te hoge kwaliteit omdat aan het begin van de segmentdownload het bufferniveau (blauwe lijn) nog hoog was. Uiteindelijk bereikt het bufferniveau nul voordat het segment aankomt, waardoor de video vastloopt.



Figuur 6: BBA-XL in hetzelfde scenario als BBA in Figuur 5. BBA-XL heeft in dit geval een grotere videobuffer. De rode lijnen duiden aan wanneer BBA-XL segmenten vroegtijdig afbreekt omdat het voorspelt dat het huidige segment niet op tijd gaat toekomen.



Figuur 7: ITU P.1203 O.46 scores van BBA en BBA-XL in het speciaal gemaakte scenario van Figuur 5 en 6.



Figuur 8: Boxplots van de ITU P.1203 O.46 scores van BBA en BBA-XL.

situaties goed werken. We hebben daarom 20 tests uitgevoerd op beide algoritmen met een dataset van een mobiel netwerk. BBA-XL mag geen valse positieven genereren. Dit zou betekenen dat BBA-XL voorspelt dat er een onderbreking gaat zijn, terwijl er geen zou geweest zijn. Figuur 8 toont de ITU P.1203 kwaliteitsscores van beide algoritmen. De scores zijn zo goed als hetzelfde. Beide algoritmen hebben geen onderbreking ervaren in alle testen. Dit geeft niet alleen weer dat BBA-XL weinig tot geen valse positieven heeft, maar het geeft ook goed weer dat het oorspronkelijke BBA algoritme al goed werkt in een uitdagend scenario.

De hypothese voor onderzoeksvraag 2 was correct. BBA-XL presteert gelijkaardig aan BBA in algemene testen. In een scenario specifiek gemaakt om BBA in de problemen te brengen presteert BBA-XL beter.

Conclusie

We hebben ABR-algoritmes Average en BBA aangepast zodat informatie uit de transportlaag beschikbaar is voor het ABR-algoritme. We hebben hiervoor een aanpassing gemaakt aan de quic-go QUIC implementatie en het goDASH framework zodat er een communicatielink is tussen beide lagen. De cross-layer versie van Average, AverageXL, presteerde minder goed dan verwacht. Maar dit is mogelijks op te lossen door optimalisaties toe te voegen. BBA-XL, de cross-layer versie van BBA, heeft de mogelijkheid om de download van segmenten af te breken als het algoritme voorspelt dat deze niet gaan aankomen voor de videobuffer leeg is. Het afbreken van segmenten en overschakelen naar een lage representatie zorgt er in de tests voor dat de video minder onderbroken wordt. In algemene tests met BBA en BBA-XL presteren beide algoritmen heel gelijkaardig.

Chapter 1

Introduction

1.1 Motivation

Video streaming is responsible for the greatest share of internet usage today. In 2021, video streaming was responsible for 53.72 percent of the total internet traffic measured by Sandvine 2022, and it is continuing to rise. According to the report, video streaming platforms like YouTube[1] and Netflix[2] are one of the major players in this. These video streaming platforms use Adaptive Bitrate Streaming (ABR) technology to stream their content. ABR is a technology that allows a client streaming video or audio to be able to switch between different versions of that content on-the-fly. This allows the client to adapt their streaming behaviour depending on the network situation. Imagine someone streaming video on a mobile network. If the user's connection to the mobile network suddenly deteriorates, the streaming client will switch to a lower bitrate version of that video to prevent stalling. This is the power of Adaptive Bitrate Streaming. However, Adaptive Bitrate Streaming still has problems that have not been solved [Akhshabi et al. 2012; Arye, Sen, and Freedman 2018]. ABR algorithms can, for example, underestimate their fair share of bandwidth on the network. They suffer from a shrinking window size during the moment no segments are downloaded, which causes them to have a lower throughput than they deserve.

In 2021, the net transport layer protocol QUIC was officially formalised in RFC 9000 by the Internet Engineering Task Force (IETF) [Jana Iyengar 2021; J. Iyengar and Thomson 2021]. QUIC is built on top of UDP and introduces features we know from TCP like reliable packet transmission, in-order delivery, congestion avoidance etc. It also uses features like 0-RTT and TLS encryption on the transport layer, among others [Mike Bishop 2021]. QUIC is an interesting development for ABR streaming because the next version of HTTP: HTTP/3 will use QUIC as its transport layer protocol instead of TCP [M. Bishop 2022; Garza 2020]. ABR technologies like HTTP Adaptive Streaming (HAS) and Dynamic Adaptive Streaming over HTTP (DASH) use HTTP to download the video and audio segments. This means ABR streaming will naturally convert to using QUIC instead of TCP as its transport layer protocol.

Currently, QUIC implementations exist in user-space. A transport-layer protocol is usually implemented in the kernel of the operating system. These QUIC implementations are essentially libraries written on top of UDP. This makes it easier to make changes to the implementation. When we use QUIC as the transport layer protocol for ABR streaming, both the application layer (ABR) and the transport layer (QUIC) are in user-space. This opens up the possibility to communicate between layers. This is called-cross layer communication. We could, for example, implement a cross-layer communication channel between the QUIC implementation and the application that uses it. In the context of ABR streaming, it essentially means that the ABR algorithm and the QUIC transport layer would be able to communicate. Cross-layer information sharing could possibly be used to improve the performance of ABR algorithms [Herbots et al. 2020]. There are many QUIC implementations in development that follow the IETF QUIC specification [quicwg 2022]. We can use these in a Proof of Concept implementation.

1.2 Research questions

Cross-layer communication between the ABR algorithm and the QUIC transport protocol can be realised in three distinct ways.

[1] https://www.youtube.com
[2] https://www.netflix.com

i The application layer receives information from the transport layer. The transport layer ignores the existence of a cross-layer communication channel and performs as usual. The transport layer protocol communicates its metrics to the layer above. The application will use this knowledge to change its behaviour according to what is happening in the transport layer.

ii The transport layer receives information from the application layer. The application layer ignores the existence of a cross-layer communication channel and performs as usual. It instead shares its metrics with the transport layer protocol, which will try to use it to its advantage.

iii Both layers receive information from each other. They both share their metrics with each other and attempt to use this information to change both layers' behaviour.

QUIC implementations are still heavily in development. There are currently also many differences between them [Robin Marx, Herbots, et al. 2020]. The current ABR algorithms have been around for several years. It is difficult to improve these ABR algorithms because there are not many metrics to work with. The transport layer is normally a black box for the ABR algorithm. The ABR algorithm has no idea what the transport layer is doing and therefore needs to use heuristics like throughput estimation and buffer-occupancy to mitigate this lack of knowledge. At the same time, the workings and intentions of an ABR algorithm are unknown to the transport layer. Not much research has been done about opening up these black boxes. Because of this, we will research cross-layer communication of type i where the transport layer informs the application layer of its decisions. Cross-layer communication of types ii and iii are interesting subjects for further development. In this thesis, we will attempt to answer the following **research questions**:

1. Can we use cross-layer information sharing to improve the Quality of Experience for existing ABR algorithms?

2. Can we use cross-layer information sharing to introduce a new type of ABR algorithm that uses metrics gathered from the transport layer to make better-informed decisions? Can this algorithm match or outperform existing ABR algorithms in terms of Quality of Experience?

There are many ABR algorithms and many implementations of them available. With research question 1 we want to adapt existing implementations and test if using cross-layer metrics sharing can improve their performance. Existing ABR algorithms have to work with the metrics that they can see in the application layer. These metrics are throughput and buffer occupation. If we can set up cross-layer information sharing, we will have access to many more metrics. This allows for the development of a new kind of ABR algorithm that is not guided by the metrics we can gather in the application layer but by all the metrics we can gather in the transport layer. This is what we will explore with research question 2.

Hypotheses for the research questions:

1. The difference between the metrics measured in the application layer and the transport layer will be so minimal that it will not affect the decision-making process of the algorithms and will thus not change their performance.

2. It should be possible to match or exceed the performance of existing ABR algorithms by using a new type of ABR algorithm that makes decisions based on cross-layer metrics.

Current ABR algorithms use throughput and buffer occupation as their primary metric for decision-making. Throughput is estimated by the time it takes the transport layer protocol to download the request video segment of a known size. If we know the size of the segment and the time it took to download that segment, we can estimate the network's throughput. This measurement is only performed after every segment is downloaded. This is a long time and could thus create inaccurate readings. The transport layer protocol can essentially perform the same measurement but on packages instead of on entire segments. When an ABR client requests a video segment, this segment is actually split into many small packets that are sent over the network. The estimation of the throughput on the transport layer level will thus be more accurate. However, we suspect that the difference between these estimations will not be large enough to make significant changes in the behaviour of the ABR client. This is described in hypothesis 1 for research question 1.

Constructing a new type of ABR algorithm that uses cross-layer metrics as on of the metrics for decision-making should at least be able to match the performance of existing ABR algorithms. We have access to an accurate estimation of the available throughput. We can still use buffer occupation as a metric. On top of that, we will know information about packet drops, packet retransmissions etc. In theory, we should be able to make an equal or better algorithm with the use of extra network information compared to existing ABR algorithms. This is described in hypothesis 2 for research question 2.

Chapter 2

Background information

The later chapters of this thesis assume that the reader has a lot of prior knowledge about the topics of this thesis. Because of that, this chapter is included. We explain how ABR streaming works and how DASH handles this in particular. Then, we discuss the differences between ABR algorithms. Because we will be changing existing ABR algorithms and building new ones in Chapter 4, it is important to know how these algorithms can be evaluated, which is also discussed in this chapter. Lastly, we explain what QUIC is, and we discuss features of QUIC that are of interest to this thesis.

2.1 DASH

The streaming technique we will be using in this thesis is DASH. In this section, we will briefly explain what DASH is, how it works and why we use it.

2.1.1 Introduction to Adaptive Bitrate Streaming

Dynamic Adaptive Streaming over HTTP (DASH or MPEG-DASH) [MPEG 2019] is a streaming technique developed by MPEG. It is one of many adaptive bitrate (ABR) streaming techniques for HTTP. Another example of an ABR streaming technique is Apple's HTTP Live Streaming (HLS) [Apple 2022].

ABR streaming is a technique to provide multiple representations of a single video to a client. The client is able to download the representation of their choice. A representation could be any number of ways to represent a video in a different way. This could be among others: resolution, framerate, audio or video codec and bitrate. Many different kinds of devices access the internet, but every device has a different display resolution, supports different codecs etc. The availability of videos in different bitrates is also important for unstable connections between the client and the server. Think of mobile networks, wireless networks and busy wired networks. The ability to change bitrates makes it possible for clients to adapt to the network situation and improve the Quality of Experience by choosing what bitrate will yield the highest quality and the least stalling. This concept is depicted in Figure 2.2.

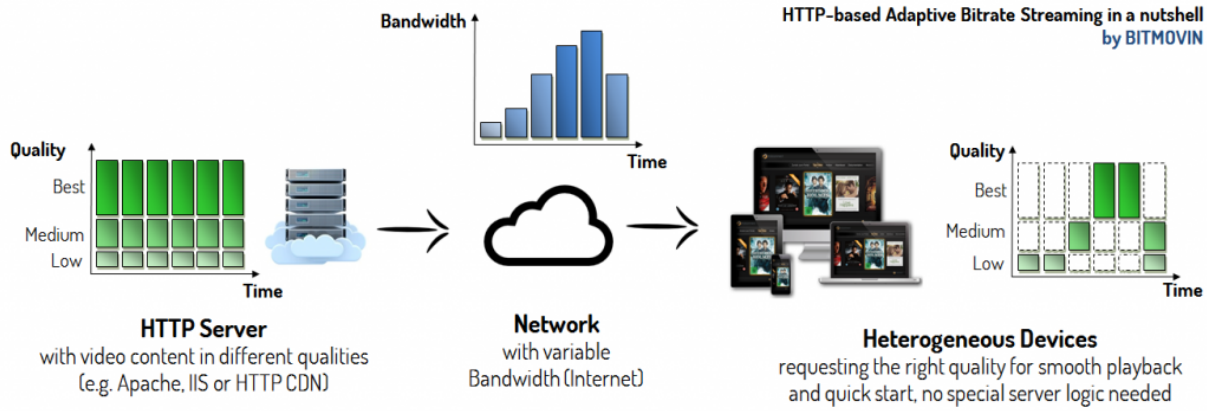


Figure 2.1: Structure of DASH streaming by Mueller 2019.

2.1.2 Inner workings of DASH

DASH provides two major ways to allow clients to switch representations while playing video. The first one is to split the video into many small segments at the server-side and provide these to the client. The client can then decide what segment to download according to their estimated bandwidth and device capabilities. The second option is to provide one whole video per representation. In this situation, the client will then download parts of that video file instead of complete segments. From this point on in this thesis, we will stay with segment-based approaches because they more clearly show the segmentation of video, even though they are functionally the same.

2.1.3 The manifest file

A client needs to know what representations exist and where it can access them. The role of the manifest file or the Media Presentation Description (MPD) is to tell the client exactly this. The MPD file is the first thing the ABR client does when starting a stream.

MPD The MPD file is an XML file. Listing 2.1 shows an example of an MPD file by DASH-IF. The root element of this file is the *MPD* tag. This tag contains global information about the video and the structure of the XML file. For example, the total duration of a video is indicated by the *mediaPresentationDuration*. In the example, this is *634.566 seconds*.

Period The child tags of the *MPD* node are *Period* tags. A period is a way to temporally subdivide a video stream. This can, for example, be used to divide a stream into chapters or insert ads. In the example, the entire stream is contained in a single period.

AdaptationSet The *AdaptationSet* tag is contained by the *Period* tag. An adaptation set indicates what type of content is defined in the child tags. Usually, video and audio are separated in a DASH stream. This is the case in our example MPD. Because of this, we have an adaptation set with attribute *contentType video* and *mimetype video/mp4* for the video segments, and we have a second adaptation set with attribute *contentType audio* and *mimetype audio/mp4* of the audio segments. If there are multiple versions for the audio, they will all have their own adaptation set. For example, if both single channel and multi-channel audio are available, both will have their own adaptation set [Garrison 2021]. The client chooses an Adaptation set at the start of the stream. It then keeps selecting segments from that Adaption unless the user would manually want to switch to a different adaptation set. For example, by changing the audio language.

Segments The child tags of the adaptation sets are used to describe the actual segments. In our example we can see the *SegmentTemplate* tag. This tag indicates how the client can find the segments in this adaptation set. The *media* attribute shows how the client needs to build the path to every segment. The parts between dollar signs in this string need to be changed into the values they represent. For example, *$RepresentationID$* will need to be changed into the ID of the representation the client has chosen. *$Number$* needs to be changed into the number of the segment that the client wants to download. The number of a segment can be determined by using the duration of the stream, the duration of a segment and the timescale. The duration of a segment is defined in the segment template. In our example this is 120 *frames*. The timescale indicates what framerate is used. This means that when a segment is 120 frames long and has a timescale of 30, every segment is 4 seconds long. The first segment will contain the data for $[0seconds, 4seconds[$. The second segment will contain the data for $[4seconds, 8seconds[$ and so forth.

Representations The *Representation* tag is used to indicate all the available representations of the adaptation set. In our example Listing 2.1, the first adaptation set is used for the video element of the stream. We can see that it has ten different *Representation* child tags, indicating that there are ten different representations that the ABR algorithm can choose from. Every representation has an ID. The client will use this ID to find the location of the segments to download. The representation also has attributes that explain the properties of this representation, like codec, bitrate, resolution and framerate. The ABR algorithm uses these properties to decide what representation is best. Bitrate is usually the most important property. The time it takes for a client to download a segment depends on the segment's bitrate, the segment duration, and the available bandwidth. ABR algorithms use all this information to

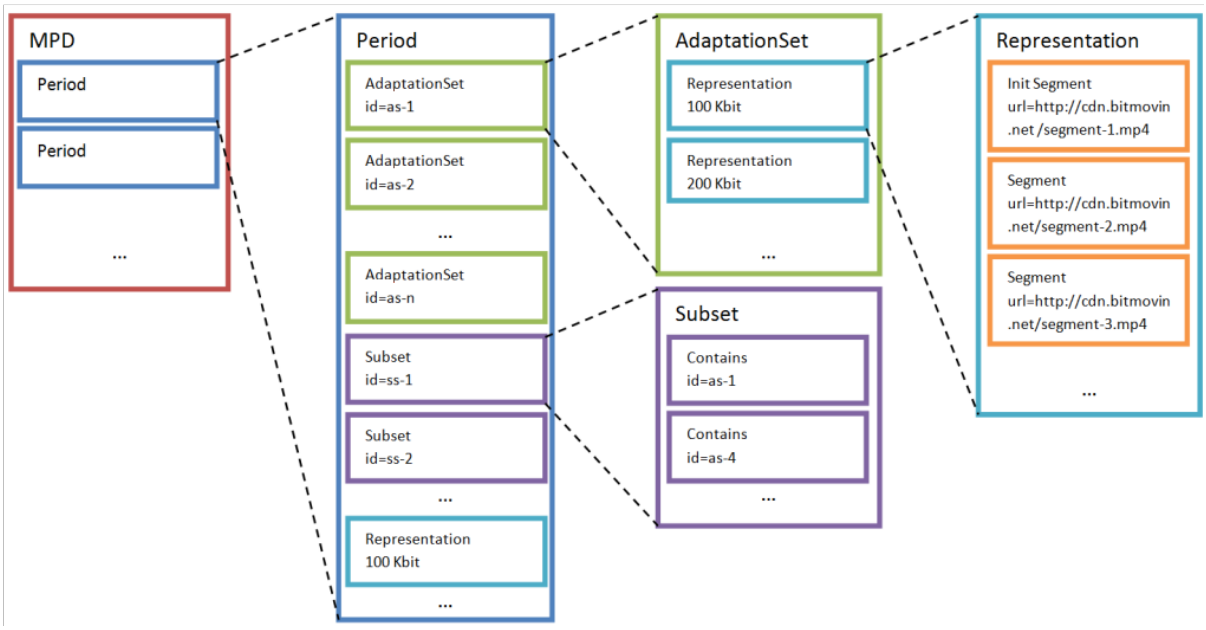


Figure 2.2: Structure of a DASH MPD file by Mueller 2019.

choose the best bitrate for the current network situation. The resolution of the representation is also useful. It is usually unnecessary to download segments at a resolution that is much higher than the resolution supported by the device. It is also important to check that the device supports the codec that is used.

Audio representations are very similar to video representations as they also define their codecs and bitrate. There are, however, some differences. Video-specific attributes like framerate, resolution etc. have no meaning for audio so they are not used. There are audio-specific attributes. For example, the *audioSamplingRate* attribute defines the sampling rate used in this representation. It is also possible to provide the client with additional metadata about the audio with *AudioChannelConfiguration* tags. This is, for example, used in more complex audio setups like Dolby Atmos surround sound [1].

Downloading video segments When a client starts to stream a video, it will need to begin by downloading the first segment. The first segment from a DASH stream is different from the rest because there is some initialisation data needed when using a codec like h.264, for example. The initialisation segment contains this data. The client's decoder needs to know this information to know how to decode the frames contained in the video segments. In our example, the location of this initialisation segment is defined in the segment template under the *initialization* attribute. If the client decides to start with representation "bbb_30fps_320x180_200k" for example, the location of the initialisation segment will be "bbb_30fps_320x180_200k/bbb_30fps_320x180_200k_0.m4v", relative to the location of the MPD file on the server. After that, the client will download the first data segment. This segment will be found at location "bbb_30fps_320x180_200k/bbb_30fps_320x180_200k_1.m4v". We know this because the first number of the first data segment is 1, as indicated by the *startNumber* attribute of the segment template. The next segment will then be "bbb_30fps_320x180_200k/bbb_30fps_320x180_200k_2.m4v" and so on until the total sum of segment durations has reached or exceeded the total duration of the stream defined in the root tag.

2.1.4 Flow of DASH streaming and representation switching

Starting a video stream begins not by downloading a video but by downloading the MPD file from the server. This is done using an HTTP GET request. Once the client has the MPD file, it can decide which representation to start downloading. When audio is involved in the stream, the client must also decide what adaptation set to choose for the audio stream.

[1] `https://ott.dolby.com/OnDelKits/DDP/Dolby_Digital_Plus_Online_Delivery_Kit_v1.5/Documentation/Content_Creation/SDM/help_files/topics/ddp_mpeg_dash_c_mpd_auchlconfig.html`

Most ABR algorithms will start with a low bitrate stream and will climb to higher bitrates when it does not encounter network problems. The client will again use HTTP GET requests to download the initialisation segment of the chosen video and audio representation. It will then also GET the first data segment of that representation. The downloaded data segments are placed in a video buffer. The purpose of the video buffer is to build up a reserve of downloaded segments for the client to playback. The buffer has a maximum size. Once this size is reached, the client will not download new segments until one of the segments is popped from the buffer. The client is playing back the segments in parallel for the consumer. Once the player has consumed an entire segment, the next segment will be popped from the buffer and fed to the video decoder for playback. The consumer does not notice the transition between segments from the same representation.

It is possible that when the consumer has viewed an entire segment, the next segment is not yet in the video buffer. The client has not managed to fill the video buffer adequately. This is usually caused by a decrease in throughput experienced by the client. The consumer now experiences stalling. Video playback has to wait until the client has managed to download the next segment. Stalling is detrimental to the QoE. It is also possible that the video segment has arrived, but the audio segment has not. At this point, the video can not be played back because the client still has to wait for the audio segment.

