ANALYSIS AND OPTIMIZATION OF STREAMING MEDIA QOE OVER QUIC ENABLED MODERN BROWSERS OVER VARIED NETWORK BANDWIDTH

Submitted By

Sapna Chaudhary

Roll Number: PhD20012

Submitted

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

to the

INDRAPRASTHA INSTITUTE OF INFORMATION TECHNOLOGY DELHI

July 2026

ANALYSIS AND OPTIMIZATION OF STREAMING MEDIA QOE OVER QUIC ENABLED MODERN BROWSERS OVER VARIED NETWORK BANDWIDTH

Submitted By

Sapna Chaudhary

Roll Number: PhD20012

Under the supervision of

Dr. Mukulika Maity

Dr. Sandip Chakraborty

INDRAPRASTHA INSTITUTE OF INFORMATION TECHNOLOGY
DELHI

July 2026

Declaration

This is to certify that the thesis titled *"Analysis and Optimization of Streaming Media QoE over QUIC-enabled Modern Browsers over Varied Network Bandwidth"* has been authored by me. It presents the research conducted by me under the supervision of Dr. Mukulika Maity and Dr. Sandip Chakraborty.

To the best of my knowledge, this thesis is an original work, both in terms of research content and narrative, and has not been submitted elsewhere, either in part or in full, for the award of any degree or diploma. Further, due credit has been given to the relevant state-of-the-art and collaborations (if any) through appropriate citations and acknowledgments, in accordance with established academic norms and practices.

Name: Sapna Chaudhary

Department of Computer Science and Engineering
Indraprastha Institute of Information Technology Delhi
New Delhi – 110020, India

Date: July 2026

Declaration Regarding the Use of AI Tools

I hereby declare that I am fully responsible for the entire content of this thesis. Any use of online tools, including Artificial Intelligence (AI)-based tools, was limited to assistance in language improvement, formatting, proofreading, or enhancing the presentation of the material. The research ideas, methodology, analysis, results, interpretations, and conclusions presented in this thesis are my own work and have been critically reviewed and verified by me.

I acknowledge that I bear full responsibility for the accuracy, originality, and integrity of the content of this thesis, including any sections where AI-assisted tools may have been utilized. I further accept full accountability for ensuring compliance with academic and publication ethics and for any ethical violations arising from the use of such tools.

Sapna Chaudhary
PhD20012
Ph.D. Scholar
Indraprastha Institute of Information Technology Delhi
New Delhi, India

CERTIFICATE

This is to certify that the thesis entitled *"Analysis and Optimization of Streaming Media QoE over QUIC-enabled Modern Browsers over Varied Network Bandwidth"* being submitted by **Sapna Chaudhary** to the **Indraprastha Institute of Information Technology Delhi**, for the award of the degree of **Doctor of Philosophy**, is an original research work carried out by her under my supervision. In my opinion, the thesis has reached the standards fulfilling the requirements of the regulations relating to the degree.

The results contained in this thesis have not been submitted, either in part or in full, to any other university or institute for the award of any degree or diploma.

Mukulika Maity

Date: July 2026

Place: New Delhi

Dr. Mukulika Maity
Department of Computer Science and Engineering
IIT Madras

Sandip

Dr. Sandip Chakraborty
Department of Computer Science and Engineering
IIT Kharagpur

Indraprastha Institute of Information Technology Delhi
New Delhi – 110020

Acknowledgements

I would like to express my deepest gratitude to my PhD advisors, Dr. Mukulika Maity and Dr. Sandip Chakraborty, for their constant guidance, encouragement, and unwavering support throughout my doctoral journey. Working under their supervision has been an immensely enriching experience. Dr. Mukulika Maity has been far more than an academic advisor to me; she has been a mentor who guided me with patience, taught me invaluable lessons, and helped me grow into an independent researcher. I am deeply grateful for her trust, motivation, and care throughout this journey. Before beginning my PhD, I had learned computer networks through NPTEL lectures by Dr. Sandip Chakraborty, and I never imagined that I would one day have the opportunity to learn directly from him. I feel truly fortunate to have received his guidance during my PhD. His teaching, insights, and mentorship have greatly influenced my research thinking and technical growth. I consider myself extremely lucky to have such a wonderful combination of advisors whose complementary guidance shaped both my academic and personal development during my PhD. I would also like to express my sincere gratitude to Dr. Arani Bhattacharya for his constant support and guidance during my PhD. Despite not being my formal advisor, he always supported me and offered valuable guidance during my PhD. I am also thankful to my evaluation committee members, Dr. Sambuddho Chakravarty and Dr. Sanjit Kaul, for their insightful comments, constructive feedback, and valuable suggestions during the evaluation process.

I am deeply indebted to my family for their unconditional love and support. I would like to thank my brother, Satish Kumar, who first planted the idea of pursuing a PhD in my mind and then supported me in every possible way throughout my education journey. My heartfelt thanks go to my husband, Prashant Kumar, who witnessed every phase of this journey closely and supported me with unwavering patience and care. His encouragement, understanding, and constant presence made this journey possible. Behind every milestone I achieved, his support played a crucial role. I also sincerely thank my sister-in-law, Rumee Chaudhary, for taking care of me and ensuring that I could focus on my studies without worry. Long before this journey began, it was my mother, Krishna Bala, who dreamed of seeing her daughter well educated, a dream she could not fulfill for herself. I am grateful to have worked toward realizing her dream, and today, seeing happiness and pride in her eyes makes every struggle worthwhile. I am equally grateful to my sister, Sangeeta Duhan, who always reminded me how far I had come and motivated me to keep moving forward, emphasizing the importance of education for women. I would also like to express

my sincere gratitude to my husband's grandfather, Gulbeer Singh, for his trust, blessings, and constant prayers for my success. I am deeply thankful to my mother-in-law, Dr. Shashi Bharti, for treating me like her own daughter, supporting me throughout my PhD, and celebrating my achievements as her own. I would also like to thank my sister-in-law, Aparna Chikara, for her exceptional care and support during my illness with tuberculosis during my PhD; her help played a vital role in my recovery.