All the communication between the client and the server is done using simple HTTP requests. All the logic behind the adaptive streaming is done by the client using the ABR algorithm. Because of this, the server of ABR video streams can be a simple HTTP-capable webserver. It does not need to be aware that it is serving ABR content.

When the ABR algorithm decides it needs to switch to a different representation during the stream (for example, when the algorithm experiences a change on the network and wants to switch to a lower bitrate), it will select a representation from the MPD that it thinks is best suited for the current situation. The segments that have already been downloaded and are in the video buffer will remain there. The client will keep playing these segments as usual. The new segments that will enter this buffer will, however, be of the newly chosen representation. We explained why the video stream needs an initialisation segment. Whenever a client switches representation, the initialisation segment of this representation also needs to be downloaded. This is because the information that is in this segment is defined by the attributes of the representation like resolution, compression rate, framerate, codec etc. The client has to download this initialisation segment to be able to decode the data segments from the new representation. Because of this, it is better to switch to a different representation as little as possible to minimise the overhead. Switching to a different representation is also noticeable to the consumer of the video, and it affects the Quality of Experience (QoE) [Rodriguez et al. 2014]. This is another reason why an ABR algorithm needs to minimise the number of switches in representations.

```
1  <MPD mediaPresentationDuration="PT634.566S" minBufferTime="PT2.00S" profiles="
       urn:hbbtv:dash:profile:isoff-live:2012,urn:mpeg:dash:profile:isoff-live:2011" type="static" xmlns
       ="urn:mpeg:dash:schema:mpd:2011" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="urn:mpeg:DASH:schema:MPD:2011 DASH-MPD.xsd">
2   <BaseURL>./</BaseURL>
3   <Period>
4    <AdaptationSet mimeType="video/mp4" contentType="video" subsegmentAlignment="true"
       subsegmentStartsWithSAP="1" par="16:9">
5     <SegmentTemplate duration="120" timescale="30" media="$RepresentationID$/$RepresentationID$_$Number
         $.m4v" startNumber="1" initialization="$RepresentationID$/$RepresentationID$_0.m4v"/>
6     <Representation id="bbb_30fps_1024x576_2500k" codecs="avc1.64001f" bandwidth="3134488" width="1024"
          height="576" frameRate="30" sar="1:1" scanType="progressive"/>
7     <Representation id="bbb_30fps_1280x720_4000k" codecs="avc1.64001f" bandwidth="4952892" width="1280"
          height="720" frameRate="30" sar="1:1" scanType="progressive"/>
8     <Representation id="bbb_30fps_1920x1080_8000k" codecs="avc1.640028" bandwidth="9914554" width="1920
         " height="1080" frameRate="30" sar="1:1" scanType="progressive"/>
9     <Representation id="bbb_30fps_320x180_200k" codecs="avc1.64000d" bandwidth="254320" width="320"
         height="180" frameRate="30" sar="1:1" scanType="progressive"/>
10    <Representation id="bbb_30fps_320x180_400k" codecs="avc1.64000d" bandwidth="507246" width="320"
         height="180" frameRate="30" sar="1:1" scanType="progressive"/>
11    <Representation id="bbb_30fps_480x270_600k" codecs="avc1.640015" bandwidth="759798" width="480"
         height="270" frameRate="30" sar="1:1" scanType="progressive"/>
12    <Representation id="bbb_30fps_640x360_1000k" codecs="avc1.64001e" bandwidth="1254758" width="640"
         height="360" frameRate="30" sar="1:1" scanType="progressive"/>
13    <Representation id="bbb_30fps_640x360_800k" codecs="avc1.64001e" bandwidth="1013310" width="640"
         height="360" frameRate="30" sar="1:1" scanType="progressive"/>
14    <Representation id="bbb_30fps_768x432_1500k" codecs="avc1.64001e" bandwidth="1883700" width="768"
```

```
        height="432" frameRate="30" sar="1:1" scanType="progressive"/>
15    <Representation id="bbb_30fps_3840x2160_12000k" codecs="avc1.640033" bandwidth="14931538" width="
        3840" height="2160" frameRate="30" sar="1:1" scanType="progressive"/>
16   </AdaptationSet>
17   <AdaptationSet mimeType="audio/mp4" contentType="audio" subsegmentAlignment="true"
       subsegmentStartsWithSAP="1">
18    <Accessibility schemeIdUri="urn:tva:metadata:cs:AudioPurposeCS:2007" value="6"/>
19    <Role schemeIdUri="urn:mpeg:dash:role:2011" value="main"/>
20    <SegmentTemplate duration="192512" timescale="48000" media="$RepresentationID$/$RepresentationID$_$
        Number$.m4a" startNumber="1" initialization="$RepresentationID$/$RepresentationID$_0.m4a"/>
21    <Representation id="bbb_a64k" codecs="mp4a.40.5" bandwidth="67071" audioSamplingRate="48000">
22     <AudioChannelConfiguration schemeIdUri="urn:mpeg:dash:23003:3:audio_channel_configuration:2011"
         value="2"/>
23    </Representation>
24   </AdaptationSet>
25  </Period>
26 </MPD>
```

Listing 2.1: Example of an MPD file by DASH-IF 2022

2.2 ABR algorithms

The part of the DASH client that makes decisions about which segment to download is called the ABR algorithm. There are many different algorithms and many different implementations. But we can divide them in two major categories. The **rate-based algorithms** and the **buffer-based algorithms**. Generally, an ABR algorithm works with a video buffer. The buffer has a maximum size and contains a series of segments that have already been downloaded, but have not yet been played back. A buffer can help deal with fluctuations in throughput. Both types of algorithms that are explained here use a buffer.

2.2.1 Rate-based algorithms

The function of an ABR algorithm is to choose what segment to download and to ensure that segment is not too big to download in time. We could say that the most ideal segment would have a bitrate that is equal to or less than the throughput the client has. If we would tackle this idea naively, we could simply try to estimate the current throughput and then select the segment with the highest bitrate lower than our estimated throughput. This way, we will always download the segment with the highest quality while making sure the segment always arrives in time. Rate-based algorithms follow this line of thought. They try to estimate the current throughput as accurately as possible, but it gets a little more complex than this naive approach we described here.

The first problem to be solved when making a rate-based ABR algorithm is how to estimate the current throughput accurately. We can actually never know what our current throughput is, because you can only calculate how many bits arrived per second when those bits have already traversed the network. This is why we call it throughput *estimation*. We try to estimate the current throughput as accurately as possible. It is a good start to look at how long it took the previous segment to be sent from the server to the client. For example: if the previous segment had a bitrate of 500 kilobits per second, and the segments are all 2 seconds long, we know that the entire segment had a size of 1000 kilobits. If that segment took 0.5 seconds to download, we had an average throughput of 2000 kilobits per second during the download of that segment. We could use this as an estimation for the current throughput. The algorithm can now look through the manifest file to find what representation has the highest bitrate of all representations, while still being below 2000 kilobits per second. The segment of that representation would then be requested for download. However, by now, the network status might have changed. There could be a different amount of traffic on the network currently than there was when we downloaded the previous segment. If the current throughput now is higher than the throughput we estimated, we could have downloaded a segment of higher quality instead. If the current throughput is lower than the estimated throughput, we might not be able to download the segment in time. This can cause the video buffer to become empty, and we experience stalling. The video player has to wait until the segment is downloaded before it can continue playing back the video. This is not good for the Quality of Experience (QoE). The QoE is the perceived quality of a videos tream. There are many factors that affect the QoE of a video stream, and stalling is one of them.

Stalling is detrimental to the Quality of Experience [Duanmu et al. 2020]. Quality adaptation also has a negative effect on the QoE. This means that every time the ABR algorithm decides to download a

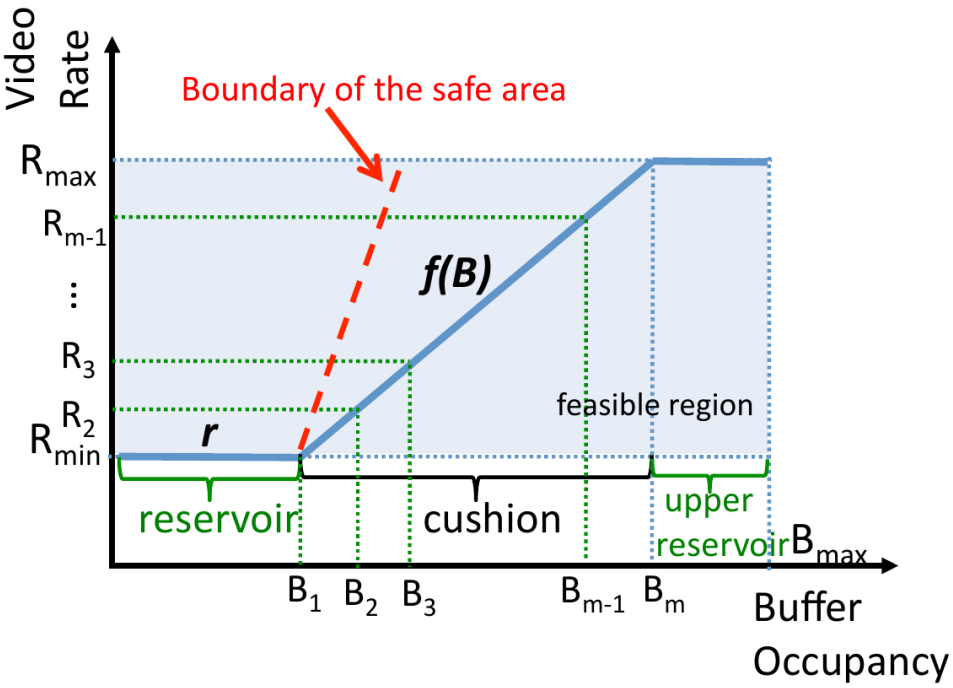


Figure 2.3: Rate map displaying BBA's behaviour by Huang et al. 2014a.

segment from a different representation than the previous segment was from, it will have a negative effect on the QoE. Thus, when designing an ABR algorithm, we want to ensure that we are not constantly changing qualities. For rate-based algorithms, this could be done by making sure we do not respond too much to short changes in throughput. For example, instead of calculating our throughput estimate by averaging the download speed of the previous segment, we could calculate the average download speed of the last five segments. Suppose there was a big fluctuation in throughput during only one of those segments. In that case, it will not have such a big impact on the estimated throughput and the ABR algorithm will be less likely to unnecessarily change representations.

2.2.2 Buffer-based algorithms

The most intuitive algorithm is not necessarily the best algorithm. Buffer-based algorithms take quite a different approach to make decisions. BBA is an example of a buffer-based algorithm [Huang et al. 2014a]. The researchers who created the BBA algorithm argue that using throughput estimation as the primary and only metric in an adaptive bitrate algorithm is unnecessary. They designed the BBA algorithm with the idea that throughput estimation is only used when it is really needed. Streaming is divided into two phases: the startup phase and the steady state. During the startup phase, streaming has just begun. The video buffer is empty, and there are not many metrics to use. In this phase, the ABR algorithm has to scale as quickly as possible to the steady-state phase by downloading segments to fill the buffer. In the steady-state phase, the buffer is filled. The researchers of the BBA algorithm argue that when we have reached the steady-state, the algorithm can decide what bitrate to use simply by looking at buffer occupancy and no longer has to rely on throughput estimation. Buffer occupancy is the amount of the video buffer that is filled.

Figure 2.3 helps explain how BBA works. The horizontal axis shows the *buffer occupancy*. This indicates how many segments are in the video buffer. If we are at B_{max} the buffer is completely filled. If we are at 0 the buffer is completely empty. The vertical axis shows the *video rate*. This is the quality of segments that will be downloaded. Function $f(B)$ is a *rate map*. This function defines what quality the algorithm needs to select for download depending on the current buffer occupancy. It maps buffer occupancy to video rates. When the algorithm has to download a new segment, it will feed the current occupancy to function $f(B)$. The output of that function is the quality that the algorithm chooses for the segment.

To make sure the video buffer is filled quickly when are close to an empty buffer, the *reservoir* is introduced. This is a phase where the algorithm will always choose the lowest available bitrate (R_{min})

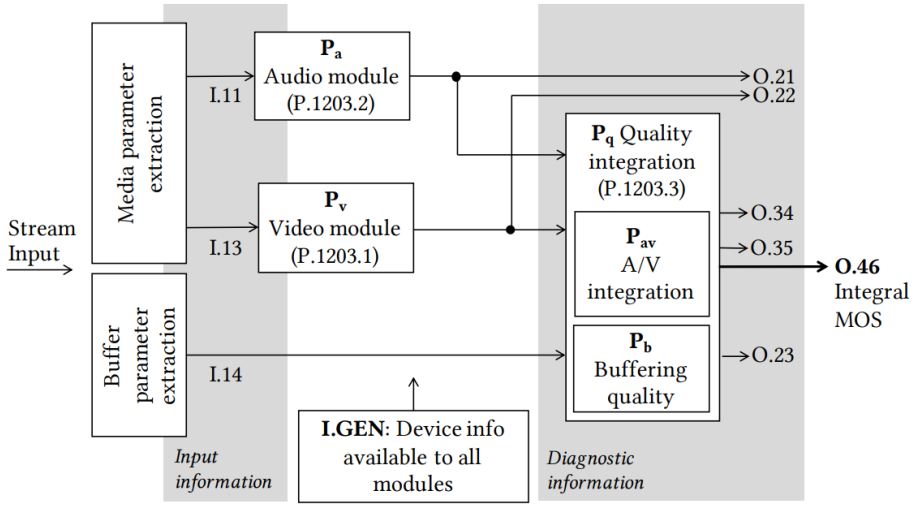


Figure 2.4: Input and output structure of P.1203, by Robitza et al. 2018.

MOS rating	Label
1	Poor
2	Bad
3	Fair
4	Good
5	Excellent

Table 2.1: Definition of Mean Opinion Score as described by Seufert 2019

until a certain amount of buffer occupancy (B_1) is reached. A similar technique is used with the *upper reservoir*. The algorithm will reach R_{max} before the buffer is completely filled. This is the upper reservoir, and it makes sure that when we have enough throughput to download segments at R_{max}, we will not suddenly switch to a lower quality when the buffer is not immediately filled back up after a segment is played back. The example in Figure 2.3 describes the reservoirs as static values that stay the same during the entire stream. In some versions of BBA, the edges of the cushion, B_1 and B_m, and thus the size of the reservoirs, can be variable depending on the size of the next segment [Huang et al. 2014b]. The size of every segment can be different, even when staying with the same representation, because video is usually encoded with a variable bitrate. Segments that have very static video can be compressed much more by the codec compared to segments that have a lot of movement, causing the segments to have differences in sizes. The bitrate that is stored in the MPD file of a DASH stream is, therefore, the average bitrate of the segments in the representations and not the exact bitrate of each segment.

What is described above is BBA-1, a version of the BBA algorithm that only uses buffer occupancy and does not use throughput estimation at all. In reality, BBA-2 is used. This algorithm initially uses throughput estimation when streaming is first started until the buffer is sufficiently occupied. After that, the buffer-based algorithm is used.

2.3 Evaluating ABR algorithms

The objective of an ABR algorithm is to provide the viewer with the best possible video streaming experience. How, then, can we evaluate the performance of new ABR algorithms? We could perform real-world tests with real users. The users could then be asked to evaluate the quality of the stream afterwards by giving scores from 1 to 5 on aspects like overall perceived quality, audio quality and video quality. This is called a Mean Opinion Score (MOS) and is usually described as in Table 2.1. This measurement, however, is subjective and time intensive to produce.

Alternatively, we could use objective measurements to define the Quality of Experience. For example, average bitrate, total stalling time, number of quality switches etc, could be metrics that we can use

to objectively compare the performance of ABR algorithms. The ITU has a standardised technique for quantifying the Quality of Experience of ABR streams: ITU P.1203. The authors of [Robitza et al. 2018] have validated that standard against real-world MOS tests and concluded that the metrics of P.1203 are very close to their MOS tests. They also released their Python implementation of the standard[2]. P.1203 outputs several metrics which can be seen in Figure 2.4. All output metrics are a number between 1 and 5 because they are meant to represent MOS ratings. Rating O.21 describes the audio quality, and O.22 the video quality. O.23 indicates the perceived stalling. A high rating would indicate that there was not much stalling perceived. Audio and visual ratings are combined in metrics O.34 and O.35. O.34 describes the audiovisual rating per time interval (for example, a score every second), while O.35 gives an overall audiovisual rating. Note that this rating only describes the quality of the audio and video that was received as input (bitrate, resolution etc.). It does not take into account metrics like stalling. Finally, O.23, O.34 and O.35 are combined into one final MOS rating: O.46.

2.4 QUIC

QUIC is the new transport layer protocol said to be a replacement of both reliable TCP communication and unreliable UDP communication. Some interesting differences to TCP are no HOL blocking and deep integration with TLS. To be able to understand the reason behind the creation of a new transport layer protocol and its functioning, we first need to look in the past to TCP and UDP.

2.4.1 Restrictions of transport layer protocols

The Transmission Control Protocol (TCP) and the Internet Protocol (IP) were originally developed in the 1970s as a protocol for a packet-switched network. The specification for TCP was described for the first time in 1974 in RFC 675 [Cerf, Dalal, and Sunshine 1974]. Since then, TCP has had many revisions described in subsequent RFCs. TCP is a reliable and fair protocol. But, not every use case has a need for such a protocol. Providing reliable and fair networking introduces some overhead that could be avoided for those use cases. Because of this, an alternative protocol was developed: User Datagram Protocol (UDP). This protocol provides fast, unreliable communication and was described in RFC 768 [Postel 1980] in 1980. UDP has never had a revision or extension. RFC 768 is the only RFC of UDP and is only a couple of pages long. This shows the simplicity of UDP.

When two devices communicate with each other across the internet, a sender's message passes through many devices before it reaches the receiving end. We call these devices the middleboxes [Robin Marx 2021]. A middlebox that runs on the third layer usually supports both TCP and UDP protocols. If we want to revise TCP with a new feature, we have to take into account these middleboxes. They might be old or never updated. If we release a new TCP extension, we cannot expect every middlebox on the internet to immediately support this new extension. It will take a long time before enough middleboxes support new TCP features for them to be usable. This issue is only really relevant for TCP because UDP has never had a revision.

2.4.2 QUIC protocol

Because it is so difficult to develop new features for TCP that will actually be used in reality, Google researched an alternative way to develop transport layer protocols. Because UDP is so simple, it has very little overhead. Google developed a new transport layer protocol from scratch: Google QUIC, on top of UDP. This development eventually led to a new standard by IETF, based on many of the principles of Google QUIC. The standard by IETF is also called QUIC and was first described in RFC 9000 [Jana Iyengar 2021; J. Iyengar and Thomson 2021].

QUIC introduces many of the features that TCP is known for, like reliable message delivery and congestion avoidance. There is also an unreliable extension for QUIC in RFC 9221 [Pauly, Kinnear, and Schinazi 2022], for the applications that do not require reliable transmission of messages to reduce overhead.

2.4.3 QUIC compared to TCP

QUIC and TCP have different approaches to several aspects of transport layer communication. In this section, two big differences are discussed: multiplexing and transport layer security.

[2] https://github.com/itu-p1203/itu-p1203

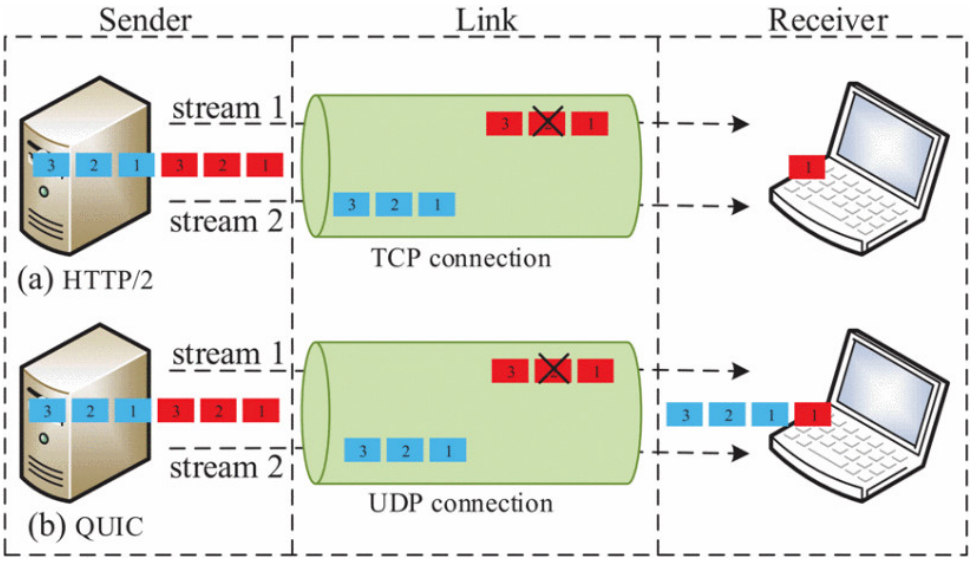


Figure 2.5: Visualisation of head-of-line blocking by Yu, Xu, and Yang 2017. a) shows multiplexing with HTTP/2 b) shows multiplexing with QUIC.