I am grateful to my lab colleagues and friends, Shubham Chaudhary, Vijay Kumar Singh, Jyoti Shokhanda, Najia Naj, Saswati Paramita and Manpreet Kaur, and to my seniors, Soumyajit Chatterjee, Nidhi Goel, Devashish Gosain, and Piyush Kumar Sharma. Their support, discussions, and encouragement at different stages of my PhD made this journey both productive and enjoyable.

I also feel fortunate to have worked with wonderful undergraduate students who taught me as much as I guided them. I sincerely thank Prince Sachdeva, Abhivandan Pandey, Naval Kumar Shukla, Chirag Bansal, Chirag Kumar Banka, Alvin Joseph Ambooken, Lakshay Chauhan, Ankit Kumar, Vinayak Goel, Kartikeya Sehgal, Tanishq Goyal, and Aashi for their enthusiasm, dedication, and contributions.

I would also like to thank the IT team, as well as all the students, faculty members, and technical and non-technical staff with whom I collaborated during my research projects. Special thanks to the IIIT Delhi administration, IT services, and facility management staff for providing a supportive and hassle-free research environment.

Finally, I would like to once again express my heartfelt gratitude to my advisors, my parents, my family, my in-laws' family, and my friends, whose constant belief in me gave me the strength to complete this journey.

"Gratitude is not only the greatest of virtues, but the parent of all others."
— Cicero

Abstract

Today, global Internet traffic is dominated by video streaming, prompting service providers to adopt emerging technologies that enhance users' Quality of Experience (QoE) while sustaining operational efficiency. In this landscape, Google introduced QUIC in 2013, a transport-layer protocol engineered to improve web performance and video delivery, especially in environments characterized by high packet loss, variable throughput, and constrained bandwidth. At the same time, the rapid deployment of high-speed access technologies such as WiFi 6/7 and 5G-capable of delivering hundreds of megabits to multiple gigabits per second is reshaping the Internet delivery ecosystem. These advancements create a critical need to evaluate whether QUIC can consistently meet the demands of both constrained and high-speed networks. Although QUIC promises improved performance, its behavior in real-world, heterogeneous environments, across browsers and network conditions is still underexplored, which directly impacts video streaming's QoE.

Recognizing these limitations and developments, this thesis primarily focuses on understanding how QUIC is implemented within modern web browsers and how these implementation choices influence video streaming performance. To enable this investigation, we develop a specialized measurement tool and collect a large-scale dataset across diverse network conditions, allowing us to systematically analyze browser-specific QUIC behavior in real-world streaming scenarios. To this end, the thesis contributes a measurement tool and a large-scale dataset for analyzing QUIC's real-world behavior along with the design of an intelligent, browser-integrated decision system that leverages network parameters to both dynamically control connection racing and select between QUIC and TCP based on prevailing network conditions, thereby improving video streaming performance. Specifically, this thesis investigates two fundamental questions:

- How do browser-specific implementations of QUIC, in particular their protocol selection mechanisms such as connection racing, impact video streaming performance under varied network conditions?
- Given QUIC's performance limitations under certain network conditions, is it viable for browsers to rely on a single default transport protocol? Is there a need for an adaptive protocol selection mechanism?

To answer these questions, we begin by examining the current browser implementation, where most modern browsers support HTTP/3 and rely on QUIC, which runs on top of UDP as the default transport protocol. Many Internet middleboxes either block or rate-limit UDP traffic, which can hinder QUIC connectivity. To ensure reliability and backward compatibility, browsers use a strategy known as *connection racing*, where both QUIC and TCP connections are initiated in parallel. The protocol that completes the handshake first is selected for subsequent communication. This raises a critical question about the conditions under which QUIC truly provides advantages over TCP, and

when TCP may still be the preferable option. To study this in detail, the thesis develops an HTTP/3-compatible browser-based dataset collection tool, H3B, which captures both application-level and network-level logs during YouTube streaming. Using H3B, we conduct a large-scale measurement study across popular browsers (Chrome/Chromium and Firefox), covering 6013 YouTube sessions and 5474 hours of playback. Our analysis reveals that YouTube QoE often degrades due to repeated connection racing events. We modify the Chromium source code to disable connection racing, and we observe a significant improvement in QoE. Building upon this finding, we design two complementary solutions: (i) an adaptive mechanism to decide when connection racing should be enabled, and (ii) a server-side emulation framework to mitigate protocol switching impact. Instrumenting the Chromium codebase further shows that selectively disabling or controlling connection racing substantially enhances QoE by 60% compared to the baseline implementation.

While this addresses the challenges arising from QUIC's connectivity issues and the instability introduced by racing, our experiments, along with evidence from prior studies, show that it continues to exhibit performance limitations in high-speed networks. This insight naturally leads to the second question: Is it appropriate for browsers to rely on a single default transport protocol across all network scenarios? Hence, In this thesis, to address this, we propose IntelSwitch a novel client-side framework that enables adaptive switching between TCP and QUIC for video streaming. It does not require any server-side modifications. IntelSwitch continuously monitors network and QoE-related metrics through a DASH player and employs a reinforcement learning (RL) based decision engine to dynamically select the most suitable transport protocol based on real-time network and host conditions. The framework comprises two components: (i) modifications to the DASH player and Chromium browser to support seamless protocol switching during playback, and (ii) an RL-based decision module that determines the optimal protocol under varying network and system states. We implement IntelSwitch by extending Dash.js and the Chromium browser with protocol-aware playback. We found that adaptive transport protocol switching within the browser leads to an average improvement of up to 114% over choosing only QUIC as the transport protocol across diverse network conditions.

Collectively, this thesis presents a comprehensive investigation into protocol performance evolution from TCP to QUIC to HTTP/3, identifies critical challenges in browser protocol management, and introduces an intelligent, adaptive framework to enhance streaming QoE across heterogeneous network conditions.