Multiplexing

The TCP protocol is designed to transport a single stream of data and deliver it reliably and in order. However, applications like HTTP, that use TCP as a transport protocol often need to download multiple resources. When a website is visited using HTTP, the client will probably need to download some HTML files, CSS files, JavaScript files, images etc. Instead of opening and closing TCP connections for every single resource, it would be beneficial to download multiple resources using a single TCP connection because every connection has a bit of overhead [Robin Marx 2021]. Transporting multiple resources or streams over a single connection is called multiplexing. TCP can be used for multiplexing since the introduction of SPDY and later HTTP/2 [Yu, Xu, and Yang 2017]. They allow multiple resources to be downloaded in "parallel". They are not actually downloaded in parallel since the packets are still a single stream that travel over the network. But, the stream of packets over the TCP connection can be from multiple resources.

There is still a big problem with multiplexing this way, which is known as head-of-line(HOL) blocking. HOL blocking occurs when a packet in the multiplexed stream is dropped. When a packet is dropped, the following packets cannot be processed by HTTP, because TCP wants to deliver packets in order. The packet that was dropped has to be retransmitted, while the packets that were transmitted after the dropped packet need to wait. In a multiplexed stream, the packets that need to wait can be from different resources than the resource the packet that was dropped. This means that other resources are blocked by the resource that was at the head of the line. This situation is shown in Figure 2.5 (a). Three packets from the red resource are sent over the TCP connection, and then three packets from the blue resource. When the second packet from the red resource is dropped, the following packets will not reach HTTP. Only the first packet will.

QUIC solves the issue of HOL blocking by handling multiplexing at the transport layer instead [Stenberg 2018; Reselman 2021]. A single QUIC connection can have multiple *streams* [Robin Marx 2021]. For example, when downloading the resources of a website, we can make a single connection with the webserver and then start a stream to download every resource independently. Features like in-order delivery are handled on the stream level, not on the connection level. Because these streams are handled independently, HOL blocking is no longer an issue. This is visualised in Figure 2.5 (b). When the second packet of the red stream is dropped, the receiver will also lose the third packet of the red stream. However, all the packets of the blue stream will arrive as if nothing happened, even though they were sent over the QUIC connection after the packet was dropped.

Security

Security was not taken into account when TCP was originally developed. TCP does not provide encryption itself. It has to be performed by a layer above. For example, the Transport Layer Security (TLS)

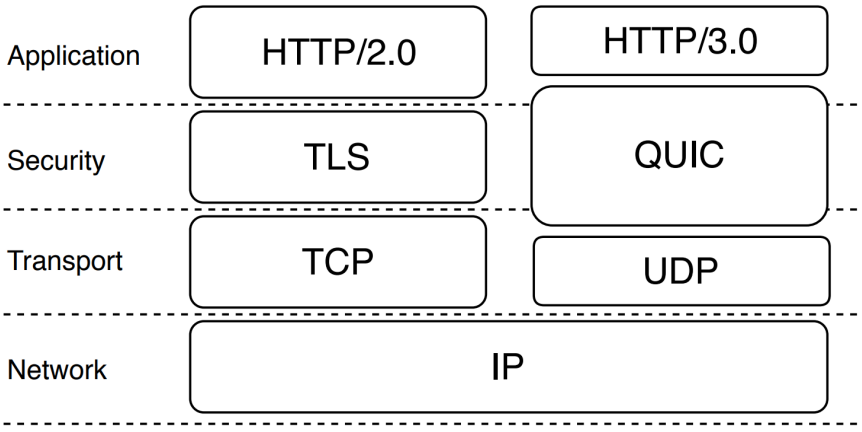


Figure 2.6: Protocol stack of a secure HTTP/2 connection compared to HTTP/3, by Mehdi Yosofie 2019.

protocol can be used for secure communication. A common use case of TLS is HTTPS, which is HTTP communication over a TCP connection secured with TLS. Figure 2.6 shows the protocol stack of the secure version of HTTP/2, which uses TLS and TCP, compared to what it looks like when QUIC is used.

QUIC takes care of encryption itself. It is deeply integrated with TLS [Robin Marx 2021]. Communicating with a server over QUIC always requires that server to have a valid TLS certificate. Because of this, any application that uses QUIC for communication will also always have encrypted communication.

2.4.4 HTTP/3

For version 3 of HTTP, QUIC was chosen as the transport layer protocol. This has several implications for the future of HTTP/3. Because data sent over QUIC is always encrypted, the need to do separate encryption in a higher layer, like with HTTPS, should no longer be necessary when using HTTP/3. Multiplexing is also handled differently from HTTP/2. With HTTP/3, we should no longer suffer from Head-of-Line blocking because QUIC treats every request as a separate stream instead of one single chain of messages that have to be acknowledged in order [Stenberg 2018; Reselman 2021].

2.4.5 qlog

When using TCP connections, it was easy to monitor and troubleshoot the network using packet tracing tools like Wireshark. This is a bit more difficult with QUIC, however, because QUIC encrypts the payload. We cannot determine what kind of communication layer we are tracing because the entirety of the application layer information is encrypted. For example, we cannot read information like HTTP headers when tracing HTTP/3 communication. To make monitoring and troubleshooting easier, qlog [Robin Marx, Lamotte, et al. 2018] was developed.

Chapter 3

Related work

As of the writing of this thesis, there has been no prior work that has implemented or tested cross-layer metrics sharing the way we describe in them this thesis. However, there is previous work that covers topics that relate to other aspects of this thesis. This chapter discusses related work about ABR streaming over QUIC and testing ABR streaming.

3.1 Problems with ABR over TCP

The general idea of ABR streaming techniques like DASH is to provide the client with video content that can be watched redardless of the clients' internet connection. But, ABR streaming is not perfect. There are studies discussing the problems of DASH and ABR in general. The research on this topic seems to conclude that DASH and TCP do not work well together. In this section, we will discuss some of the problems.

3.1.1 The inadequate window syndrome

The inadequate window syndrome, as Arye, Sen, and Freedman 2018 like to call it, is a problem caused by the sequential nature of DASH streaming [Akhshabi et al. 2012] and the double control loop. The problem causes a client streaming DASH video to underestimate their fair share of the network's bandwidth, causing a degradation in the Quality of Experience.

Most DASH algorithms try to fill a video buffer. It means that they will start to download video segments that fill the buffer. Once the video buffer has been filled, the algorithm stops downloading segments. In the meantime, the client is playing the downloading video segments and popping them from the buffer. Every time a segment is popped, the DASH algorithm will download the next segment to fill the buffer again. This phase of the algorithm is called the **steady-state** [Akhshabi et al. 2012]. During this phase, there will be many moments where nothing is communicated over the TCP connection. This silence causes the congestion window to shrink [Arye, Sen, and Freedman 2018]. When the next segment needs to be downloaded, the DASH algorithm will have an incorrect indication of the available bandwidth and might underestimate its fair share of the network.

3.1.2 The negative feedback loop

The underestimation of available bandwidth can cause a negative feedback loop. When a client underestimates its bandwidth, the DASH algorithm will choose to download the next segment in a lower bitrate. The smaller segments will cause the congestion window to grow less. The smaller congestion window will be cut down again during the next period of silence on the TCP connection [Arye, Sen, and Freedman 2018], shrinking it even more. This creates a negative feedback loop.

3.1.3 The double control loop

The origin of these problems is the double control loop. Both TCP and ABR are essentially doing the same thing, trying to decide the fair share of the network. TCP tries to increase the congestion window to allow for higher throughput but will shrink down when a lower throughput is experienced. An ABR algorithm will try to increase the bitrate of the segments it is downloading but will scale down to a lower

representation when a lower throughput is experienced. These are two control loops that try to do the same thing and do not always work well together. It is the cause of the inadequate window syndrome.

Now that a new transport layer protocol is being developed, it would be beneficial to prevent the same problems from occurring with ABR/QUIC. Unfortunately, this problem has not exactly been fixed. In the research by Bhat, Rizk, and Zink 2017, the authors test an ABR client using QUIC. They test multiple QUIC congestion control algorithms against ABR/TCP. Because the QUIC specification does not define its own congestion control algorithm [Engelbart and Ott 2021], these tests use TCP congestion control algorithms that have been adapted to work for QUIC. Thus, some of the problems that TCP and ABR streaming have persisted when using QUIC instead of TCP.

3.2 Cross-layer

Considering the networking branch of Computer Science, cross-layer means that something is done that covers multiple layers of the OSI model. Often this is used to cross between the transport layer and the physical layer.

3.2.1 Multipath

Sometimes, research uses the term cross-layer because their protocol or implementation uses multiple protocols in the same layer, to gain some kind of performance benefit. For example, we often see this in multipath algorithms. Examples of multipath research are Lu et al. 2015 and Deng et al. 2021. In Deng et al. 2021 the authors discuss a DASH client that uses both LTE and Wi-Fi to download video segments simultaneously to increase the throughput for the client. The idea is that the bottleneck for the client usually is the 'last mile'. The last mile is the final step over the entire network path a package has to take when being sent from a sender to a receiver. When using two interfaces to download these segments simultaneously, the total throughput will, in theory, be higher. In Lu et al. 2015 the authors discuss the same idea, but for TCP connections in general. Multipath is an interesting approach to improve throughput at the client side, but it is not the kind of cross-layer technology being researched in this thesis.

3.2.2 Cross-layer information sharing

More interesting for this thesis is cross-layer information sharing. This means that we are going to share information that came from one OSI layer and move it up or down in the OSI stack. Usually, this is a very hard thing to do, because TCP, for example, is very rigid and is baked into operating systems. Because of this, ABR algorithms that work in the application layer have to guess what the transport layer is doing. For example, an ABR algorithm would like to know the current throughput of the TCP connection. But because this cannot be communicated across layers an estimation needs to be made by the ABR algorithm.
When using QUIC, this is very different.

The authors of "Cross-Layer Metrics Sharing for QUICker Video Streaming" [Herbots et al. 2020] propose the idea of cross-layer metrics sharing to possibly improve video streaming. Traditional transport layer protocols are a "black box" usually provided by the operating system of a device. This means that for one operating system there is only a single implementation of a transport layer protocol. The only decision that an application developer needs to make is what protocol to use. Because QUIC is in essence a transport layer protocol that runs in user-space, there is not one single implementation for one device or operating system. There are many developers making their own implementation of the same protocol standard provided by the QUIC RFC [J. Iyengar and Thomson 2021]. For example, aioquic[1] is a QUIC package developed for Python, while quic-go[2] is a QUIC implementation written in Go. Because there are so many different implementations, it is important that they all work together. This is called interoperability. The QUIC interop runner[3] is one way to test the interoperability of these QUIC implementations.

In the paper "Same Standards, Different Decisions", the authors test 15 different QUIC+HTTP/3 implementations and test their support of the following features: Flow and Congestion Control, 0-RTT, Multiplexing and Packetisation. The reason why there are big differences between QUIC implementations

[1] https://github.com/aiortc/aioquic
[2] ttps://github.com/lucas-clemente/quic-go)
[3] https://interop.seemann.io

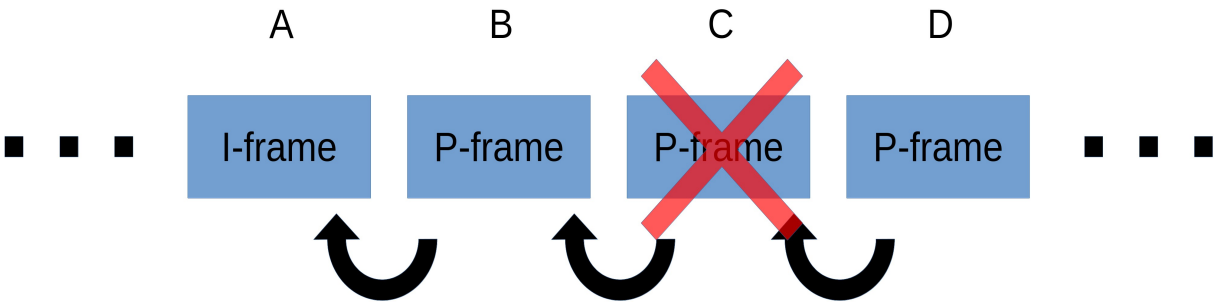


Figure 3.1: Example of losing a P-frame during a video stream. P-frames base pixel information on the previous frame.

is because the QUIC standard does not specifically define all aspects of the protocol. Some things are mandated by the protocol, but other aspects are open for interpretation by the developers.

3.3 Optimising ABR

VOXEL [Palmer et al. 2021] is a paper that handles ABR streaming very differently to other ABR algorithms. Usually, an ABR algorithm requests segments over HTTP and waits until they arrive. VOXEL handles streaming unreliably. They identify frames of a video segment that are not detrimental to video quality if lost. VOXEL streams those frames over an unreliable version of QUIC. This way, they try to minimise the amount of stalling by eliminating a lot of packet retransmits while retaining the video quality. Some type of frames, like I-frames, can never be dropped. Video compression techniques do not save all pixel data of every single frame, because this would create very large video files. Instead, there are frames like I-frames, which hold all information about a single frame and predicted frames like P-frames, which only store the difference in pixel values compared to the previous frame. For example, the order of frames in a video file could be: *IPPPIPPP*. In this example, every fourth frame stores all pixel data, while the other frames store the difference compared to the previous frame. If an I-frame were dropped from a video, the P-frames would have nothing to base themselves on. The P-frames would be useless. Thus, VOXEL makes sure I-frames are delivered reliably, while P-frames might be delivered unreliably. Figure 3.1 shows an example of a situation where a P-frame is lost during a stream. When frame C is lost, the video player will have no information about that one frame and will display frame B again instead of C. Losing one frame is not very noticeable. However, if the frame that comes after the frame we lost is also a P-frame, that P-frame will have based itself on the information of the frame that we do not have. In the example, frame D is based on frame C, but we do not have the information of frame C. Instead, we will have to use frame B instead. This will mean that the resulting image of frame D will not be exactly correct. If there are more P-frames after frame D, they will all carry that difference, and some artefacts might be visible in the images. According to the authors of VOXEL, the QoE lost by dropping a P-frame is less than what would have happened if the segment was delivered reliably.

The authors of VOXEL also describe the option of using segment abandonment for this ABR algorithm. Segment abandonment means that during the download of a segment, the ABR algorithm decides to abort that download, discard the downloaded data and start a new download from a lower representation. This is something that BOLA does [Spiteri, Urgaonkar, and Sitaraman 2016]. VOXEL does this a little differently. When a segment is abandoned, the part that was downloaded is retained. The video segment is then played back partially in the higher representation and partially in the lower. VOXEL also claims to be cross-layer, because they use different transport layer settings (reliable vs unreliable) depending on the type of frame the application layer is sending. In a way, the application layer has knowledge of the transport layer. In Chapter 1, we defined three distinct ways of using cross-layer information sharing. VOXEL qualifies as a cross-layer implementation of type ii.

Days of Future Past (DoFP) [Lorenzi et al. 2021] is a paper that also tries to optimise ABR using QUIC. Their optimisation attempts to improve the QoE by not only trying to download the highest possible bitrate for the current segment, but also redownloads previously downloaded segments if they are of a lesser bitrate. This happens in moments when the client has enough available throughput that it can use some of that throughput to download the old segment again. Because of this, the number of quality switches is lowered and this improves the QoE.

DoFP uses some of the features that QUIC offers that other transport protocols do not. The first one

of these features is the ability to multiplex streams over one QUIC connection. When the client decides to redownload a previous segment, the next segment is also still being downloaded. All this happens over a single QUIC connection with the content server. If this had been implemented with TCP, a second connection would have to be opened with the content server,

3.4 QUIC

The first IETF QUIC RFC was only recently released. Because of this, much of the research about ABR streaming over QUIC predates RFC 9000. However, the work that has been done with QUIC is very interesting. In this section we will discuss literature that covers ABR over QUIC and QUIC in general.

3.4.1 Streaming frameworks

Streaming frameworks are pieces of software that make it easier for researchers and developers to test many aspects of ABR streaming like ABR algorithms, different transport layers, different settings, network scenarios etc. An example of such a framework is goDASH [Raca, Manifacier, and Quinlan 2020], which allows streaming DASH over QUIC. This is a framework for a DASH client written in Go. There is already a second version available [O'Sullivan, Raca, and Quinlan 2020]. This framework provides a headless DASH client. This means that it simulates all the steps a real DASH client would do to watch a video stream, but there is no actual video played. Testing frameworks often work like this because there is no need to view the video content that is being streamed. If we want to know the QoE of the video stream, we will use more objective ways to determine that.

GoDASH provides the option to test both TCP and QUIC streams. It also provides numerous other options for testing purposes like assistance QoE determination using, for example, ITU P1203. The framework separates itself from other frameworks like TAPAS, AStream, and dashc because they implement more ABR algorithms and it supports QUIC [Raca, Manifacier, and Quinlan 2020]. This can be very useful if we want to test our own versions of these algorithms. Because goDASH is open-source, it is possible to add your own algorithms. Overall, goDASH has an impressive set of features that are useful for testing ABR streaming both over TCP and QUIC.

Unfortunately, goDASH is not perfect. There are some known issues[4] with the framework. GoDASH should not be used for testing without fixing these issues because they influence the streaming behaviour gravely. One of the biggest issues with the framework is that it creates a new QUIC client for every segment, and thus the congestion window has to be rebuilt for every segment. Research that uses goDASH for testing the performance of QUIC might not have reached the full potential of QUIC unless they fixed this issue. In Chapter 4 we talk more about the issues we faced while using goDASH.

3.5 Quality of Experience

Measuring the quality of a stream can be done in many different ways. We see this in research as well, the discussed papers in this chapter use various different techniques to determine the QoE of their implementations. We already discussed one way of measuring QoE in Section 2.3, where we explained how ITU P.1203 works. This is also one of the techniques goDASH supports. VOXEL measures their performance in a very different way. Because they are dealing with frame drops caused by the unreliable transport of predicted frames, they use a technique called Structural Similarity Index Measure (SSIM) [Nilsson and Akenine-Möller 2020], which looks at the actual video data of the arrived frames instead of looking at segment quality and delivery timing. The authors of DoFP make use of several basic metrics to determine the performance of their algorithm, like stall time, number of quality switches and the chosen quality. They also use ITU P.1203 to make an overall assessment of their algorithm. Some authors decide to make their own QoE estimation [Hoßfeld et al. 2013; Stohr et al. 2016].

3.6 Observations

We see that DASH streaming is a capable technique for streaming video in multiple bitrates. We do however also recognise that there are still significant improvements that can be made. We learned in Section 3.1 that DASH and TCP do not always work well together. A logical conclusion would be that

[4] https://github.com/JorisHerbots/godash-qlogabr/blob/master/PROBLEMS.md

TCP should be improved to work better with video streaming techniques. The problem, as we discussed in section 2.4.1, is that it is very difficult to make changes to the TCP protocol.

QUIC is the new big transport protocol, and a lot of research about it is being conducted. Because QUIC is now standardised, we can investigate if the QUIC protocol could be used to improve ABR streaming. Even though QUIC has its own problems streaming DASH videos [Bhat, Rizk, and Zink 2017], the current situation, with QUIC being implemented in user-space and the existence of tools like qlog, provides us with the perfect opportunity to see if ABR algorithms can be improved using cross-layer metrics sharing using QUIC. We see that QUIC is used for a variety of different research. QUIC performance is compared to that of TCP [Bhat, Rizk, and Zink 2017] for ABR streaming, and research using the cross-layer capability also exist [Palmer et al. 2021]. However, there seems to be no research about the use of cross-layer metrics sharing between QUIC and the application layer.

Measuring the performance of a new ABR algorithm correctly and transparently is important but not easy. Papers presenting new ideas for ABR streaming sometimes give very little information about how their algorithm is tested and evaluated. Questions like: "What datasets were used?", "How many tests were performed?", "Does the algorithm behave differently depending on what test scenario is used?" should be clearly answered when discussing results. However, simulating network traffic is not trivial. Network traffic can be simulated by creating artificial throughput variations and packet loss. That can be based on either arbitrary values or based on datasets that represent real-world situations. It can be beneficial to look at both artificial scenarios and real-world scenarios. For comparing the QoE of different implementations, various techniques are used. However, the ITU P.1203 metric is standardised and is also used by many researchers [Raca, Manifacier, and Quinlan 2020; O'Sullivan, Raca, and Quinlan 2020; Lorenzi et al. 2021; Bermudez et al. 2019; Robitza et al. 2018; Guzmán Castillo, Arce Vila, and Guerri Cebollada 2019] and should thus be a good metric for evaluating ABR algorithms.

Chapter 4

Cross-layer ABR implementations

To answer the research questions, we need to set up an environment where we can pass messages from the transport layer to the application layer. To answer the first question, we implement a variation of a simple rate-based ABR algorithm.

It is interesting to note that multiple papers researching ABR optimisations try to implement some way of segment abort. We have seen Palmer et al. 2021 and Spiteri, Urgaonkar, and Sitaraman 2016 implement cutting off segments before their download is completed. We can see if we can use cross-layer information sharing to implement something similar. We could use the results of such an implementation to answer research question 2.

4.1 Setting up cross-layer metrics sharing

For the implementation of this cross-layer concept, several different existing implementations were used. Firstly, for the transport layer we used the quic-go [Clemente 2022] implementation. As the name suggests, it is a QUIC implementation written in Go. We forked this repository and made changes to the initialisation of the connection. Our fork of the quic-go implementation features a messaging channel where qlog messages are pushed to and can be read in the application layer.