Contents

List of Figures

List of Tables

Abbreviations

RTT	**R**ound **T**rip **T**ime
QoE	**Q**uality of **E**xperience
TCP	**T**ransmission **C**ontrol **P**rotocol
UDP	**U**ser **D**atagram **P**rotocol
HTTP	**H**yper **T**ext **T**ransfer **P**rotocol
RQ	**R**esearch **Q**uestion
DRL	**D**eap **R**einforcement **L**earning
eBPF	**E**xtended **B**erkeley **P**acket **F**ilter

Chapter 1

Introduction

"A person who never made a mistake never tried anything new."
— Albert Einstein

A significant portion of today's Internet traffic is dominated by video, accounting for nearly 65% of global traffic [1]. Each year, global internet traffic continues to grow at a remarkable pace. According to the 2024 Global Internet Phenomena Report [2], daily internet usage has increased by 33 exabytes, with average per-user consumption rising by 4.2 GB per day. This rapid growth in Internet traffic and in bandwidth-intensive applications has intensified the need for transport technologies that can sustain high performance and deliver superior Quality of Experience (QoE). One major advancement in this direction is QUIC, a transport-layer protocol originally developed by Google in 2012 and later standardized as IETF QUIC in 2021 to enhance the performance of web and video streaming, particularly in high-loss or bandwidth-constrained environments. QUIC is developed as a user-space transport protocol that utilizes UDP as its underlying transport protocol. QUIC employs a cryptographic handshake designed to minimize connection setup latency by leveraging cached server credentials for repeated connections and eliminating redundant handshake overhead across multiple layers of the network stack. In addition, QUIC mitigates head-of-line blocking by introducing lightweight multiplexed streams within a single connection, ensuring that packet loss affects only the streams carrying the lost data rather than blocking the entire connection. Later in 2022, QUIC standardization was followed by the standardization of the application layer protocol, HTTP/3. Since then, HTTP/3 has been supported by the majority of browsers, content delivery networks (CDNs), and several major internet companies, including Google,

Facebook, Akamai, Cloudflare, and Apple. Recent measurements from APNIC Labs show that QUIC over HTTP/3 adoption on the public Internet has reached substantial levels: only about 15–20 % of users employ QUIC on the initial fetch, but over 70 % of users use QUIC for subsequent fetches once it is triggered [3]. Geographic analysis further indicates that QUIC usage exceeds 75% in most economies, although it remains comparatively low in regions such as China (approximately 15%), with intermediate adoption observed in Morocco, Iran, and Nigeria [3].

However, the adoption of HTTP/3 within browsers presents an important challenge. QUIC operates over UDP, and, as noted in [4], many legacy middleboxes still block or rate-limit UDP traffic due to long-standing security concerns [5, 6, 7]. Interestingly, a Cloudflare study [8] reports that HTTP/3 traffic share dropped to approximately 26% in November 2022. Similar trends were observed in Russia and Iran, where the share of HTTP/3 traffic fell significantly to only 10% and 2.1%, respectively, during the measurement period [6]. The authors attribute this decline to the deployment of middleboxes that block or throttle QUIC traffic. To maintain connectivity in the presence of such middleboxes, HTTP/3-enabled browsers employ a backward-compatibility strategy in which they initiate *connection racing* between the legacy TCP protocol and the newer QUIC protocol [5, 6, 7]. Notably, this backward-compatibility mechanism is explicitly designed for scenarios where middleboxes hinder QUIC connectivity. Moreover, unlike TCP, which is implemented in the highly optimized kernel networking stack, browsers such as Chrome/Chromium and Firefox implement QUIC entirely in user space.

Despite this crucial architectural difference, prior studies [9, 10, 11, 12, 13, 14] have paid limited attention to how browser-specific implementation decisions can impact QUIC's performance for video streaming applications. They [11, 15, 16, 17] compared QUIC and TCP for web and video streaming applications, with several studies [9, 12] reporting that QUIC can improve streaming performance under lossy and variable network conditions. At the same time, other studies [16, 17] have shown that QUIC does not always consistently outperform TCP, particularly in stable or high-bandwidth environments. Existing works [18, 19] have also proposed enhancements to QUIC to improve multimedia delivery and application QoE. However, these studies focus primarily on protocol-level behavior and treat browser implementations as black boxes. Consequently, limited attention has been paid to understanding how browser-level mechanisms, such as connection racing and protocol fallback behavior, influence the performance of HTTP/3 video streaming applications in real-world environments. Hence, this gap between what is implemented in the

browser and what implications it has on the popular transport protocol motivates the central question of this thesis:

> *Do browser-level backward compatibility mechanisms such as connection racing between QUIC and TCP have a measurable impact on video streaming performance when QUIC is used as the underlying transport protocol? If so, how can this impact be systematically quantified and analyzed? Furthermore, is this impact always detrimental to application QoE, or can such mechanisms be intelligently leveraged to enable adaptive protocol switching based on prevailing network conditions?*

Answering these questions is non-trivial, as it requires a controlled experimental setup and large-scale statistical analysis. Real-world network measurements are inherently noisy, with multiple interacting factors simultaneously influencing application performance. In addition, gaining meaningful insights necessitates a deep understanding of the source code and internal behavior of popular web browsers, as well as the ability to statistically correlate browser-level mechanisms with observed application-level outcomes.

This thesis addresses these challenges by systematically examining the benefits and limitations of QUIC for video streaming, with a particular emphasis on how browser-level implementations influence performance under both constrained and high-speed network conditions. Existing tools and frameworks [20, 21] for adaptive streaming and network evaluation primarily focus on collecting QoE-related measurements for YouTube streaming applications using application statistics and packet traces. However, these tools are mainly designed for application-level QoE analysis in mobile streaming environments and provide limited visibility into browser-level transport behavior and protocol dynamics. In particular, they do not support fine-grained synchronized analysis of browser-controlled mechanisms such as connection racing, protocol fallback between QUIC and TCP, and transport protocol switching in modern HTTP/3-enabled browsers. These limitations motivate the need for a browser-centric measurement framework capable of collecting synchronized fine-grained application- and network-layer measurements for systematic analysis of transport protocol behavior and its impact on video streaming QoE. To this end, this thesis designs a dedicated measurement framework to collect fine-grained application- and network-layer data and perform extensive statistical analysis on the collected dataset. Building on these insights, the thesis further introduces an adaptive transport protocol switching framework that leverages browser and network signals to improve video streaming QoE.