Secondly, for the application layer we used the goDASH implementation [Raca, Manifacier, and Quinlan 2020]. This is a DASH client made purely for testing and also written in Go. The client features a headless implementation of a DASH client, meaning that the video that is streamed to the client is not actually rendered. The client simply downloads the segments as if it would render the video. During the stream, data can be gathered about the Quality of Experience. It is useful for testing different ABR algorithms in a controlled environment. The authors of goDASH also created the goDASHbed [O'Sullivan, Raca, and Quinlan 2020]. This is a web server that can be used together with the goDASH client for easy testing. But this was not used in this thesis. Under the hood, goDASH also makes use of quic-go when QUIC is used instead of TCP. This made is relatively easy to swap the original quic-go implementation with our own fork.

In our fork of goDASH, a cross-layer message channel is created and passed to the transport layer (quic-go). The transport layer will then use that message channel to pass its cross-layer messages. This all happens in separate threads to make sure it does not interfere with the normal workings of the transport layer. Our fork is actually not based on the original goDASH implementation by Raca, Manifacier, and Quinlan 2020; O'Sullivan, Raca, and Quinlan 2020, but is based on a fork by Mike Vandersanden and Joris Herbots[1]. The original goDASH implementation was a research product as well and not a fully maintained framework. Because of this, there are several problems with the implementation that need to be addressed before we can use it for our own research. The problems and their fixes are addressed in the repository[2]. The biggest problem that was addressed was the fact that for every segment request a new QUIC connection was established. This is obviously not how ABR clients are supposed to work since starting a new client for every segment means that for every segment the congestion-control algorithm has to start from scratch. We want the client to reuse the existing QUIC connection instead.

If we want tot test the ABR client over QUIC we need a QUIC (or HTTP/3) capable webserver. For initial testing, we again used quic-go to serve a folder containing an MPD and segments of the Big Buck Bunny open-source video. In our tests, we use self-signed TLS certificates on the webserver. Usually, a

[1] https://github.com/JorisHerbots/godash-qlogabr

[2] https://github.com/JorisHerbots/godash-qlogabr/blob/master/PROBLEMS.md

browser would not allow these certificates because it cannot verify their authenticity. But, because we are working in a testing environment, we can make the transport layer implementation skip the TLS verification step. This will help us a lot during the testing phase.

4.2 Average ABR algorithm

The goDASH client implementation provides several different ABR algorithms from rate-based to throughput-based. As explained in Section 2.2.1, rate-based algorithms are usually less complex. Because of this, they are a good place to validate if using cross-layer can actually work at all. We start with an algorithm that we call the Average algorithm. This algorithm tries to estimate throughput by keeping track of all segments received and the time it took to receive them. This is then used to calculate an average throughput over all segments. We can describe this using the following equation:

$$throughput = \frac{totalBitsReceived}{totalDownloadTime} \tag{4.1}$$

Every time a new segment needs to be selected by the ABR algorithm, this estimation is calculated. The algorithm will then select a representation from the available representations that has a bitrate that is smaller but closest to this estimate. The implementation goDASH tackles this calculation a little bit different. For every segment received, the throughput during the download of that segment is calculated by dividing its size by the time it needed to download that segment.

$$segmentThroughput = \frac{segmentSize}{segmentDownloadTime} \tag{4.2}$$

This throughput is stored in an array every time it is calculated. When the average throughput is requested, we can simply calculate the average of the throughputs in this array.

$$throughput = \frac{\sum_{n=1}^{numberOfSegments} segmentThroughput(n)}{numberOfSegments} \tag{4.3}$$

When implementing these ABR algorithms, it is important to keep track of what units these values are calculated. The bandwidth of a segment is defined in bits per second (bps) in a DASH MPD. The body of an HTTP package is usually read in bytes. We have to make sure that we work with the same units. The easiest way to do this is to convert everything to bits.

4.2.1 Cross-layer metrics

The only metric we need for this algorithm is the throughput metric. We can extract this metric from the qlog messages that our QUIC implementation communicates with our DASH implementation. Qlog messages feature a *RawInfo* section that gives information about the amount of data in a message. We can for example see this in the *EventMessageReceived* and the *EventMessageSent* events. The *RawInfo* section has a child *length* and can have a child *payloadlength*. The *length* child indicates the total length in bits of the message that was sent or received, including the headers. The *payloadlength* child indicates only the length in bits of the payload that was sent. These are the metrics we can use for estimating throughput on the transport layer.

```
RawInfo = {
    ; the full byte length of the entity (e.g., packet or frame),
    ; including headers and trailers
    ? length: uint64

    ; the byte length of the entity's payload,
    ; without headers or trailers
    ? payload_length: uint64

    ; the contents of the full entity,
    ; including headers and trailers
    ? data: hexstring
}
```

Listing 4.1: RawInfo definition from the qlog specification, R. Marx, Niccolini, and Seemann 2022

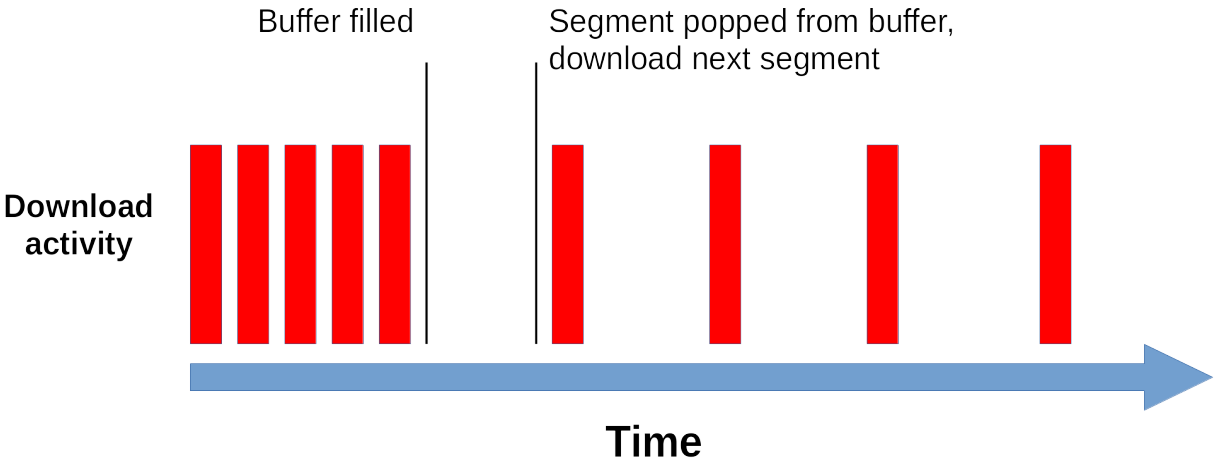


Figure 4.1: Visualisation of the on and off periods of network activity from an ABR client. During every red block, a segment is being downloaded. Between the segments is some idle time, because the ABR algorithm needs a little time to perform calculations. When the video buffer is filled, the next segment will only be downloaded when an entire segment is taken out of the video buffer for playback. At this point in the video stream, the periods of silence between the segments will be larger.

4.2.2 Measuring time

The qlog metric provides us with the number of bits that arrive over the network. What we also need to calculate the throughput is the time it took that data to arrive at the client. It is not sufficient to simply measure the total time the client has been streaming video because the client is not requesting video segments all the time. Figure 4.1 visualises this. At the start of a video stream, the client will download segments back-to-back from the server until its video buffer is filled. When an entire segment is played-back, the segment is popped from the video buffer. At that point, the client can download the next segment. Because of this, there can be large moments of silence on the connection. It would not be correct to keep measuring the time during these periods of silence because no data is being downloaded.

Every video segment is requested with an HTTP GET request. Once we send out a request, we have to start measuring time. When the packet arrives, we stop measuring time. If we divide the total amount of data we have downloaded this way by the elapsed time we measured, we have estimated the throughput during the download of that segment. In an application layer-only implementation, we would only be able to request this information after a segment has been downloaded. But, because we are implementing this using cross-layer information sharing, we can actually request the estimated throughput at any point during the download of a video segment. This feature can be useful when we want to develop the algorithm for testing research question 2 of this thesis. For now, we will simply request this information at the same time the original application layer average ABR algorithm would request this.

4.2.3 Problems encountered with goDASH

During testing, a problem with the goDASH implementation was encountered, other than the problems we already discussed at the beginning of this chapter. The base implementations Average and BBA underperformed enormously. Figure 4.2 shows the buffer occupancy of the Average algorithm streaming a video in challenging network conditions. The vertical lines indicate every moment a new segment arrived. For this test, an initial buffer size of 2000ms was used, and the segments are all 1s long. An initial buffer size is a buffer occupancy that needs to be achieved by the client before playback of video starts. In case of an initial buffer size of 2000ms, the client will first download 2 segments to fill the video buffer initially, and then the playback of the video will start. We can see this happening in Figure 4.2. The buffer occupancy quickly rises to 2000ms after receiving two segments. After that, the video buffer starts emptying until a little after 40000ms; the buffer has only one segment remaining. The measurements are taken every time a segment arrives. When there is only one segment in the buffer at

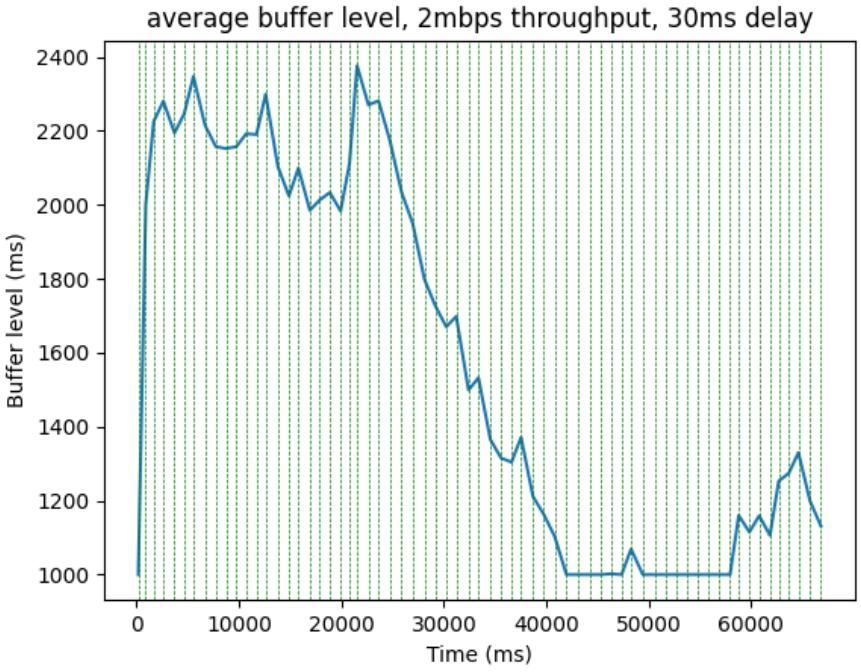


Figure 4.2: Buffer occupancy of the Average algorithm during a 60 second video stream, using ns-3 with a constant 2mbps throughput, 30ms delay and a packet queue of 1. A packet queue of 1 means that the simulator can only keep hold of one packet at a time to introduce delay, packet loss and rate limitation. For this test, a packet queue of 1 was chosen because large queues can introduce unrealistic behaviour, causing a very high average RTT. The buffer level (or buffer occupancy) indicates the amount of video remaining in the video buffer in milliseconds. The vertical green lines indicate every time a new segment has arrived at the client. Buffer occupancy is measured every time a segment arrives.

the time a measurement is taken, it means that the buffer was empty before the segment arrived. In other words, the player experienced stalling. We can see a lot of stalling happening in the second half of the test. The video buffer never reaches its original occupancy again.

This result is unexpected because the algorithm estimates the throughput as we described in Equation 4.3. The scenario used for testing in Figure 4.2 was a constant throughput of 2mbps and a constant delay of 30ms. An algorithm that calculates the average throughput of all segments it ever received should have no problem with this scenario. The only moment some stalling could occur is if there is a large variation in the sizes of the segments. Not every segment of the same representation has the exact same size. Depending on the scene in the video segment and the encoding that was used, every segment will have a little difference in size. The bitrate that is shown in the MPD is only an average of all the segments in the representation.

However, the variation in sizes is not that extreme and could not be the only cause of these large moments of stalling. Thus, we have to look further into possible explanations. During our tests, we log QUIC events using qlog. We can then load these files into qvis to get a good overview of what was happening on the transport layer. Figure 4.3 shows a congestion graph of the Average algorithm. In Figure 4.3a we can see a trace of about 9 segments being downloaded. Between every segment, there is a pause before segments start being downloaded again. Figure 4.3b zooms in on two segments to make it clear how large these gaps between the segments are. They are about 200ms. That is quite a long time, and it is time that the algorithms of goDASH do not take into account during their calculations. For reference, the same scenario was also tested with dash.js[3]. Figure 4.4 shows the gaps between segments measured with dash.js. They are a lot smaller and are about 80ms. This is acceptable for a browser implementation. The large difference between the gaps of goDASH and dash.js could be explained if the code that is executed between every segment request is inefficient.

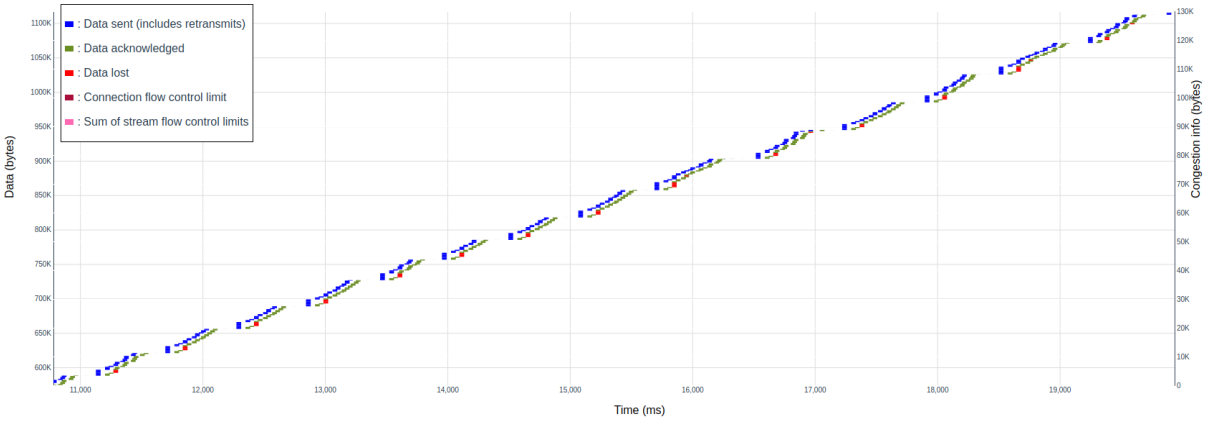
The extreme delays between segment downloads were caused by the ITU P.1203 Python script that we also mentioned in section 2.3. The goDASH framework has this metric integrated to help evaluation of algorithms. The problem is that the framework calculates the MOS rating for every segment, after every segment download so that the score can be displayed in real-time. The script takes a while to fully execute. This was the cause of the delay. We can, however, still use the ITU P.1203 metric for the evaluation because the script can also be executed after the test has been completed. We only need some metadata to be written to a file during the test. Figure 4.5 shows the delay between the download of two segments when the ITU P.1203 script is not used. The delay is now only about 60ms, which is a lot better than the previous tests.

4.2.4 Evaluation of cross-layer implementation

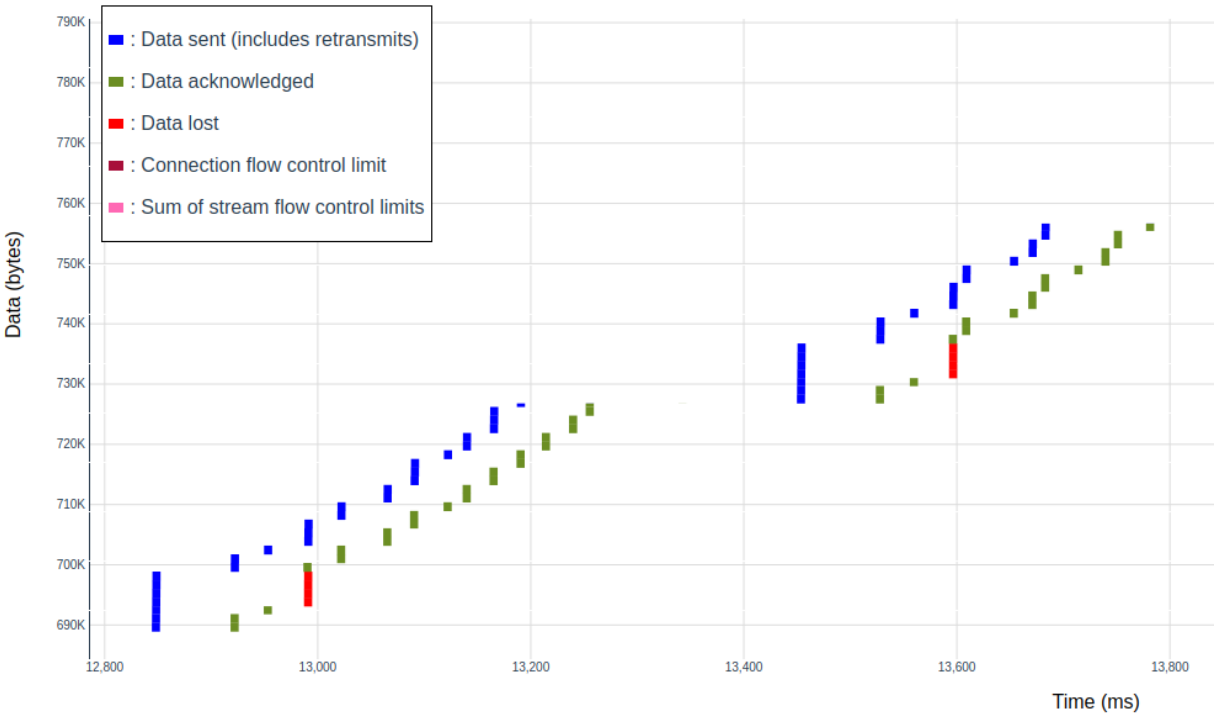
In chapter 1, we hypothesised for research question 1 that replacing information used in existing ABR algorithms with information gathered on the transport layer will not make a noticeable performance gain. To confirm or deny hypothesis 1 for the Average algorithm, we will compare the performance of AverageXL (Average Cross-Layer), which is our cross-layer implementation of Average, to the original algorithm. We will look at buffer occupancy, chosen representations and the QoE metrics of ITU P.1203. The base algorithms implemented by goDASH, like the Average algorithm, calculate throughput by measuring segment size and the delivery time of the segments. However, segments are also written to the disk, and, this time, is included in the measurements. AverageXL does not have this problem since the times are measured on the transport layer. To make the comparison as fair as possible, writing segments to the disk was disabled for these tests.

First, we measured the performance of both algorithms in a very basic simulation with a constant throughput of 2mbps, a constant delay of 30ms and a packet queue of 25 using ns-3. Figure 4.6 shows a comparison of the buffer occupation and the representations the algorithms selected. Secondly, the performance was measured in a more realistic scenario using a cellular dataset by Akamai. Every test that is run on the dataset is a bit different because the simulation does not always start at exactly the same moment, and extra packet loss is also introduced semi-randomly. Because of this, 20 tests were performed for each algorithm. The buffer occupation selected representation and ITU P.1203 O.46 is shown in Figure 4.7.

[3]https://github.com/Dash-Industry-Forum/dash.js

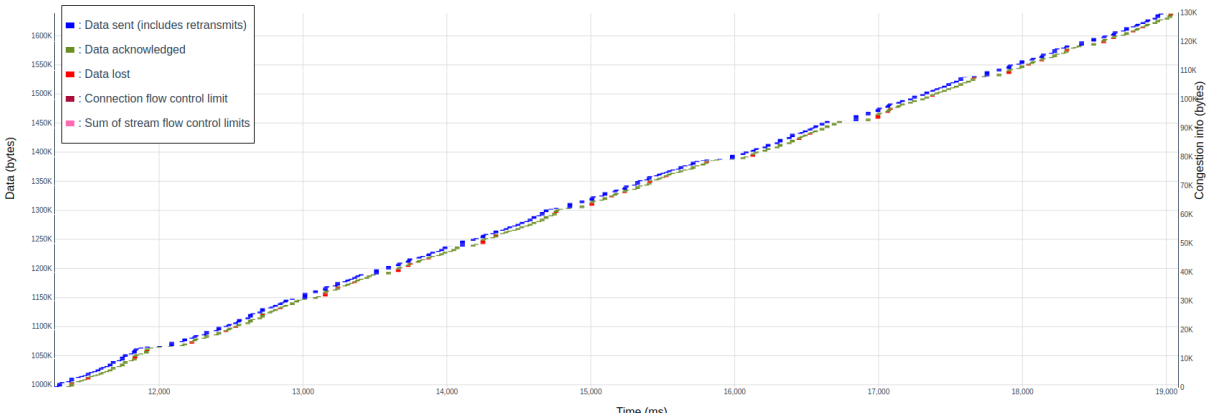


(a) Graph showing a series of segment downloads from the server side.

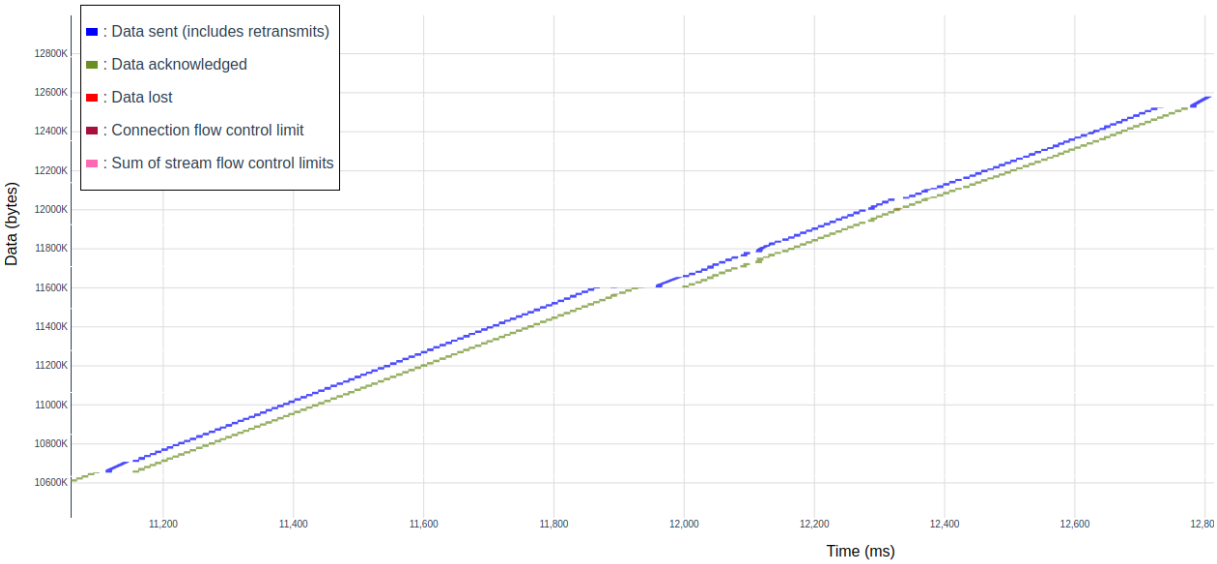


(b) Enlarged graph showing the large gap between two segments. This is about 200ms. This is a lot longer than it should take for an ABR algorithm to do the calculations and start downloading the next segment.

Figure 4.3: Congestion graph of another test using the same scenario as Figure 4.2 (a network simulation with ns-3 using a constant throughput of 2mbps, a constant delay of 30ms and a packet queue of 1), but using the Average algorithm. The congestion graph shows how much data is in flight (blue) at what point in time and when they are acknowledged by the receiver (blue).



(a) Graph showing a series of segment downloads from the server side.



(b) Enlarged graph showing the large gap between two segments. The gap that dash.js has between its segments is a lot shorter than what we see with Average of goDASH in Figure 4.3.

Figure 4.4: Congestion graph of another test using the same scenario as Figure 4.2 (a network simulation with ns-3 using a constant throughput of 2mbps, a constant delay of 30ms and a packet queue of 1), but using dash.js. The congestion graph shows how much data is in flight (blue) at what point in time, and when they are acknowledged by the receiver (blue). The gap is only about 80ms, which is an acceptable calculation time for an ABR algorithm in the browser.

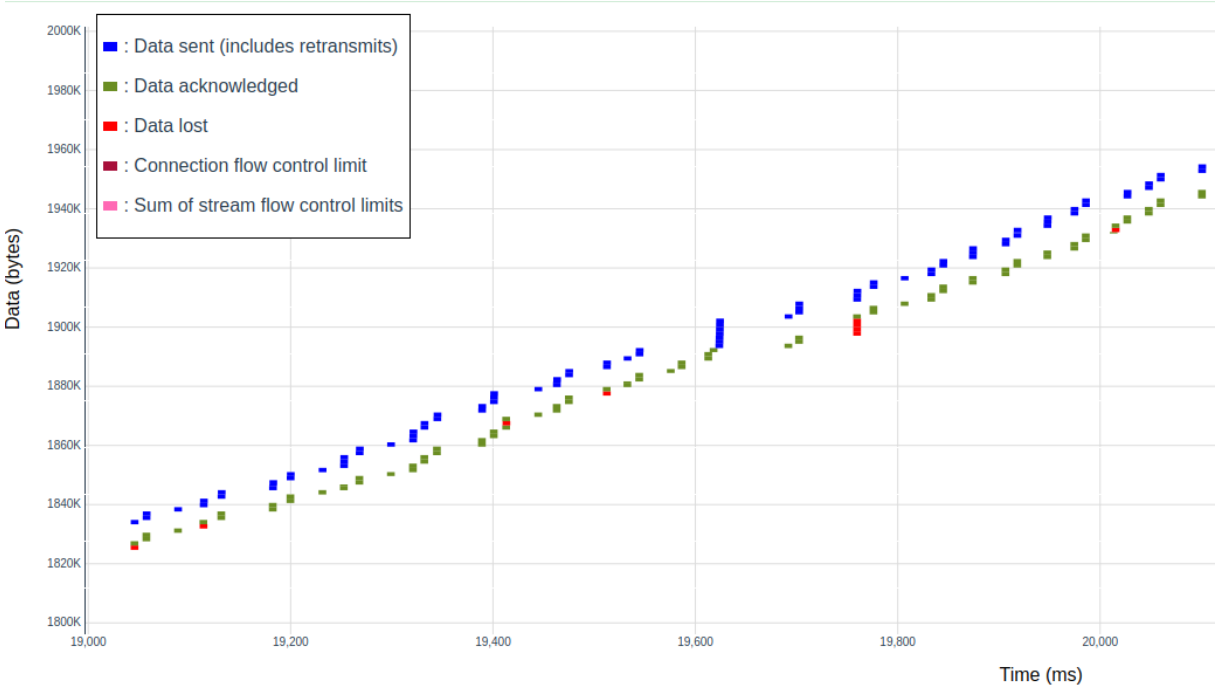


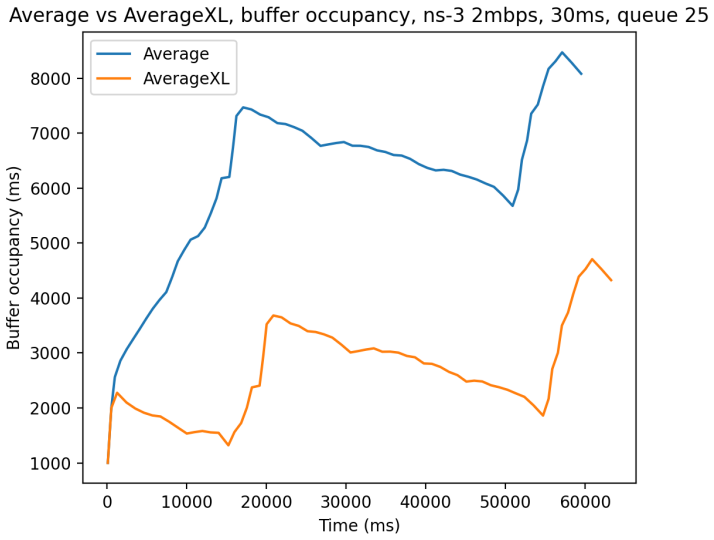
Figure 4.5: Congestion graph showing the smaller gaps between segments when the ITU P.1203 script is not executed for the Average algorithm of goDASH. The congestion graph shows how much data is in flight (blue) at what point in time, and when they are acknowledged by the receiver (blue). The gap is only about 60ms now, which is a lot better than the 200ms we had previously.

4.2.5 Results

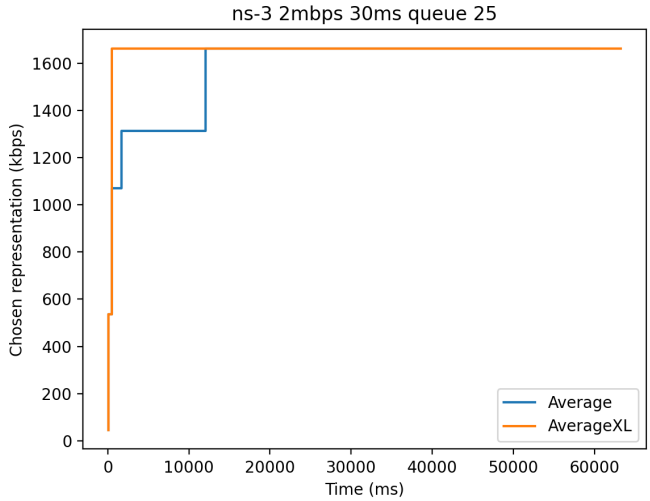
The performance of both algorithms is not exactly the same. In fact, AverageXL seems to perform worse in both scenarios compared to Average. Figure 4.6b shows the representation both algorithms chose during the test. We can see that AverageXL scaled to the highest possible representation in a 2mbps scenario very quickly. Average did not. Because of this, the buffer occupation is much lower with AverageXL for the first 15000ms compared to Average in Figure 4.6a. The video buffer never became completely empty, but it was very close for AverageXL in this particular scenario. Choosing the high representation early on was risky and could have caused buffering. After 15000ms, both algorithms chose the representation of 16000kbps, as shown in Figure 4.6b. After that point, the buffer occupation of both algorithms show the same behaviour, only with an offset because of the earlier switch to a higher representation by AverageXL.

It is remarkable to see that even though both algorithms chose the same representation of 1600kbps after 15000ms, the buffer occupation is still fluctuating heavily. In fact, we can point out two distinct peaks in Figure 4.6a. The first peak happens just after 20000ms and the second at around 55000ms. The reason why this is remarkable is because the network simulation for this test used a constant throughput limitation of 2mbps and a constant delay of 30ms. In theory, if we keep downloading segment from the same representation, we should not see this kind of variation. We investigated this further by looking at the segment sizes of the representation with a throughput of 1600kbps. This throughput is how the representation is advertised in the DASH MPD. This throughput is what ABR algorithms base themselves on when choosing what representation to download. However, the throughput advertised in the MPD is only an average throughput of all the segments in that specific representation. The bitrate for every single segment will be different because of variable bitrate encoding (VBR) [Qin et al. 2018]. When an audio or video segment is encoded with VBR, more bits will be used in segments with a lot of movement. Segments with little to no movement can be heavily compressed and thus do not need as many bits.

Figure 4.8 shows a plot of the segments that AverageXL received over time, just like in Figure 4.6b. The data for this plot was also taken from that same test. Instead of showing the bitrate reported by the representation in the DASH MPD on the plot, this figure shows the size in kilobits of the actual segments that were downloaded by AverageXL. The variation in sizes is very high. We can see dips in the graph, meaning that in some periods of the stream, the segments were a lot smaller than usual. The dips of Figure 4.8 match with the peaks of Figure 4.6a. Because the segments are smaller, they arrive faster and

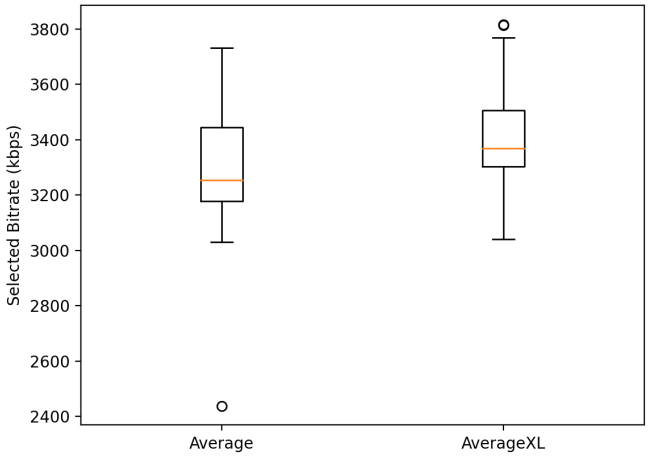


(a) Comparison of the buffer occupancy of Average and AverageXL with an ns-3 simulation.

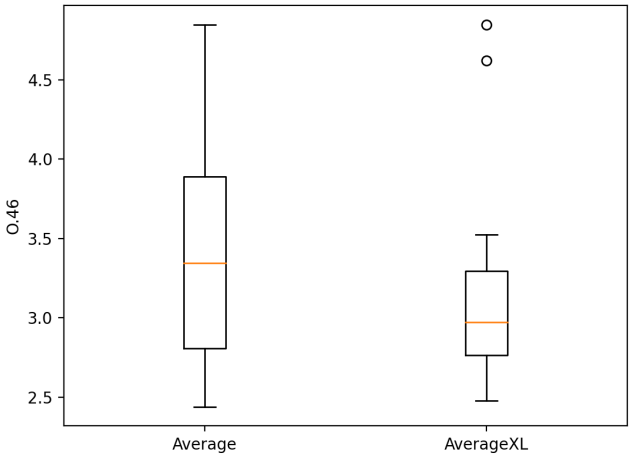


(b) Comparison of the chosen representation between Average and AverageXL with an ns-3 simulation. The representation is the quality level of segments the algorithms choose to download. They are displayed in kbps, according to the bitrate of the chosen segment. AverageXL (in orange) chose a representation with a bitrate of 1600kbps earlier than Average (in blue).

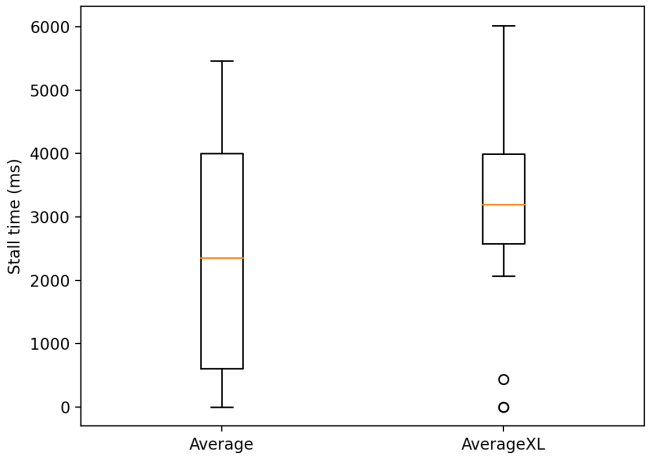
Figure 4.6: Comparison between Average and AverageXL using an ns-3 network simulation with a constant throughput of 2mbps, a constant delay of 30ms and a packet queue of 25. We visualise both the buffer occupancy (buffer level) over time and the representations the algorithms chose over time.



(a) Boxplots of the chosen bitrates by Average and AverageXL. The bitrate is shown in kbps and it defines the bitrate of the segments the algorithm chose to download.



(b) Boxplots of ITU P.1203 O.46 score of Average and AverageXL. The score represents how real users would rate the quality of this stream between 1 and 5. The higher the score, the higher the perceived quality is.



(c) Boxplots of the total amounts of stalling the clients experienced using Average and AverageXL. Fewer seconds spent stalling is desirable.

Figure 4.7: Comparison between Average and AverageXL using a simulation of a cellular network provided by Akamai. 20 tests were performed for each algorithm.

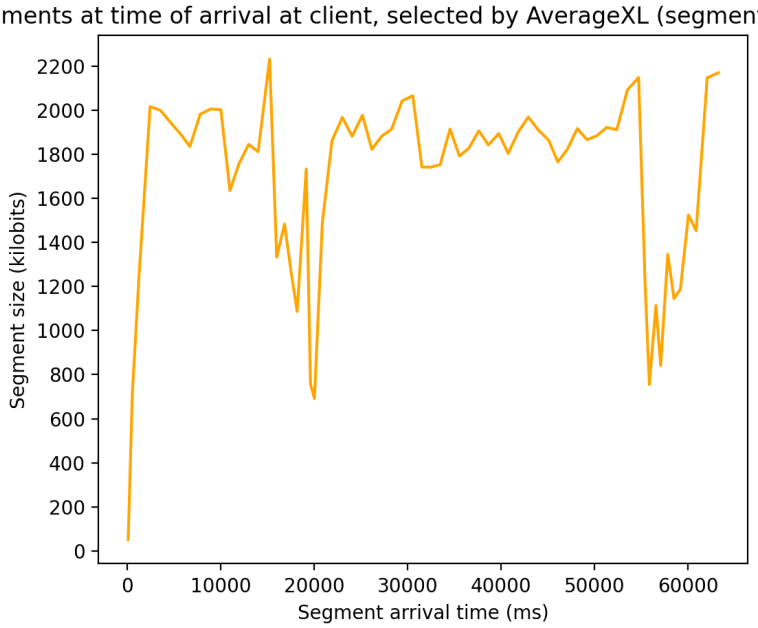


Figure 4.8: This is the size of the segments that AverageXL downloaded during the ns-3 test from Figure 4.6. The fluctuation shows the difference between the sizes of segments from the same representation.

fill up the buffer a lot faster, causing a peak in buffer occupancy.

Between the peaks in buffer occupancy, we can also observe the buffer occupancy dropping slowly. For example, between the peaks at 20000ms and 55000ms in Figure 4.6a, the buffer occupancy dropped by almost 20000ms. Figure 4.8 shows that during that period, the size of the segments was close to 2mbps. The network simulation allowed a throughput of 2mbps, so naively, we could assume that downloading segments of this size should just about be possible without losing buffer occupancy. However, there are some losses in the entire system of the client. Firstly, there is a small but still significant delay between the download of every segment by goDASH. We already discussed this briefly in Section 4.2.3. When a segment arrives, the client logs all the data about the previous segment. After that, it will use the data it gathered to execute the ABR algorithm. The algorithm will decide what segment to download next. Only after all that is done can the next segment actually start downloading. This takes about 80ms with goDASH using AverageXL on our testing machine. That is 80ms nothing can be downloaded from the server, so this is already a part of the 2mbs capped throughput that is not utilised. Also, while downloading, the client never actually experiences a throughput of 2mbps. Figure 4.9 shows the actual throughput the client using AverageXL perceived during that test. The throughput peaks at around 1800kbps, which is a little less than the 2mbps the network simulation allows. This could be because the segments are not big enough to build op a big enough congestion window to allow for a throughput of 2mbps. It could also be caused by inefficiencies in goDASH or quic-go, or by the way the network simulation works under the hood. This is a topic worth investigating more in the future.

Because the bitrate of the segments being downloaded is actually higher than the throughput (segments of 200kbps during a throughput of 1800kbps), the buffer occupancy slowly drops. That is what we see between the peaks in Figure 4.6a.

The tests with the Akamai dataset in Figure 4.7 give similar results. We can see that the average bitrate that the ABR algorithms selected are a bit higher for AverageXL in Figure 4.7a. At the same time the time spent stalling is higher for AverageXL in Figure 4.7c. This also explains why Average has a higher QoE score than AverageXL in Figure 4.7b.

The cross-layer implementation seems to overestimate the available bandwidth. This is expected because measuring the data that passes through the transport layer means that all application layer data is measured. The size of the segment plus the size of the HTTP headers. An application layer ABR algorithm estimates the throughput by measuring the time it took to download a segment, and then dividing the size of the segment by that time. The application layer measures the *goodput* (the amount of data that we receive that is actually useful), while our cross-layer implementation measures the *throughput*. What was not anticipated was that the impact of this slight overestimation could have

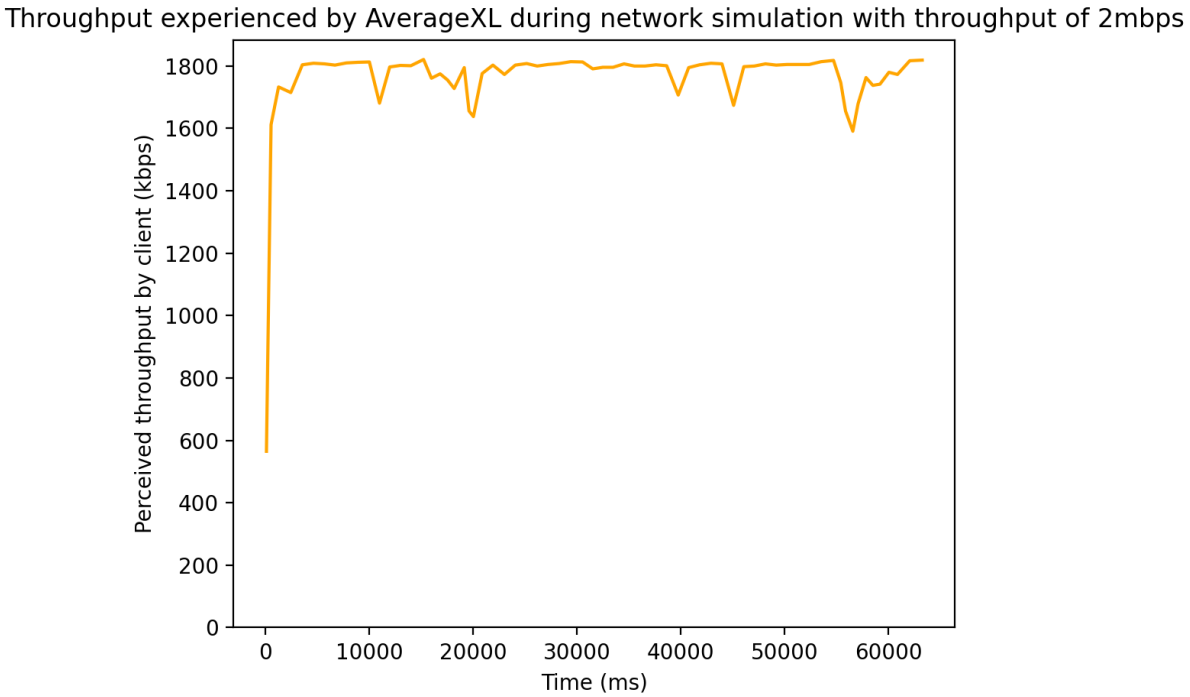


Figure 4.9: This is the actual throughput the client perceives during the download of segments (using AverageXL). This data is from the same test as the data in Figure 4.6. The throughput is lower than the throughput allowed by the ns-3 network simulation, which is 2mbps.

this big of an impact in the behaviour of the algorithm.