1.1 Motivation

In this section, we discuss the key motivations that shape the research problems addressed in this thesis. Specifically, we highlight three interlinked aspects: (i) the need for a structured tool capable of collecting synchronized application-level and network-level logs for different network conditions, (ii) the need for systematic statistical analysis to quantify the impact of browser-level connection racing on application QoE when QUIC is used as the transport protocol and to determine when to enable connection racing, and (iii) the need for an adaptive transport protocol switching mechanism, motivated by evidence that no single transport protocol consistently delivers optimal performance across diverse network conditions.

1.1.1 Need for a Data Collection Tool

To understand how browser-level implementations affect the performance of the user-space transport protocol QUIC for applications such as video streaming, it is essential to analyze video streaming behavior using large-scale, representative datasets. Such analysis requires synchronized application-level and network-level logs to accurately correlate observed QoE degradation with underlying transport- and browser-level behavior. In particular, there is a need to first understand how video streaming applications perform under poor and highly variable network conditions when using QUIC, and how this behavior compares to that of the legacy TCP-based protocol. However, to the best of our knowledge, existing datasets either do not capture poor or highly variable network conditions at scale or lack fine-grained, synchronized measurements of both applications and networks for video streaming applications. This gap motivates the need for a well-structured, generalized measurement tool capable of collecting large-scale datasets that jointly capture application and network behavior under challenging network conditions, enabling a systematic comparison of video streaming performance over QUIC and legacy transport protocols.

1.1.2 Browser-Level Connection Racing and Its Impact on Video Streaming QoE

Popular browsers such as Chrome/Chromium and Mozilla Firefox implement built-in fallback mechanisms to handle situations in which QUIC experiences repeated handshake

failures or persistent connection timeouts. In such cases, browsers fallback to the legacy transport protocol, TCP, to ensure continuity of data transfer. This behavior is largely transparent to users, who primarily care about achieving good QoE, particularly for video streaming applications.

The fallback mechanism is primarily designed for scenarios in which QUIC connectivity is impaired due to middleboxes that block or rate-limit UDP traffic. While this design choice improves robustness by preserving connectivity in the presence of such middleboxes, it can introduce unintended side effects if the browser does not explicitly differentiate between QUIC degradation caused by UDP blocking and degradation caused by poor or congested network conditions. In both cases, the browser may observe similar signals, such as increased RTTs or connection errors, and respond by triggering protocol switching. Note that unnecessary protocol switching can lead to frequent transitions between QUIC and TCP, introducing instability and negatively impacting video streaming QoE. Therefore, it is crucial to understand *how browser-level decisions influence application performance, and to what extent connection racing contributes to QoE degradation.* A closely related and critical follow-up question is: *when connection racing should be triggered, do browsers differentiate between QUIC degradation caused by adverse network conditions and that caused by deliberate UDP rate limiting?* Addressing these questions requires a detailed examination of the connection racing implementation and decision logic within the Chromium browser, along with the design of mechanisms that can intelligently decide when connection racing should be enabled or disabled to mitigate its impact on application QoE.

1.1.3 Limitations of Default Transport Protocol Selection and the Need for Adaptive Switching

Modern browsers such as Chrome/Chromium use QUIC as the default transport protocol for data transfer. With the rapid deployment of high-bandwidth access technologies such as Wi-Fi 6/7 and 5G networks capable of delivering hundreds of Mbps to multi-Gbps throughput, transport protocols are required to efficiently utilize high-capacity links. While the legacy transport protocol TCP is known to suffer from head-of-line (HoL) blocking at the transport layer, particularly under lossy network conditions, this does not imply that TCP performs poorly in all scenarios. In fact, prior studies [12, 16] have shown that TCP can outperform QUIC in certain environments, especially in high-speed and stable

networks. Conversely, although QUIC offers clear advantages in lossy or highly variable networks, it has been reported to underperform in high-bandwidth settings due to host-side processing overhead [22] and browser-level implementation factors such as connection racing.

These observations indicate that selecting a single transport protocol as the default choice is insufficient to consistently deliver optimal performance across diverse network environments. More importantly, this raises a fundamental question: *Is relying on a default transport protocol within the browser sufficient, or is there a need for an adaptive transport protocol selection framework that can dynamically choose the most suitable protocol based on prevailing network conditions?* Addressing this question is essential for ensuring robust and consistently high QoE for video streaming applications across heterogeneous network scenarios.

All these together motivate the need for:

- A structured data collection tool and dataset that synchronously captures both application-level and network-level logs on a large scale for characterizing the video streaming performance under poor/variable network.
- A detailed analysis of the video streaming dataset by correlating application-level QoE metrics with network-level measurements. A need to identify and quantify the impact of browser-level connection racing on video streaming performance under poor network conditions, and design solutions to address identified limitations or implementation issues.
- A framework that adaptively selects the most suitable transport protocol within the browser based on prevailing network conditions to optimize the user's application QoE.

1.2 Research Questions

Bringing these motivations together, this thesis is structured around three research questions, which are addressed in the subsequent sections in this thesis. The research questions are as follows:

RQ1: *How is video streaming QoE under poor and variable network conditions in modern browsers?*

RQ2: *How frequently does connection racing occur in QUIC-enabled browsers, how does it impact video streaming QoE, and can a dynamic browser-level control of connection racing mitigate this impact?*

RQ3: *Is relying on a single transport protocol sufficient for video streaming across heterogeneous networks, or is adaptive protocol switching needed to improve QoE*

Together, these research questions guide the empirical analysis and system design presented in the remainder of this thesis. RQ1 and RQ2 focus on understanding the prevalence and impact of browser-level connection racing on video streaming QoE, while RQ3 explores whether adaptive transport protocol selection can address the limitations identified in earlier analyses and prior works. The subsequent chapters systematically address these questions through large-scale measurement, detailed analysis, and the design and evaluation of adaptive solutions.

1.3 Objectives of This Thesis

Guided by the motivations and research questions outlined in Section 1.1, this thesis pursues three primary objectives. Each objective addresses a distinct aspect of transport protocol behavior and browser implementation for video streaming, while collectively reflecting a unified goal: to understand, measure, and improve video streaming Quality of Experience (QoE) in real-world Internet environments. Specifically, these objectives span large-scale measurement and analysis of browser behavior, identification of protocol-level and implementation-driven performance bottlenecks, and the design and evaluation of adaptive mechanisms that intelligently select transport protocols based on network conditions.

1.3.1 Objective 1: To design and develop a browser-based data collection tool and collect large dataset for QUIC-enabled browsers

The primary objective of this thesis is to design and develop a structured and reproducible data collection framework that can synchronously capture application-level QoE metrics and network-level logs in QUIC-enabled browsers under both controlled and real-world network conditions. Leveraging this framework, we aim to collect large-scale datasets for video streaming applications, particularly in a poor network scenario.

1.3.2 Objective 2: To investigate the frequency, causes, and QoE implications of browser-level connection racing in QUIC-enabled browsers

The second objective of this thesis is to systematically understand the implementation of connection racing within the browser. To analyze how frequently browser-level connection racing occurs in QUIC-enabled browsers, understand the conditions under which it is triggered, and quantify its impact on video streaming QoE. Using the large-scale dataset collected with our data collection tool, we aim to characterize protocol switching behavior during HTTP/3 streaming and to examine does browsers fallback to TCP even in the absence of explicit UDP blocking or rate limiting. Through detailed QoE analysis and statistical hypothesis testing, we investigate how connection racing impacts key performance metrics, including streaming bitrate, rebuffering events, and bitrate stability. Furthermore, by examining browser implementation details and network-level signatures, we identify scenarios where connection racing is unnecessarily triggered, resulting in degraded application performance. This analysis forms the basis for motivating and designing a solution that can intelligently control connection racing and mitigate its adverse impact on video streaming QoE. The key intuition in designing the solution is that even though a UDP rate-limiting middlebox and a poor network show a similar signature to browsers regarding RTT and browser-level errors, TCP remains unaffected by UDP rate-limiting middleboxes. To operationalize this idea, we need to modify the Chromium source code.

1.3.3 Objective 3: To design and evaluate adaptive protocol switching mechanisms for improving video streaming QoE

The third objective of this thesis is to design and evaluate adaptive transport protocol switching mechanisms that mitigate the negative impact of protocol switching and improve video streaming QoE. We aim to develop a framework that adapts protocol usage in real-time based on observed network conditions and application context within the Chromium browser. Specifically, we design and implement a reinforcement learning based adaptive transport protocol switching framework that dynamically selects between TCP and QUIC to optimize streaming QoE across heterogeneous network environments. We evaluate this framework under diverse network scenarios to assess its effectiveness in improving bitrate stability, reducing rebuffering, and enhancing overall user QoE.

1.4 Contributions of the Thesis

We now summarize the key contribution of this thesis as follows:

1.4.1 Design and Development of a Tool for Data Collection

- We develop the *H3B* tool, an emulation-based toolbox that captures application performance statistics for YouTube video streaming, along with the underlying network traffic traces. We use H3B to launch a measurement campaign for YouTube across 5 different geographical locations and 5 different bandwidth patterns. Out of them, 2 are traces collected under mobility in WiFi and cellular networks. Specifically, we focus on poor and variable network bandwidth patterns, as there is minimal data on the performance of video streaming applications in such networks [23, 24].
- Using this tool, we collected a dataset of over 6013 YouTube sessions covering 5474 hours of streaming (228 days) over 5 different geographical locations and 5 different bandwidth patterns using both HTTP/3 and HTTP/2. We have made both our tool and dataset open-source [1]. The collected dataset allows us to compare and evaluate

[1] `https://github.com/NKShukla/H3B` (Access: July 2026)

the impact of browser configurations on video streaming QoE, where performance also depends on the underlying network conditions.

1.4.2 Measurement and Mitigation of Browser-Level Connection Racing Effects on YouTube Video Streaming QoE

- We analyzed how the use of a QUIC-enabled browser impacts the performance of YouTube compared to the legacy QUIC-disabled browser. The analysis is done over 6013 video sessions combining more than 5474 streaming hours over two popular browsers, namely Chrome/Chromium and Mozilla Firefox, which includes (i) emulation over different bandwidth patterns, (ii) in-the-wild data collection, and (iii) over real network traces. We observe that enabling QUIC in the browser does not consistently improve the QoE; instead, YouTube suffers over a QUIC-enabled browser when network quality fluctuates or the network bandwidth is low.

- We explore the code flow of the Chromium (the open-source implementation of Google Chrome) and Firefox browsers' implementation of the connection racing mechanism. We validate our observation that streaming applications suffer in a poor network due to the browser's implementation of connection racing. We made necessary modifications to the Chromium browser source code to forcibly stop connection racing in cases where a protocol is affected by a middlebox or a poor network. We observe that the *Modified Browser* (when the data is transmitted over a pure QUIC stream with disabled connection racing) outperforms the *Original Browser* (with the connection racing option) for more than 60% cases.

- Next, we implement a dynamic solution, namely *Dynamic-Racing*, for deciding when to enable connection racing. We made this implementation and data open-sourced[2]. We evaluate our solution in poor networks and in the face of UDP rate-limiting firewalls. We observe that it correctly enables connection racing in the case of a firewall and disables it in the case of a poor network. Our solution improves the YouTube video QoE compared to the original browser. We then propose an eBPF-based solution to minimize the impact on application performance during protocol switching. Our solution is based on the principle of transferring congestion state during protocol switching between QUIC and TCP.