We could try to solve this by estimating what part of the data passing through the transport layer is actually goodput, for example, by measuring the size of the HTTP headers in the application layer, or by using a metric that serves as an average of the size of the data that is not goodput. However, the aim of AverageXL was to prove that we can make an algorithm work by using cross-layer metrics sharing. Even though it currently does not perform as well as the original algorithm, it is likely to reach that level of performance by optimising it. At the same time, matching the performance of existing algorithms is not very useful. It would be more interesting if we can change how ABR algorithms work by using the power of cross-layer metrics sharing. This is what we explore in the next section.

4.3 Buffer-based ABR with download abandonment

Instead of trying to alter an existing ABR algorithm with cross-layer metric to create an algorithm that matches or exceeds the performance of that original algorithm, the power of cross-layer metrics sharing could be used to add new functionality to an ABR algorithm. The reason why we stated in hypothesis 1 that no improvement in performance is expected by adapting existing ABR algorithms is because those algorithms tend to make decisions the moment new information arrives. That is, every time a new segment is downloaded. A buffer-based algorithm will know the buffer occupancy at that time, while a rate-based algorithm will determine the throughput of the latest segment. The power of using cross-layer metrics sharing of type i (which we defined in Chapter 1) is that the transport layer receives updates about the network more often than the application layer. It is able to make decisions while a segment is still being downloaded. Analysis of related work in Section 3.3 shows that some researchers try to detect stalling and try to prevent it by aborting the download of the current segment and switching to a lower representation. This is a good use-case for cross-layer metrics sharing because the decision of aborting a segment has to be made during the download of that segment. Using cross-layer metrics sharing, we can try to detect stalling before it happens and switch to a lower representation.

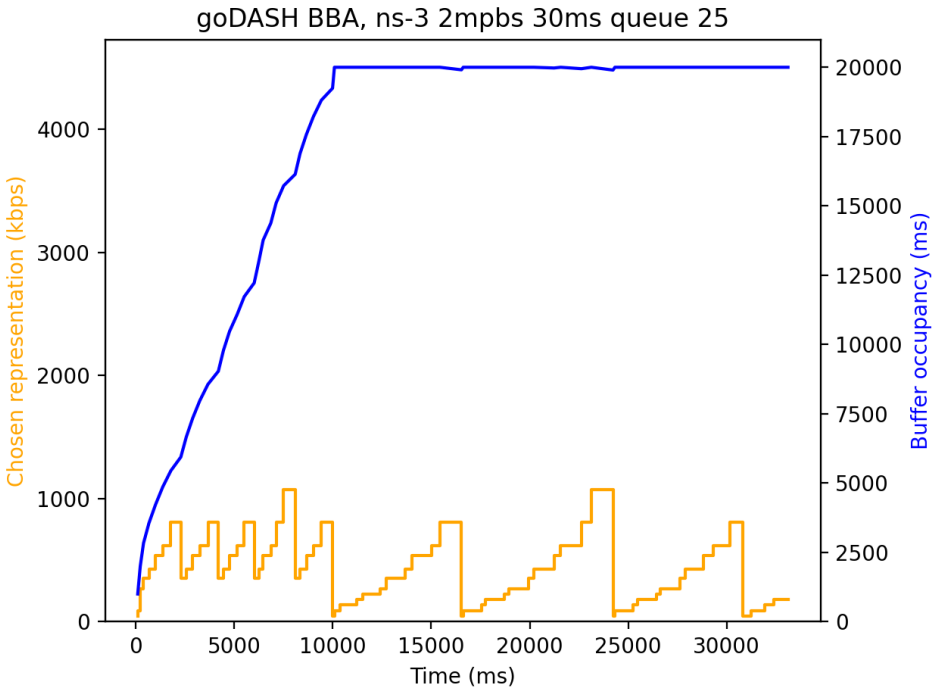


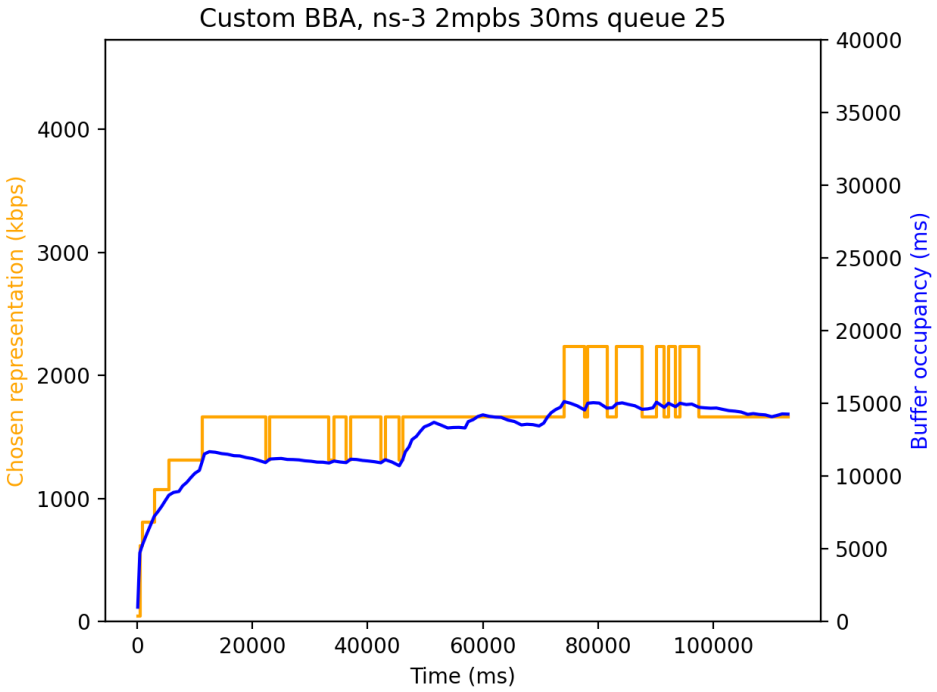
Figure 4.10: Selected bitrate and buffer occupancy of the BBA-2 implementation by goDASH with a video buffer size of 20 seconds. The chosen representation is displayed using the downloaded segments bitrate in kbps on the left y-axis. The buffer occupancy (buffer level), is displayed on the right y-axis in milliseconds. We would expect to see the chosen representation follow the line of the buffer occupancy if BBA was working as it should be.

4.3.1 BBA-1 implementation

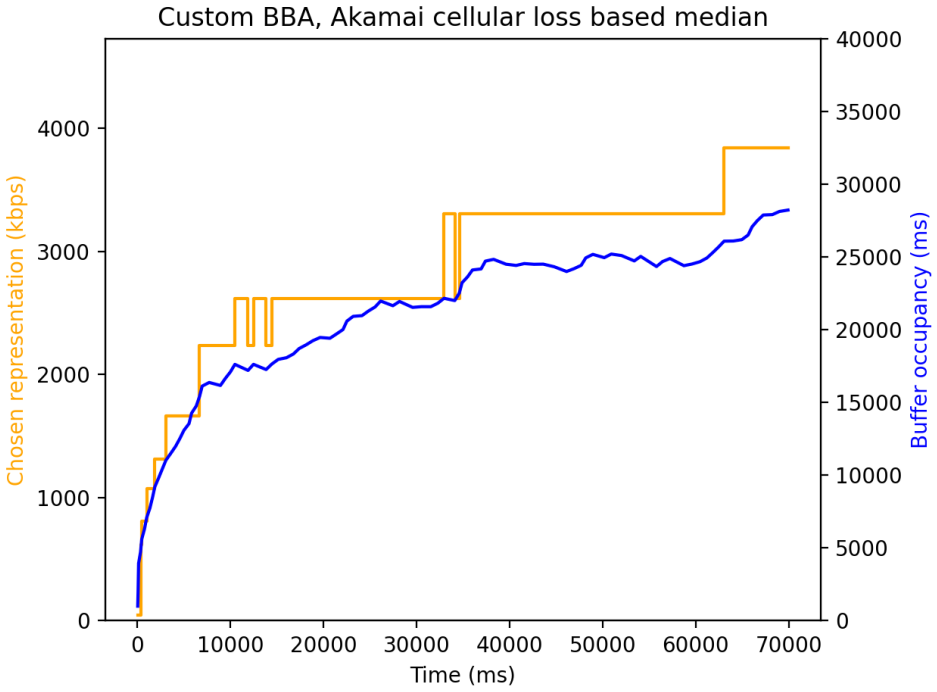
The goDASH framework provides an implementation of the BBA-2 algorithm. However, there are several problems with the implementation. The first problem is that it uses the version of BBA that has a variable reservoir [Huang et al. 2014b], as we briefly described in Section 2.2.2. The problem is that acquiring the size of the next segment, which is needed to calculate the reservoir size, takes too long. It is an extra RTT, because the algorithm requests the segment size using HTTP. The algorithm has to wait for this response and that is long enough to cause the algorithm to underperform in higher latency situations. Instead of requesting the next segment and waiting for it to arrive, it first makes a request to acquire the size of the next segment, which takes 1 RTT. Then, it requests the actual segment it is going to download. In the scenario we tested it, our delay was 30ms. This means it takes 30ms for a packet from the client to reach the server. If the segment size is requested first in this scenario, it takes 30ms for the request to reach the server, another 30ms for the response to reach the client and then we need another 30ms for the request of the next segment to reach the server. That is at least 90ms before the server can start sending packets of the segment. It would be better if this information was somehow available at the start of the stream by including this in the MPD for example.

The second problem is that the algorithm acted in an unexpected way. The algorithm would constantly be switching representations, even if the buffer occupancy was not changing that drastically, as shown in Figure 4.10. Since representation changes are not good for the QoE, this implementation is not a good representation of the BBA-2 algorithm. First, we attempted to fix the existing algorithm, but we did not manage to solve all problems. Instead, we implemented our own BBA-1 algorithm into the framework.

We defined a lower and an upper reservoir in our implementation with a fixed size of 90 percent of the total video buffer size. In the case of our tests, we use a video buffer size of 40 seconds, which means both reservoirs have a size of 4 seconds. These are values that could probably be tweaked depending on variables like the size of segments, but for the use case in this thesis, this implementation is good enough. The *cushion*, as the authors of Huang et al. 2014a call it, is the range in the buffer occupation that is mapped to representations. We kept the implementation of the rate map very simple. We calculated at what percentage the current buffer occupancy is in this cushion range. For example, in case we have a video buffer size of 40 seconds and reservoirs of 4 seconds, the cushion runs from 4 seconds buffer



(a) Test using a constant throughput of 2mbps, a constant delay of 30ms and a packet queue of 25 in ns-3.



(b) Test using a cellular dataset by Akamai.

Figure 4.11: Selected bitrate and buffer occupancy of our own BBA-1 implementation with a video buffer size of 40 seconds. BBA-1 only looks at buffer occupancy to decide what segment it will select nest. There are no optimisations done to reduce quality switches. That is why it is expected to see the yellow line fluctuate, even though quality switching is not good for the Quality of Experience.

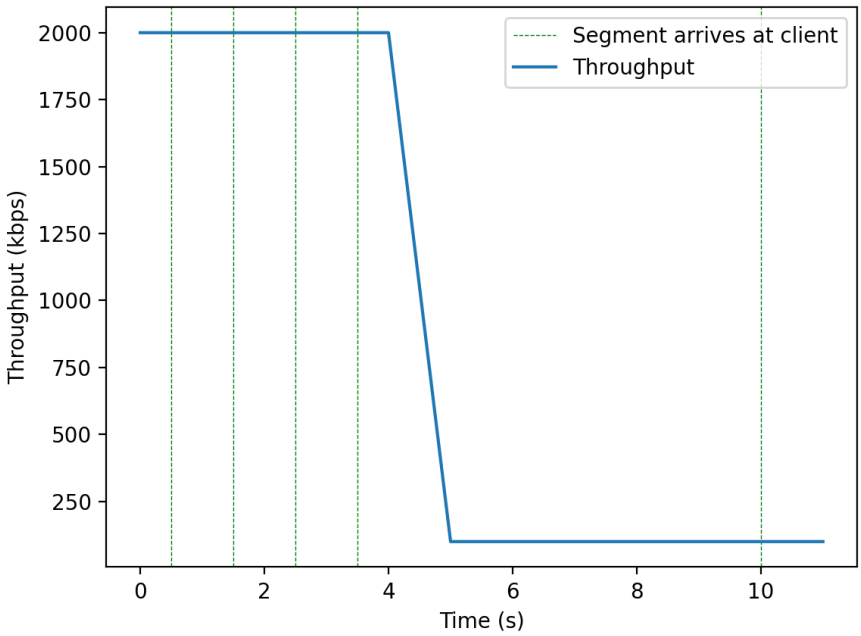


Figure 4.12: Hypothetical scenario that might cause BBA to experience stalling. The blue line shows the throughput the client would experience on the network over time. The vertical green lines indicate every point in time when a new segment arrives at the client. After the fourth green line, the fifth segment has just started downloading. When the throughput suddenly drops, that same segment will take a lot longer to arrive at the client.

occupancy to 36 seconds of buffer occupancy. When our current buffer occupancy is 19.2 seconds, we are using 60% of the cushion. We then directly map this on the available bitrates. If our lowest bitrate is 46kbps and our highest bitrate is 4700kbps, a buffer occupancy of 19.2 seconds will give us a desired bitrate of 2847,6kbps. We then select the bitrate that is equal to that desired bitrate or right below it. The same terminology is used as in Figure 2.3. B values indicate a position on the Buffer Occupancy axis. B_1 is the top of the lower reservoir, B_m is the start of the lower reservoir and B_{max} is the buffer size. The value *bufferOcc* is the current buffer occupancy. R values indicate all the possible representations. R_{min} is the lowest representation and R_{max} is the highest representation. The cushion is the part of the rate map that linearly maps buffer occupancy to bitrates of representations, starting from the lower reservoir and ending at the upper reservoir.

$$cushionUsage = \frac{bufferOcc}{B_{max} - lowerReservoir - upperReservoir} \tag{4.4}$$

$$desiredBitrate = (cushionUsage * R_{max}) + R_{min} \tag{4.5}$$

Figure 4.11 shows how our implementation performs in two different scenarios. This is how we would expect a buffer-based algorithm like BBA to perform. What is interesting is that these measurements also clearly show how slow BBA-1 climbs to a steady-state. This is exactly why the authors of Huang et al. 2014a used throughput estimation at the start of the stream in BBA-2 to ramp the bitrate up to a higher representation quickly.

4.3.2 Creating challenging scenarios

Before we can start detecting that stalling is about to happen using cross-layer information sharing, we first need to create a scenario where a good ABR algorithm like BBA will experience stalling. The test with the cellular dataset from Akamai is already quite challenging, but as Figure 4.11b shows, BBA-1 has no problems handling this scenario. We need to create extremely challenging situations with big variations in throughput.

Figure 4.12 shows a hypothetical scenario that could cause stalling for the BBA algorithm. At the start, the throughput is high and consistent, allowing the video buffer to fill. The vertical lines indicate the moment when a segment arrives. The BBA algorithm decides what segment to download next every time a segment arrives. In this example, after the fourth segment arrives, the buffer will be quite full and the algorithm will likely decide to download a segment with a bitrate close to the current throughput. However, when downloading the fifth segment, the throughput suddenly drops drastically. It takes a long time to download the segment, and by the time it arrives, the buffer might already be empty. Only after the fifth segment arrives the BBA algorithm will be able to react to the change on the network.

4.3.3 Detecting stalling before it happens

The implementation we created in Section 4.2 is capable of reading throughput on the transport layer. Now, we try to expand on this idea to not only report the throughput to the application layer but to actively make decisions based on this data. The first step to designing an algorithm that can decide to abandon segments is to make an algorithm that can predict stalling. We use a value that we call the *prediction window* to run this calculation on. The prediction window is a range of recently received QUIC packets. We track the incoming QUIC packets, and we keep the packets that are in this window. You can see it as a FIFO queue where QUIC packets come in and get pushed out once they are no longer inside the window. This window stays within one segment download, meaning that every time a new segment download is started, the FIFO queue is emptied. We use a window for the calculation rather than the information of all packets because we want to use very recent throughput information. This is also why the queue is emptied with every new segment. As we explained with Figure 4.1, there can be relatively large moments of silence between the download of each segment. If our prediction window would span multiple segments, there might be a lot of time between those measurements in the window.

```
1  func stallPredictor() {
2     // Calculate sum of all bits received
3     var sum_bytes int = windowFIFO.bytes.sum()
4     sum_bits := sum * 8
5     predictionWindowStartTime := windowFIFO.time[windowSize]
6
7     // Calculate the average throughput of the prediction window
8     windowTotalTime_ms := time.Since(predictionWindowStartTime).Milliseconds()
9
10    // bits  := (kbps == bpms)  / ms
11    segmentSize := (representationBitrate) / segmentDuration
12
13    // Only do predictions when we have received less bytes than we expect to receive
14    if bitsReceived < segmentSize && windowTotalTime_ms > 0 {
15       bitsToDownload := segmentSize - sum_bits // Number of bytes that need to be downloaded
16       // bits / ms := bits / ms
17       windowBitrate := sum_bits / int(windowTotalTime_ms)
18       // Time it will take in ms to download the remaining bits at this rate
19       requiredTime_ms := bitsToDownload / windowBitrate
20
21       if requiredTime_ms > a.calculateCurrentBufferLevel() {
22          // Report stall prediction
23          Print("STALL PREDICTED")
24       } else {
25          Print("NO STALL")
26       }
27    }
28    }
29 }
```

Listing 4.2: Code that does stall prediction.

Listing 4.2 shows the pseudocode of the prediction algorithm. This function is executed every time we receive a QUIC packet. The algorithm calculates the throughput experienced in the prediction window. Then, it calculates how many bits of the current segment still need to be downloaded. We can calculate this value if we know the current segment's total size and then subtract the measured amount of bits from that. One option would be to extract the payload size parameter from the HTTP headers while the segment is still being downloaded to know exactly what the size of the current segment is. However, for this implementation we took a shortcut and calculated the average segment size by dividing the bitrate

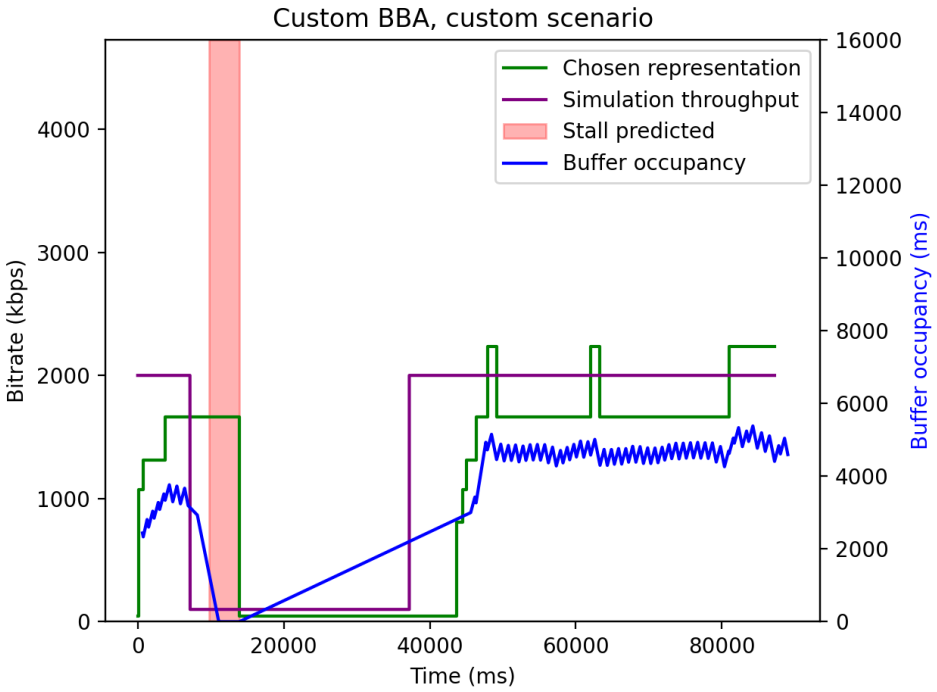


Figure 4.13: Visualisation of the stalling prediction algorithm. The red bar is a visualisation of the period the stalling predictor was predicting a stall. The green line shows the chosen representation. It uses the bitrate of the segments it received and displays this in kbps, using the left y-axis. The blue line shows the buffer occupancy over time. The buffer occupancy is the number of milliseconds of video that still remains in the video buffer. The buffer occupancy uses the right y-axis. With a correctly working BBA algorithm, the green line should somewhat follow the behaviour of the blue line, since the representation choice is dependent on the buffer occupancy. The purple line indicates the throughput the client experienced in kbps, using the left y-axis, enforced by the custom network scenario we created. As expected, the buffer occupancy drops after the simulation throughput drops.

of the representation by the segment duration. Lastly, it predicts how long it would take to download the rest of the segment at the average bitrate of the prediction window by dividing the amounts of bit left to download and the experienced average throughput of the prediction window. If it would take longer to download the current segment than that we have video remaining in the buffer, the algorithm predicts stalling. The calculation is only triggered when the prediction window is full. This way of predicting stalling pairs well with buffer-based algorithms like BBA, because the buffer occupancy is always related to the network condition. If the buffer is full, it is likely that the network conditions are favourable. If the buffer occupancy is very low, we are either in the startup phase, or the network conditions are not favourable. If we are actually in the startup phase, the algorithm will select a very low bitrate, meaning that we are filling the buffer faster than we are consuming it. The startup phase is, therefore, unlikely to trigger the stalling predictor. At the start of each segment, the queue is emptied and the algorithm knows the buffer occupancy at that point. When calculating the current buffer occupancy, the algorithm calculates how much time has passed since the start of the segment. Then, it subtracts it from the memorised buffer occupancy to know what the current buffer occupancy would look like.