[2] `https://github.com/sapna2504/Chromium-Modification-V2` (Access:July 2026)

1.4.3 Characterizing Transport Protocol Performance across Diverse Network Conditions for Video Streaming and Designing an Adaptive Protocol Switching Framework

- We demonstrate that the QoE of video streaming applications can significantly degrade on high-speed links when QUIC is used as the transport protocol, primarily due to client-side processing overhead. Through a series of pilot experiments, we observe that no single static transport protocol is optimal across all network conditions.

- We propose IntelSwitch, a deep reinforcement learning (DRL) based framework that dynamically selects a suitable transport protocol (TCP or QUIC) for each video segment based on network conditions and application QoE. We implement IntelSwitch by extending both the `dash.js` player and the Chromium browser to enable adaptive protocol switching. We contributed about 140 and 503 lines of code to `dash.js` and the Chromium browser, respectively. We have open-sourced our implementation[3].

- We conduct extensive evaluations of IntelSwitch under 9 diverse network patterns covering varied types of networks, from controlled to real 5G networks. We compare its performance against static protocol choices (TCP and QUIC) and a Heuristic baseline. IntelSwitch improves the application QoE by upto 47% over only TCP, 114% over only QUIC and 58% over the Heuristic baseline.

[3] `https://github.com/sapna2504/IntelSwitch-adaptive-protocol-switching/`

1.5 Organization of the Thesis

We now summarize the work in this thesis as follows

- Chapter 2 provides a brief overview of the browser's implementation of connection racing, followed by a survey of related work on publicly available video streaming datasets, prior studies analyzing QUIC's performance for video streaming applications, the impact of middleboxes on video streaming performance, and existing approaches for adaptive transport protocol selection.
- Chapter 3 discusses the design of our data collection tool H3B and the dataset collected using the tool.
- Chapter 4 presents a comprehensive analysis of the collected dataset spanning diverse geographical regions and heterogeneous network conditions, with the objective of quantifying how connection racing influences YouTube streaming performance. Furthermore, this chapter introduces two complementary solution directions: (i) a *Dynamic-Racing*-driven strategy for determining when to activate connection racing, and (ii) an eBPF-based design for mitigating the impact of protocol switching on application-level QoE by enabling efficient congestion state transfer.
- Chapter 5 presents the motivation, key design challenges, and an architectural overview of the IntelSwitch system, along with implementation details for each of its components. Additionally, it describes the IntelSwitch evaluation setup and provides a comprehensive performance assessment across diverse network conditions and multiple ABR algorithms.
- Finally, Chapter 6 concludes the thesis and outlines future directions, including the development of a fully adaptive framework that operates effectively across diverse network conditions, integrates with various ABR algorithms, and remains compatible with different congestion control strategies.

Chapter 2

Related Work

"Excellence is a continuous process, not an accident."
— A. P. J. Abdul Kalam

In this chapter, we first provide a brief description of connection racing implementation within the browsers (2.1), which will help in understanding the thesis. We then review prior work on QoE datasets, followed by studies examining video streaming over QUIC and the impact of middleboxes on application performance. Finally, we discuss related work on QUIC performance in high-speed networks and existing adaptive protocol switching frameworks for web services.

2.1 Background: Browser's Implementation of Connection Racing

Chromium Implementation [25]: Fig. 2.1 illustrates how connection racing is implemented inside the Chromium browser. The browser first checks its cache to determine whether the YouTube server has previously advertised QUIC as an alternate service. If no such information is found, the browser initiates a main job that establishes a TCP connection to query the server for QUIC support. This support for an alternative service is conveyed through the `alt-svc` header. If supported, the browser first adds this information about the YouTube server inside the cache. The browser then creates a new QUIC connection via another job, namely the `alternate` job, and blocks the `main` job to prioritize