To test the implementation, we recreated the scenario of Figure 4.12 using Linux traffic control network emulation scripts. As expected, it did introduce stalling. Our first implementation only reports if it is predicting stalling and does not interfere with BBA itself. We used this on our custom scenario to test it. Figure 4.13 visualises what the algorithm reported. Initially, the throughput of our scenario was 2mbps. We can see that the video buffer fills quickly at the start and eventually selects a representation with a bitrate just below 2mbps. However, soon after reaching that representation, the buffer starts to decline steadily. This is because at that moment, the throughput was limited to 100kbps, which is only enough to download some of the lowest representations. However, this change in bitrate happened during a segment was being downloaded. This is why we see a steady decline in buffer occupancy. No segments are arriving, but the buffer is still being consumed at a rate of 1 segment per second (because of a segment size of 1 second). The BBA algorithm can not make a decision until the segment arrives, at

which point it is already too late. The red area indicates at what point during the simulation the stalling predictor was reporting that at the current download speed, the segment would not arrive in time. At around 38 seconds into the simulation, the throughput is set back to 2mbps, and BBA manages to build up a healthy video buffer again. This test shows that the prediction algorithm could work in a scenario where BBA falls short.

In our implementation, we used a prediction window of 20 packets, which is rather small. Tests were also performed with larger prediction windows of 50 to 100 packets. This sometimes causes the prediction algorithm to make decisions very late because it waits to calculate predictions until the prediction window is full. Instead of using a fixed amount of packets for the prediction window, we could also use a window that depends on the current segment's size. For example, if the size of the current segment is Y, the prediction window would have a size of $X * Y$ where X is a fixed value between 0 and 1. Downloading smaller segments (a smaller Y) would mean that we have to wait for fewer bits to arrive, and thus fewer segments, to start making predictions. Downloading larger segments would cause the opposite, Y would be larger, so we would have to download more bits, before we can start making predictions.

4.3.4 Abandoning a segment when stalling is predicted

The next step is to make the client stop downloading the current segment and switch to a different representation. In our goDASH and quic-go implementation, we are able to do this by giving a context to the object that handles HTTP requests, and then cancel the context in a different thread before the download is complete. The client then makes a new request for a segment of a lower representation. Because we are using BBA as the ABR algorithm for this cross-layer implementation, we can always choose the lowest representation after aborting a segment. This might, however, not always be the most optimal choice. Another option would be to start using a rate-prediction algorithm that uses the cross-layer metrics instead, but this is future work.

Problems with BBA for low throughput scenarios

When testing our BBA implementation together with our cross-layer segment abort algorithm (BBA-XL), we noticed that the aborts were not having the expected result. Figure 4.14 shows one of these tests. The segments were being aborted during logical moments, but after a segment was aborted, the throughput would seem to be lower than expected.

The BBA algorithm also tends to switch back to a higher representation too soon. This is because in these tests, we were using a video buffer of 15 seconds, with upper and lower reservoirs of size 1.5 seconds. We also have 20 different representations available in our test stream, which is a lot. All these parameters combined cause the BBA algorithm to be very sensitive and switch representations too quickly. The available throughput at the lowest points in these tests are 100 to 200kbps. Our lowest representations have a rate of 46, 92, 135 and 182kbps. Because the BBA algorithm switches representations too quickly, it causes more stalling when being very limited in available throughput, because it is not able to build a video a sufficient buffer occupancy to deal with the quick representation changes.

Final solution

After we increased the video buffer size, we noticed that the prediction algorithm would sometimes abort segments while the buffer is still very full. This is not that useful. Let's say the buffer is filled with 30 seconds of video, and the prediction algorithm predicts that the current segment will not be downloaded in 30 seconds. The prediction algorithm only looks at a part of a segment download packet information to make fast predictions of the current network situation. However, it is very likely that during those 30 seconds, the network situation might change again. Aborting now would be a premature decision. This is why an additional buffer occupancy threshold was introduced. While the level is above the threshold, the prediction algorithm will never call for an abort of the segment. We selected a threshold of 10 percent of the buffer size in our tests. This is another parameter that can be tweaked to optimise the algorithm depending on what kind of video content is played and what kind of client we are using, together with the prediction window size, reservoir sizes and the buffer occupancy itself.

If we look back at the results of the base BBA implementation in Figure 4.13, it also shows a long stall after the throughput is raised again at around 40000ms. This is because the goDASH framework works with an initial buffer size. It will pause the simulated playback of video until this initial buffer size is reached. This is something that all video players do. However, after experiencing stalling, if the buffer occupancy is below that initial buffer size again, it would pause playback again until the buffer

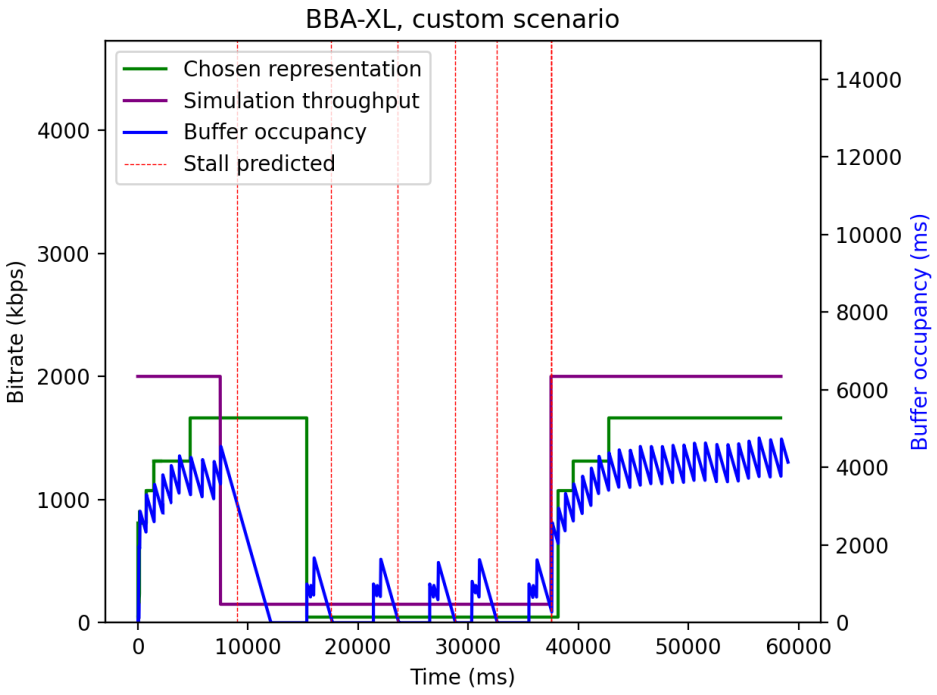


Figure 4.14: Visualisation of BBA with cross-layer segment abort with a video buffer that is too small. A vertical red line indicates that the algorithm has triggered the abort of the segment that was being downloaded at that point. After an abort, BBA-XL switches to the lowest available representation. The green line shows the chosen representation. It uses the bitrate of the segments it received and displays this in kbps, using the left y-axis. The blue line shows the buffer occupancy over time. The buffer occupancy is the amount of milliseconds of video there still remains in the video buffer. The buffer occupancy uses the right y-axis. With a correctly working BBA algorithm, the green line should somewhat follow the behaviour of the blue line since the representation choice is dependent on the buffer occupancy. The purple line indicates the throughput the client experienced in kbps, using the left y-axis, enforced by the custom network scenario we created. The buffer occupancy never gets to a high enough level and drops back to zero several times.

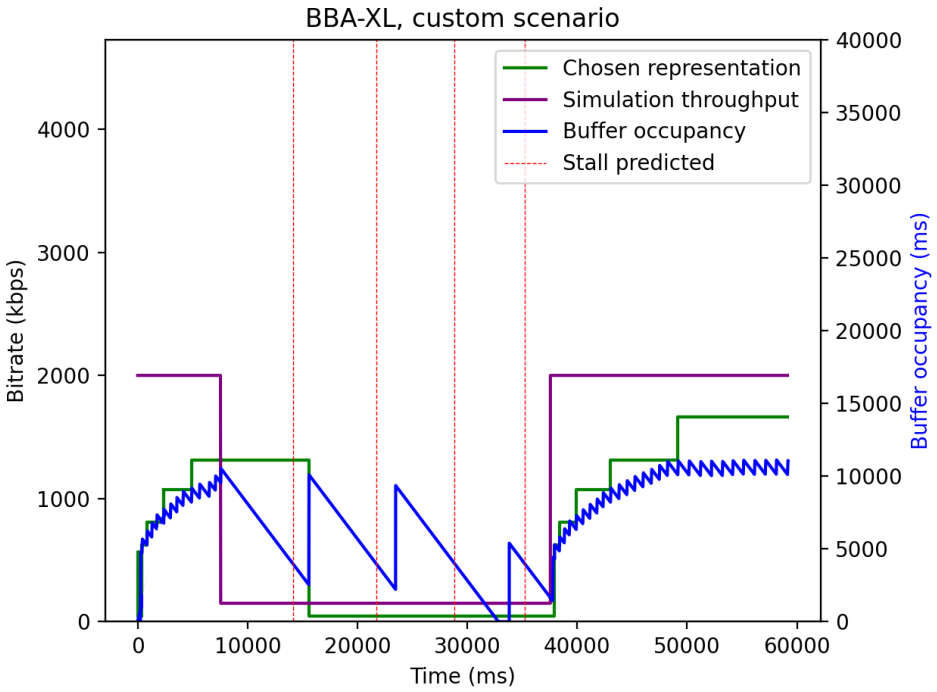


Figure 4.15: Visualisation of BBA with cross-layer segment abort with a big video buffer. A vertical red line indicates that the algorithm has triggered the abort of the segment that was being downloaded at that point. After an abort, BBA-XL switches to the lowest available representation. The green line shows the chosen representation. It uses the bitrate of the segments it received and displays this in kbps, using the left y-axis. The blue line shows the buffer occupancy over time. The buffer occupancy is the number of milliseconds of video there still remains in the video buffer. The buffer occupancy uses the right y-axis. With a correctly working BBA algorithm, the green line should somewhat follow the behaviour of the blue line, since the representation choice is dependant on the buffer occupancy. The purple line indicates the throughput the client experienced in kbps, using the left y-axis, enforced by the custom network scenario we created. The buffer occupancy only reaches zero once for a small duration of time.

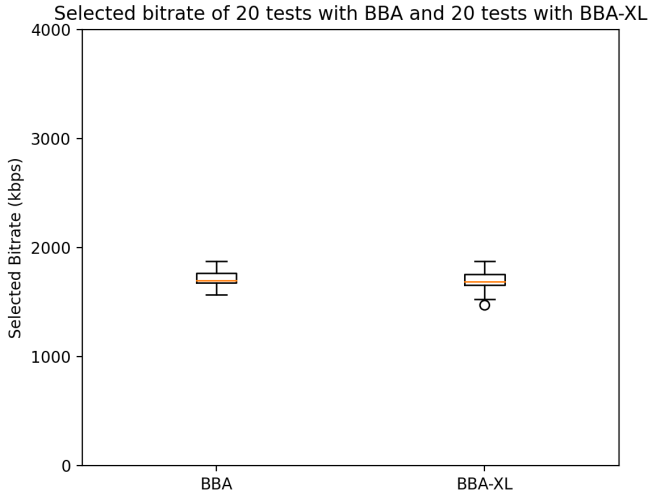
is sufficiently filled. This is not very desirable, though, because it can cause additional stalling like we see in Figure 4.13. Because of this, we changed the goDASH framework so that it would only do this for filling the video buffer initially.

4.3.5 Results

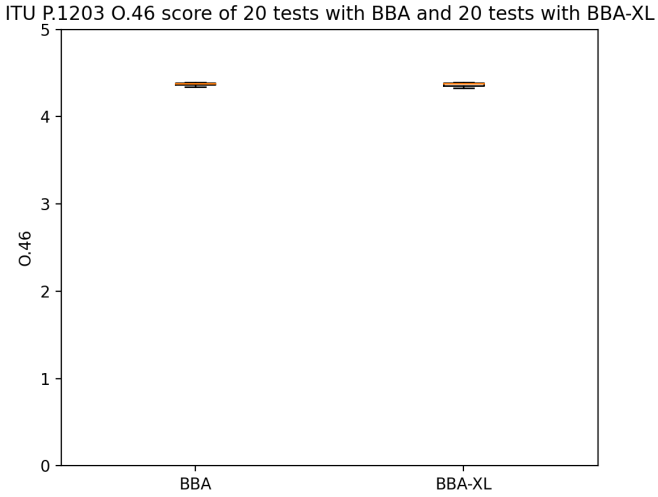
We performed 20 tests for our custom BBA implementation and 20 tests for the cross-layer version with segment abort: BBA-XL (BBA Cross Layer). We used a poor cellular scenario based on an Akamai dataset. We then measured the same metrics as we did when comparing Average against AverageXL in Section 4.2.4. Figure 4.16 shows boxplots comparing BBA and BBA-XL. Figure 4.16a shows the variation in average bitrate. Figure 4.16b shows what ITU P.1203 O.46 QoE scores the tests received. Lastly, Figure 4.16c shows the amount of perceived stalling for the streams.

Both BBA and BBA-XL experienced 0 stalling in their 20 tests respectively, even though this is a challenging cellular scenario with many variations in throughput. This shows again the BBA is already a good base algorithm to start from. The importance of this test is to see if BBA-XL does not perform worse in situations where segment aborts are not needed. In other words, we check how many false positives BBA-XL produces. According to our tests results, BBA-XL reported 0 aborts in all 20 tests with each a duration of 80 seconds. This proves that, at least in this cellular scenario, false positives are very low. The small differences between the results of BBA and BBA-XL are likely caused by the small variations each test with the dataset has.

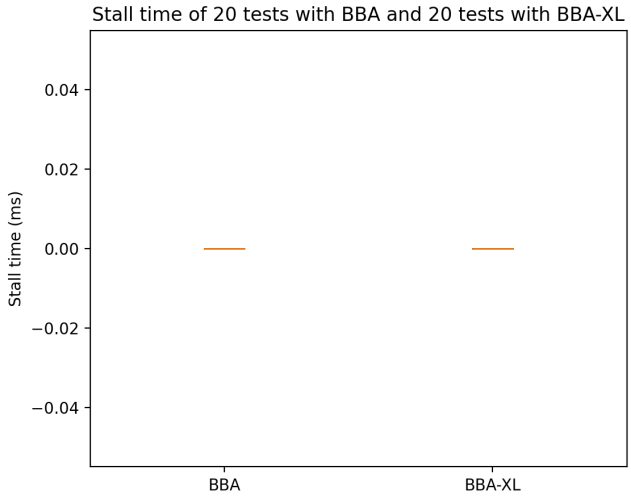
In Figure 4.15 we showed that the algorithm can work in a scenario where the original BBA algorithm would fail. However, the final BBA-XL implementation does still experience a little bit of stalling at around 32000ms. This happened because even though the throughput was still low, it was high enough at



(a) Boxplots of the chosen bitrates by BBA and BBA-XL.

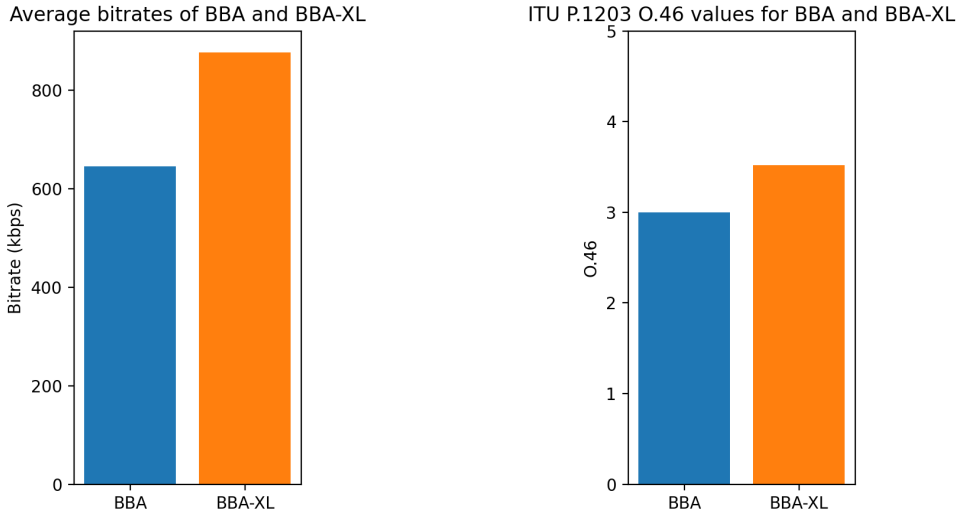


(b) Boxplots of ITU P.1203 O.46 scores of BBA and BBA-XL. The score shows the quality rating a real group of people would give to the video stream. The score is a value between 1 and 5. The stream with the highest score has the highest perceived quality.



(c) Boxplots of the total amounts of stalling the clients experienced using BBA and BBA-XL.

Figure 4.16: Comparison between our BBA implementation and the cross-layer BBA-XL version using a simulation of a cellular network provided by Akamai. 20 tests were performed for each algorithm.



(a) Average bitrate of segments downloaded by BBA and BBA-XL in custom scenario.

(b) ITU P.1203 O.46 scores of BBA and BBA-XL in custom scenario.

Figure 4.17: Comparison between our BBA implementation and the cross-layer BBA-XL version with the custom scenario.

150kbps to fill up the buffer enough with segments of 46kbps to make BBA step to a higher representation. This is again an error of BBA-1. BBA-XL notices this, abandons the segment and switches back to 46kbps. This is why we see four stall predictions. However, by the time the stall was predicted for the third time, the buffer was already too low. This is why there is still a little bit of stalling. This might be resolved by estimating if aborting a segment now and then downloading the lower representation would take less time than to just finish the current segment. It might also be resolved by making sure BBA does not make that decision to go to a higher representation by using a better version of BBA. Figure 4.17b shows the ITU P.1203 O.46 score of BBA and BBA-XL in the custom scenario. BBA-XL performs slightly better in this case with a slightly higher QoE score. Figure 4.17a shows the average bitrate of segments downloaded by BBA and BBA-XL in that same scenario. BBA-XL also performs slightly better than BBA in this case. This means that BBA-XL was able to download video segments of higher quality overall.

4.3.6 Aborting segments only if that is the fastest option

Figure 4.15 shows that BBA-XL can perform better than BBA. However, there still is some stalling during this test. We already explained why this happens in Section 4.3.5. The third abort might have been a bad decision. The overall QoE might have been better if we did not abort that segment, even though the bitrate was too high and triggered the stalling detection. If the time spent stalling had been similar, we would at least have had a segment of a higher bitrate. If the segment was almost completely downloaded when the abort was triggered, it might even have stalled less. A new feature was added to BBA-XL to find out if this was the case. It now not only estimates how long it will take to complete the current segment, but it would also estimate how long it would take to download the segment of the lower representation at the current throughput in case we abort the current segment. If it would take longer to download the new segment than to complete the current segment, the abort would not be called. We tested this implementation in the same scenario as in Figure 4.15, but this addition caused segment aborts to no longer trigger at all in this scenario. The reason why this did not work as expected is not entirely clear. It is likely a bug in the implementation of the new logic that we were not able to find. Thus, we did not include this feature in our final tests, but it is still something that could be researched further.

4.4 Discussion

In this Chapter, we explained how we made AverageXL, a cross-layer adaptation of the existing Average algorithm. We replaced the measurement of the metrics the algorithm uses (received bits and delivery times) with metrics that we read in the transport layer. This is what we need to answer research question

1. According to hypothesis 1, we originally thought that the difference between these metrics is not big enough to cause the ABR algorithm to make a different decision. This hypothesis turned out to be incorrect for Average and AverageXL, because they did not perform exactly the same. In fact, the cross-layer version performed worse than the original algorithm. This is caused by the fact that we are not measuring goodput, but throughput in the cross-layer version. The difference in throughput and goodput seems to be big enough to cause the ABR algorithm to make a different decision. Thus, hypothesis 1 is incorrect.