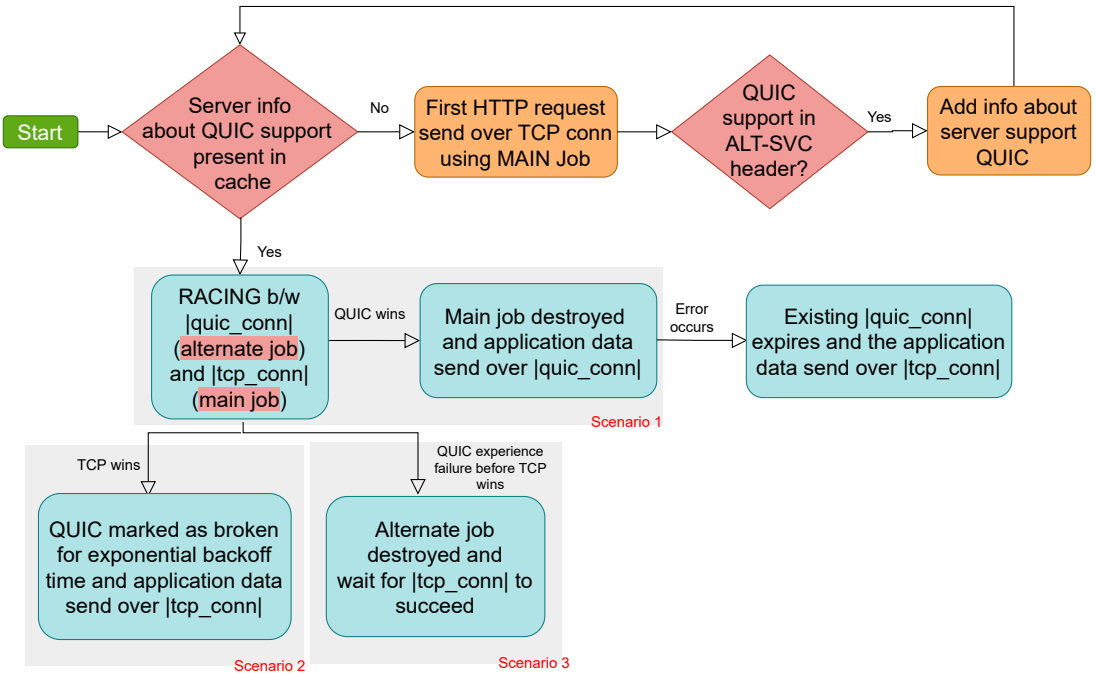


FIGURE 2.1: Connection Racing Implementation of Chromium Browser

the QUIC connection. Then, within 3 seconds (based upon the RTT value), it reinitiates a TCP connection by resuming the `main` job (uses pre-existing connection if present) [9, 26] and races the TCP connection with the QUIC connection. ***Scenario 1***: If the `alternate` job can successfully establish a QUIC connection, then the `main` job is destroyed. The application data is sent over QUIC by associating a data stream with the QUIC connection. Otherwise, ***Scenario 2***: QUIC is marked as broken for a duration determined by an exponential backoff, and the application data is then sent over TCP. Further, ***Scenario 3***: if QUIC experiences failure before TCP wins the race, the browser destroys the `alternate` job and waits for the TCP connection to succeed. Note that even if a job is destroyed, the connection remains existent. Moreover, if the browser experiences a stream error [27] such as `QUIC_STREAM_CONNECT_ERROR` or `QUIC_STREAM_REQUEST_INCOMPLETE` while associating the data stream with the `alternate` job, it waits for the TCP connection to succeed and then binds the application data with the `main` job. Furthermore, if, for any reason, the browser (idle application, poor bandwidth, etc.) fails to communicate with the server within the idle timeout duration (default: 29 seconds), the existing QUIC connection expires. The next time the browser wants to communicate with the server, it needs to create a new QUIC connection and repeat all the above steps.

Mozilla Firefox Implementation [28]: In the case of Firefox, a browser gets to know

about QUIC support either via `alt-svc` header or `HTTPSSVC DNS` record. If QUIC is supported by the YouTube server, it tries to establish a QUIC connection. If the HTTP/3 transaction is not completed within 100 ms duration (due to handshake failure/connection timeout/longer RTT), the browser either creates or reuses an existing backup connection (TCP). Once the backup connection is ready, the browser immediately shifts the transaction to the backup connection. Note that the backup connection may or may not be with the same server supporting QUIC. In case of failure or connection timeout, before QUIC handshake completion, the browser adds the server domain to the HTTP/3 excluded list (maintained by the browser). Hence, further, QUIC connection will not be tried with this server unless the browser window is closed and the list gets reset.

Connection racing can influence application performance. To analyze its impact on application QoE, particularly for video streaming, it is important to study various datasets collected from real video streaming sessions. Therefore, in the next section, we begin by reviewing various video streaming datasets relevant to this study.

2.2 Video Streaming Datasets

Several researchers have studied YouTube data collection and video QoE in various network environments. Gutterman et al. [23] predicted video QoE indicators, including buffer state, video quality, and stalling events, using a dataset of 425 YouTube sessions collected over WiFi and LTE in both static and mobile scenarios. Karagkioules et al. [20] collected approximately 374 hours of YouTube playback on mobile devices using their Wrapper-app tool, capturing both network- and application-layer logs, including YouTube Stats for Nerds and DNS queries. Their experiments spanned controlled bandwidth levels of 500 kbit/s, 1024 kbit/s, 3000 kbit/s, and 100 kbit/s. Wassermann et al. [24]utilized YoMoApp [21] to collect YouTube datasets from over 360 mobile users across 70 cellular operators, resulting in more than 3,000 video sessions. Their tool replicated the core functionalities of the native YouTube application to enable controlled measurements.

Beyond YouTube-specific studies, prior work has also produced several video QoE datasets designed to support rate adaptation in DASH streaming. These efforts typically employ either subjective or objective quality assessment techniques. Subjective assessment datasets involve user-perceived video quality scores such as MOS. For instance, Chen et al. [29] modeled time-varying subjective quality (TVSQ), while Duanmu et al. [30] introduced

a QoE index capturing instantaneous quality degradation and stalling impacts. Bampis et al. [31] created a Netflix-based dataset with distorted videos generated through controlled compression and rebuffering events, simulating realistic bandwidth variations; user studies were conducted to obtain MOS per frame index. Objective assessment datasets compute metrics such as PSNR and SSIM. Bampis et al. [32] released a dataset of 420 distorted videos with continuous and retrospective MOS, PSNR, and SSIM scores, along with metadata such as playback bitrate and rebuffering count.

We compare the datasets collected by prior work in Table 2.1. While these datasets provide valuable insights, they exhibit important limitations. First, they do not sufficiently capture the poor or highly variable network conditions commonly observed in developing regions. Second, key QoE metrics are sometimes missing: for example, [20] does not record stalling information, and [24] reports only resolution instead of bitrate, limiting the granularity of quality assessments. Third, some datasets do not rely on YouTube's production endpoints [21], reducing their representativeness. Fourth, most datasets rely on short video clips, typically under a few minutes in length, which are insufficient for studying long-lived streaming behavior or the impact of repeated protocol switching events over time. Lastly, several of them rely on artificial distortions in the collected dataset by artificially inserting stalls, which does not accurately reflect the real impact of network conditions on video streaming. Finally, none of them captures the browser's fallback behaviour. For these reasons, existing datasets were insufficient for evaluating browser-level QUIC behavior, protocol connection racing mechanisms, or transport protocol switching. This motivated us to create our comprehensive dataset, designed specifically to support in-depth analysis of transport protocol behavior for modern video streaming applications along with modern browsers. Hence, our dataset includes videos of approximately 4000 seconds each, capturing the full dynamics of buffer, bitrate, and protocol interaction in video streaming. Since the earlier dataset lacks fine-grained temporal logs, we fill this gap by capturing high-resolution application-level metrics (bitrate, stall duration, and buffer length) and network layer logs, enabling detailed, packet-level, and QoE-level correlation. Moreover, our dataset is collected under real bandwidth variability across diverse geographical and network settings, providing realistic insights into the connection racing mechanism within a browser.

In addition to the above-mentioned studies, several other prior works have examined the performance of video streaming applications. Next, for this thesis, we concentrate on the subset of these studies that evaluate QUIC's role as the transport protocol, as they directly relate to our problem context.