With the lessons we acquired while developing AverageXL, we made a second algorithm to answer research question 2. We based this algorithm on BBA and transformed it into a new algorithm that works closely together with cross-layer communication of type i. We call it BBA-XL. BBA-XL performs better in the scenario that is specifically designed to make BBA fail. In general tests, BBA and BBA-XL perform the same. However, there are a lot of parameters involved like the prediction window, the threshold, BBA's reservoir size and BBA's buffer size that can heavily influence how the algorithm performs. Nevertheless, we can say that we created a new kind of ABR algorithm with BBA-XL, which exploits the strength of having transport layer metrics in the application layer. BBA-XL can make decisions more frequently and better informed than typical ABR algorithms. Research question 2 asks if it is possible to create a new kind of ABR algorithm by using cross-layer information sharing. It also asks the question if this algorithm can match or outperform existing ABR algorithms. In hypothesis 2 we stated that it should be possible to match or outperform existing algorithms. This hypothesis is correct according to the test results of BBA-XL. We created a new kind of ABR algorithm, BBA-XL, that uses cross-layer information sharing. When comparing it to BBA, a base implementation of a popular ABR algorithm, we see that in general tests, the new algorithm matches the performance of BBA, achieving a higher QoE. We also see that it outperforms BBA in a scenario created specifically to point out the flaws of BBA-1, where our BBA implementation is based on. Thus, we can state that hypothesis 2 is correct.

Chapter 5

Conclusion

Network layers have always been a black box to each other. There are good reasons why this abstraction is made. However, in the case of video streaming, the black box can cause suboptimal performance [Arye, Sen, and Freedman 2018]. If the transport and application layer would be able to communicate with each other, video streaming performance might be improved. Cross-layer information sharing between a QUIC transport layer and a DASH client is possible because QUIC implementations are being developed in user space [Herbots et al. 2020]. In this thesis, we investigate whether cross-layer information sharing is useful for ABR video streaming.

We discussed three different types of cross-layer information sharing. A software stack that does information sharing of type i will communicate information from the transport layer to the application layer. A cross-layer software stack of type ii will communicate information from the application layer to the transport layer. VOXEL [Palmer et al. 2021] does something like this. With VOXEL, the application layer let the transport layer know if a particular set of frames can be streamed unreliably or not. Lastly, type iii defines cross-layer communication between an application layer and a transport layer that communicates both ways. You can see it as a combination of type i and type ii. In this thesis, we looked at communication of type i, where an edited version of quic-go would send metrics about the transport layer to an edited version of the goDASH application.

We used this implementation to answer the two research questions. In short, the first research question asks if we can simply replace the parameters that existing ABR algorithms use with information gathered from the transport layer, without changing how the algorithms work and if that would impact the performance. The second question asks if we can use cross-layer information sharing to develop a new kind of ABR algorithm that uses those metrics for a feature that was not previously possible. We were able to answer the first question by making a cross-layer version of the Average algorithm, which we call AverageXL. We hypothesised that there would be no change in performance because the metrics from the transport layer are not different enough from the application layer to impact the decision-making process of an existing ABR algorithm. Our hypothesis is not entirely correct. Our tests show that Average outperforms AverageXL. This is likely due to the fact that we were estimating throughput in the transport layer and comparing that to an estimation in the application layer. The transport layer measures *throughput* while the application layer is actually measuring *goodput*. This slight difference in values causes AverageXL to make slightly different decisions compared to Average. Thus, our hypothesis is incorrect. However, we do point out that with further optimisations of AverageXL, we should be able to have the same performance with AverageXL as we have with Average.

We answered the second question by making a cross-layer version of BBA, which we call BBA-XL. BBA-XL uses cross-layer metrics to decide if segment should be aborted to prevent stalling. We tested this algorithm in a simulated network scenario where BBA would typically fail. In this test, BBA-XL outperforms BBA. In other tests that evaluate the performance of both algorithms in a general scenario, both algorithms performed equally because BBA-XL experienced no false positives. It is important to note that BBA-XL has a lot of variables that can be adjusted to change its behaviour, like the prediction window and the segment abort threshold. There are variables in the original BBA algorithm that can have a big impact on its performance, like the buffer and reservoir sizes. We hypothesised that the new algorithm would have at least equal performance and might outperform existing ABR algorithms. This hypothesis is correct, as BBA-XL has equal performance to BBA in general tests and outperforms BBA in specific scenarios.

5.1 Future work

This thesis answers research questions, but also creates some new questions. We defined three types of cross-layer information sharing, be we only conducted research about type i. Future work can research if the other types are useful as well. BBA-XL performs well compared to BBA, but our BBA implementation is based on BBA-1. There are a lot of improvements and features added to new versions and variations of BBA. Is cross-layer stalling detection still useful in these versions? Can we use the same logic used in BBA-XL to make cross-layer versions of other ABR algorithms?

BBA-XL in its current from also has a lot of room for improvement. The prediction window estimates throughput, but not goodput, just like AverageXL. However, it should be possible to compensate for this miscalculation by extracting the content length from the HTTP headers, or by making an estimation of the HTTP header size and using that to estimate the goodput. This would make stall prediction more accurate and it should make AverageXL perform exactly like Average. A fixed number of packets currently defines BBA-XL's prediction window. Depending on how QUIC divides a segment and how big the current segment is, this amount of packets might not be a good window to make predictions in. The window might be too large for small segments or too small to make average predictions in for bigger segments. Future work can research if switching to a number of received bits relative to the segment size is a better way of defining the prediction window.

Only basic metric extraction from the transport layer is used in AverageXL and BBA-XL. When QUIC packets are received, the send and arrival time is passed to the application layer, together with the packet size. The application layer then uses these metrics for throughput estimation. There are a lot more metrics that can be extracted from the transport layer, like RTT, congestion window size, packet loss etc. Can these metrics be used to make better ABR algorithms?

In Section 4.2.4, we observed that goDASH never reaches a throughput of 2mbps with a network simulation that allows a throughput of 2mbps. In reality, it experienced a throughput of about 1800kbps. It is worth investigating what exactly is the cause for this lower throughput. Also, it takes goDASH 80ms to go from receiving the final packet of a segment to requesting the next segment. This is already less in our implementation than it was in the original goDASH implementation, but it still leaves some network capacity unused. If we can find a way to lower the time goDASH needs by making it more efficient, we might be able to increase the performance of the client slightly. A first step would be to let all the logging and data gathering be executed by a separate thread. Currently, it seems that the main play loop of goDASH is single-threaded.

Lastly, it became obvious during the testing phases that realistic network simulation is not trivial. Even the non-realistic network simulation that was used to create the stress scenario for BBA was not trivial to create. There are a lot of parameters that can be set that can completely change how the simulation interacts with the application that is being tested.

5.2 Reflection

I started this thesis with a completely different idea of what it would become than what it is now. I had a lot of previous knowledge about ABR streaming, because my bachelor's thesis also covered this topic. However, during this thesis, I learned so much more about how ABR algorithms work, what metrics they use, what the differences are between rate-based and buffer-based algorithms etc. With this knowledge, I learned that my original plan to adapt existing ABR algorithms with cross-layer metrics would not have any positive impact on the performance of these algorithms. I described this in my hypothesis for research question 1. I had to understand all the related topics to this thesis thoroughly, before I could understand that my original plan for this thesis would not work. Learning about ABR streaming, QUIC and how to do network simulations correctly was the most difficult and time-consuming part of this thesis.

The implementation of two cross-layer algorithms turned out good. AverageXL was a proof of concept, intended to prove that cross-layer metrics sharing was possible, in particular with the combination of quic-go and goDASH. Although it did not perform as expected, it was an excellent first step into getting to know Go and the inner workings of quic-go and goDASH. I learned enough to find flaws in goDASH and how to solve them. I learned how to do network simulations and how to display the performance of ABR algorithms using graphs. I also think that the more experienced I became with presenting the results in graphs, the better and more complete the graphs became. The final visualisations of the BBA stress test showed more granularity and more information in a single graph, so that more correlations between the results could be made. It is possible to overdo this and create a graph that shows so much information that the essence of the data is lost. However, I believe that readers that have some experience with video

streaming research will appreciate the graphs but they might be too difficult to understand for readers with only a general knowledge of the topic. Because my knowledge of making graphs kept improving, and because I started rewriting parts of the goDASH framework that provide the logs used for these graphs, the early graphs featured in this thesis are in my opinion not of the quality I desire them to be. Their data is less granular and because of that, they can sometimes give the wrong impression of the data. If I had more time, I would have considered doing all testing again from scratch with my new logging format to make better graphs for the thesis.

I am proud of how the implementation of BBA-1 turned out. I was not happy at all with the performance of goDASH's BBA-2 implementation, so I decided to implement one myself according to the specification of the original BBA paper [Huang et al. 2014a]. Even though it is an implementation of the most basic version of BBA, I was surprised by how well it works and how it performs almost exactly like I expected when reading the paper. I believe this shows that I have a good understanding of ABR algorithms and BBA in particular. It has also given me a particular appreciation for BBA, because its design is extremely clever in my opinion.

In hindsight, it would have been better to look even deeper into existing frameworks and QUIC implementations before deciding on which I would use for an implementation. The combination of quic-go and goDASH seemed like an obvious choice at first. But after working with goDASH and learning how it works internally, I started to understand the problems that the framework has. However, it is unlikely that there is a video streaming framework that supports QUIC without any flaws. Eventually, everything turned out good, and I am happy with the result of this thesis.

References

Akhshabi, Saamer et al. (2012). "What Happens When HTTP Adaptive Streaming Players Compete for Bandwidth?" In: *Proceedings of the 22nd International Workshop on Network and Operating System Support for Digital Audio and Video*. NOSSDAV '12. Toronto, Ontario, Canada: Association for Computing Machinery, pp. 9–14. ISBN: 9781450314305. DOI: 10.1145/2229087.2229092. URL: https://doi.org/10.1145/2229087.2229092.

Apple (2022). *HTTP Live Steaming*. https://developer.apple.com/documentation/http_live_streaming.

Arye, Matvey, Siddhartha Sen, and Michael J. Freedman (2018). *Poor Video Streaming Performance Explained (and Fixed)*. arXiv: 1901.00038 [cs.NI].

Bermudez, H.-F. et al. (2019). "Live video-streaming evaluation using the ITU-T P.1203 QoE model in LTE networks". In: *Computer Networks* 165, p. 106967. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2019.106967. URL: https://www.sciencedirect.com/science/article/pii/S1389128619304311.

Bhat, Divyashri, Amr Rizk, and Michael Zink (2017). "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, pp. 13–18. ISBN: 9781450350037. DOI: 10.1145/3083165.3083175. URL: https://doi.org/10.1145/3083165.3083175.

Bishop, M. (Mar. 2022). *Hypertext Transfer Protocol Version 3 (HTTP/3)*. Tech. rep. URL: https://quicwg.org/base-drafts/draft-ietf-quic-http.html.

Bishop, Mike (June 2021). "HTTP/3 and QUIC: Past, Present, and Future". In: URL: https://www.akamai.com/blog/performance/http3-and-quic-past-present-and-future.

Cerf, Vinton, Yogen Dalal, and Carl Sunshine (Dec. 1974). *Specification of Internet Transmission Control Program*. RFC 675. http://www.rfc-editor.org/rfc/rfc675.txt. RFC Editor. URL: http://www.rfc-editor.org/rfc/rfc675.txt.

Clemente, Lucas (2022). *quic-go*. https://github.com/lucas-clemente/quic-go.

DASH-IF (2022). *DASH-IF Reference Client*. https://reference.dashif.org/dash.js/.

Deng, Zhenjie et al. (2021). "Cross-layer DASH-based multipath video streaming over LTE and 802.11ac networks". In: *Multim. Tools Appl.* 80.10, pp. 16007–16026. DOI: 10.1007/s11042-020-10393-8. URL: https://doi.org/10.1007/s11042-020-10393-8.

Duanmu, Zhengfang et al. (2020). *Assessing the Quality-of-Experience of Adaptive Bitrate Video Streaming*. DOI: 10.48550/ARXIV.2008.08804. URL: https://arxiv.org/abs/2008.08804.

Engelbart, Mathis and Jörg Ott (2021). "Congestion Control for Real-Time Media over QUIC". In: *Proceedings of the 2021 Workshop on Evolution, Performance and Interoperability of QUIC*. EPIQ '21. Virtual Event, Germany: Association for Computing Machinery, pp. 1–7. ISBN: 9781450391351. DOI: 10.1145/3488660.3493801. URL: https://doi.org/10.1145/3488660.3493801.

Garrison, Ron (2021). *Structure of a MPEG-DASH MPD*. https://ottverse.com/structure-of-an-mpeg-dash-mpd/.

Garza, Javier (Apr. 2020). *A QUICk Introduction to HTTP/3*. https://www.akamai.com/blog/developers/a-quick-introduction-http3.

Guzmán Castillo, Paola, Pau Arce Vila, and Juan Carlos Guerri Cebollada (2019). "Automatic QoE Evaluation of DASH Streaming Using ITU-T Standard P.1203 and Google Puppeteer". In: *Proceedings of the 16th ACM International Symposium on Performance Evaluation of Wireless Ad Hoc, Sensor, & Ubiquitous Networks*. PE-WASUN '19. Miami Beach, FL, USA: Association for Computing Machinery, pp. 79–86. ISBN: 9781450369084. DOI: 10.1145/3345860.3361519. URL: https://doi.org/10.1145/3345860.3361519.

Herbots, Joris et al. (2020). "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, pp. 542–543. ISBN: 9781450379489. DOI: 10.1145/3386367.3431901. URL: https://doi.org/10.1145/3386367.3431901.

Hoßfeld, Tobias et al. (2013). "Internet Video Delivery in YouTube: From Traffic Measurements to Quality of Experience". In: *Data Traffic Monitoring and Analysis: From Measurement, Classification, and Anomaly Detection to Quality of Experience*. Ed. by Ernst Biersack, Christian Callegari, and Maja Matijasevic. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 264–301. ISBN: 978-3-642-36784-7. DOI: 10.1007/978-3-642-36784-7_11. URL: https://doi.org/10.1007/978-3-642-36784-7_11.

Huang, Te-Yuan et al. (2014a). "A Buffer-Based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: Association for Computing Machinery, pp. 187–198. ISBN: 9781450328364. DOI: 10.1145/2619239.2626296. URL: https://doi.org/10.1145/2619239.2626296.

— (Jan. 2014b). "Using the Buffer to Avoid Rebuffers: Evidence from a Large Video Streaming Service". In: DOI: 10.48550/ARXIV.1401.2209. URL: https://arxiv.org/abs/1401.2209.

Iyengar, J. and M. Thomson (May 2021). *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. RFC Editor, pp. 1–151. URL: https://www.rfc-editor.org/rfc/rfc9000.txt.

Iyengar, Jana (2021). *QUIC is now RFC 9000*. https://www.fastly.com/blog/quic-is-now-rfc-9000.

Lorenzi, Daniele et al. (2021). "Days of Future Past: An Optimization-Based Adaptive Bitrate Algorithm over HTTP/3". In: *Proceedings of the 2021 Workshop on Evolution, Performance and Interoperability of QUIC*. EPIQ '21. Virtual Event, Germany: Association for Computing Machinery, pp. 8–14. ISBN: 9781450391351. DOI: 10.1145/3488660.3493802. URL: https://doi.org/10.1145/3488660.3493802.

Lu, Feng et al. (2015). "CQIC: Revisiting Cross-Layer Congestion Control for Cellular Networks". In: *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*. HotMobile '15. Santa Fe, New Mexico, USA: Association for Computing Machinery, pp. 45–50. ISBN: 9781450333917. DOI: 10.1145/2699343.2699345. URL: https://doi.org/10.1145/2699343.2699345.

Marx, R., L. Niccolini, and M. Seemann (May 2022). *Main logging schema for qlog draft-ietf-quic-qlog-main-schema-02*. Tech. rep. IETF. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-qlog-main-schema.

Marx, Robin (Aug. 2021). "HTTP/3 From A To Z". In: URL: https://www.smashingmagazine.com/2021/08/http3-core-concepts-part1/.

Marx, Robin, Joris Herbots, et al. (2020). "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, pp. 14–20. ISBN: 9781450380478. DOI: 10.1145/3405796.3405828. URL: https://doi.org/10.1145/3405796.3405828.

Marx, Robin, Wim Lamotte, et al. (2018). "Towards QUIC Debuggability". In: *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. EPIQ'18. Heraklion, Greece: Association for Computing Machinery, pp. 1–7. ISBN: 9781450360821. DOI: 10.1145/3284850.3284851. URL: https://doi.org/10.1145/3284850.3284851.

Mehdi Yosofie, Benedikt Jaeger (June 2019). "Recent Progress on the QUIC Protocol". In: *Seminar IITM WS 18/19*. DOI: 10.2313/NET-2019-06-1_16. URL: https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2019-06-1/NET-2019-06-1_16.pdf.

MPEG (2019). *Dynamic Adaptive Streaming over HTTP*. https://www.iso.org/standard/79329.html.

Mueller, Christopher (2019). "MPEG-DASH (Dynamic Adaptive Streaming over HTTP)". In: *bitmovin*. URL: https://bitmovin.com/dynamic-adaptive-streaming-http-mpeg-dash/.

Nilsson, Jim and Tomas Akenine-Möller (2020). *Understanding SSIM*. DOI: 10.48550/ARXIV.2006.13846. URL: https://arxiv.org/abs/2006.13846.

O'Sullivan, John, Darijo Raca, and Jason J. Quinlan (2020). "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)*, pp. 185–187. DOI: 10.1109/WoWMoM49955.2020.00043.

Palmer, Mirko et al. (2021). "VOXEL: Cross-Layer Optimization for Video Streaming with Imperfect Transmission". In: *Proceedings of the 17th International Conference on Emerging Networking EXper-*

iments and Technologies. CoNEXT '21. Virtual Event, Germany: Association for Computing Machinery, pp. 359–374. ISBN: 9781450390989. DOI: 10.1145/3485983.3494864. URL: https://doi.org/10.1145/3485983.3494864.

Pauly, T., E. Kinnear, and D. Schinazi (Mar. 2022). *An Unreliable Datagram Extension to QUIC*. RFC 9221. RFC Editor.

Postel, J. (Aug. 1980). *User Datagram Protocol*. STD 6. http://www.rfc-editor.org/rfc/rfc768.txt. RFC Editor. URL: http://www.rfc-editor.org/rfc/rfc768.txt.

Qin, Yanyuan et al. (2018). "ABR Streaming of VBR-Encoded Videos: Characterization, Challenges, and Solutions". In: *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '18. Heraklion, Greece: Association for Computing Machinery, pp. 366–378. ISBN: 9781450360807. DOI: 10.1145/3281411.3281439. URL: https://doi.org/10.1145/3281411.3281439.

quicwg (2022). *base-drafts*. https://github.com/quicwg/base-drafts/wiki/Implementations.

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

Reselman, Bob (June 2021). "5 key HTTP/3 facts every Enterprise Architect needs to know". In: URL: https://www.redhat.com/architect/http3.

Robitza, Werner et al. (2018). "HTTP Adaptive Streaming QoE Estimation with ITU-T Rec. P. 1203: Open Databases and Software". In: *Proceedings of the 9th ACM Multimedia Systems Conference*. MMSys '18. Amsterdam, Netherlands: Association for Computing Machinery, pp. 466–471. ISBN: 9781450351928. DOI: 10.1145/3204949.3208124. URL: https://doi.org/10.1145/3204949.3208124.

Rodriguez, Demostenes Zegarra et al. (Dec. 2014). "The impact of video-quality-level switching on user quality of experience in dynamic adaptive streaming over HTTP". In: *EURASIP Journal on Wireless Communications and Networking* 2014, p. 216. DOI: 10.1186/1687-1499-2014-216.

Sandvine (Jan. 2022). *The Global Internet Phenomena Report 2022*. Tech. rep. URL: https://www.sandvine.com/hubfs/Sandvine_Redesign_2019/Downloads/2022/Phenomena%5C%20Reports/GIPR%5C%202022/Sandvine%5C%20GIPR%5C%20January%5C%202022.pdf.

Seufert, Michael (2019). "Fundamental Advantages of Considering Quality of Experience Distributions over Mean Opinion Scores". In: *2019 Eleventh International Conference on Quality of Multimedia Experience (QoMEX)*, pp. 1–6. DOI: 10.1109/QoMEX.2019.8743296.

Spiteri, Kevin, Rahul Urgaonkar, and Ramesh K. Sitaraman (2016). "BOLA: Near-optimal bitrate adaptation for online videos". In: *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pp. 1–9. DOI: 10.1109/INFOCOM.2016.7524428.

Stenberg, Daniel (2018). "HTTP/3 explained". In: URL: https://http3-explained.haxx.se/en/why-quic/why-tcphol.

Stohr, Denny et al. (2016). "QoE Analysis of DASH Cross-Layer Dependencies by Extensive Network Emulation". In: *Proceedings of the 2016 Workshop on QoE-Based Analysis and Management of Data Communication Networks*. Internet-QoE '16. Florianopolis, Brazil: Association for Computing Machinery, pp. 25–30. ISBN: 9781450344258. DOI: 10.1145/2940136.2940141. URL: https://doi.org/10.1145/2940136.2940141.

Yu, Yajun, Mingwei Xu, and Yuan Yang (2017). "When QUIC meets TCP: An experimental study". In: *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pp. 1–8. DOI: 10.1109/PCCC.2017.8280429.