TABLE 2.1: Comparison of Prior YouTube and QoE Datasets with Our Dataset

Work	**Distortion Type**	**Video Duration**	**Logs**	**HTTP / QUIC Version**	**Temporal QoE Detail**
Gutterman et al. (2020) [23]	Real network conditions	Short, varied clips	Packet traces (TCP/QUIC)	HTTP/2 and Google QUIC	QoE inferred via ML; no fine-grained logs
Karagkioules et al. (2018) [20]	Real network conditions	559–734 s (avg. 600 s)	DNS + Stats for Nerds; no stall info	HTTP/2 and Google QUIC	Aggregated; resolution only
Wassermann et al. (2019) [24]	Real network (crowd-sourced)	Varied, user-driven	QoE KPIs (stall, delay, resolution)	HTTP/1.1 / HTTP/2 (no QUIC)	Coarse-grained QoE events
Chen et al. (2014) [29]	Artificial distortion (compression, stalls)	Short clips (< 300 s)	×	HTTP-based HAS	Subjective MOS; no network traces
Duanmu et al. (2016) [30]	Artificial distortion (fixed stalls)	Short clips	×	HTTP-based HAS	QoE index; fixed stall patterns
Bampis et al. (2017) [31]	Artificial distortion (compression + rebuffering)	Short Netflix clips	×	HTTP-based HAS	Frame-level MOS; simulated conditions
Bampis et al. (2021) [32]	Artificial distortion (compression)	Short distorted clips	×	HTTP-based HAS	Aggregated metrics; one stall per video
Thesis Dataset	**Real bandwidth variability**	**∼4000 s per video**	**application + network logs**	**HTTP/2 and HTTP/3 (IETF QUIC)**	**Fine-grained QoE (bitrate, stalls, buffer, switching)**

2.3 Performance Analysis of Video Streaming over QUIC

So far, we have discussed prior work on the collected datasets available over QUIC. We now discuss prior work analyzing the performance of video streaming applications when QUIC is used as the transport protocol.

In the seminal paper from Google on QUIC, Langley *et al.* [9] discussed the development and design of the QUIC protocol. They observed that QUIC reduces the re-buffering rates on YouTube by 18% on desktop and 15% on mobile. They also stated that QUIC benefits the users even when the network RTT is high. Engelbart *et al.* [10] have shown that QUIC congestion control needs to be tuned to support QoE for multimedia traffic. Similarly, a few other works [33, 34, 35, 36] in the literature have also adopted the QUIC protocol to support improved streaming performance. Recently, Seufert *et al.* [11] compared the performance of TCP and QUIC over 900 YouTube video streaming sessions in terms of initial playback delay, video quality, and stalling. They performed a statistical analysis and did not observe a significant improvement in QoE QUIC. However, the authors did not conduct any root cause analysis for this behavior. Interestingly, the authors picked only 500 out of 900 videos, only the streams that contained either TCP or QUIC. We believe that, possibly due to connection racing, some of the QUIC streams were discarded because they contained a significant number of TCP packets. Shreedhar *et al.* [12] measured the QoE of YouTube videos using a headless Chrome browser. They found that QUIC experiences fewer stalls compared to TCP/TLS when run over a network with 10% packet losses. Furthermore, they observed that QUIC experiences higher-quality switches compared to TCP in a lossy network. Kakhki *et al.* [13] conducted an in-depth analysis of QUIC and compared the performance of QUIC with TCP. They observed that QUIC experiences fewer stalls, i.e., QUIC outperforms TCP only at higher video resolution. However, at low to medium resolution, no significant improvement in QoE was observed compared to TCP. A few other works like [18, 19] proposed an improvement of QoE of video streaming by extending the QUIC protocol. Palmer *et al.* [19] suggested supporting unreliable streams in QUIC. They proposed an extended version of QUIC called *Clip-Stream* which offers reliability only on selected frames, i.e., I-Frames. They developed a prototype and observed that the modified QUIC provides better QoE than original QUIC. Zheng *et al.* [18] design a multipath scheduling algorithm named *XLINK* for video services by using QUIC. They extracted the video QoE and utilized it for multipath scheduling and management. They conducted a large-scale study over e-commerce short videos and found XLINK is useful for video applications. XLINK can reduce the start-up delay and re-buffering rate of shorter videos.

Table 2.2 compares prior studies on video streaming over QUIC with the scope and contributions of this thesis. Several key observations emerge from the comparison. First, although several works perform root-cause analysis of QUIC's performance, most consider only high bandwidth settings (typically 5–100 Mbps) and short video durations

(60–300 seconds). These settings do not capture the behavior of long-running streaming sessions or the performance degradation that arises under constrained bandwidth. In contrast, our work focuses on low-bandwidth environments (64 Kbps–1 Mbps) and long-duration videos (2700–4500 seconds), enabling a deeper understanding of QUIC's performance over sustained sessions. Second, nearly all prior studies restrict their analysis to Chrome/Chromium and do not examine cross-browser differences. Our dataset and analysis encompass both Chrome/Chromium and Firefox, highlighting browser-specific implementation gaps that have been largely overlooked in existing research. Third, our thesis explicitly studies the interaction between browser QUIC implementations, connection racing, and video streaming, thereby offering insights not covered in earlier studies. Finally, previous studies typically evaluate synthetic workloads, short clips, or limited applications. In contrast, this thesis focuses on realistic video streaming workloads, using YouTube long sessions and varying network conditions to analyze the practical implications of QUIC deployment.

Overall, this comparison highlights a clear research gap: the lack of a comprehensive, browser-centric study of QUIC under low-bandwidth, long-duration streaming scenarios. This thesis fills that gap by providing an in-depth analysis and corresponding solutions for improving QoE in such environments. Next, we examine prior work on the impact of middleboxes on application performance, as middlebox behavior is closely intertwined with the browser's connection racing mechanism. Understanding this relationship is crucial for analyzing how middleboxes affect protocol selection and, consequently, application QoE.

2.4 Middleboxes and its impact on application QoE

So far, we have discussed prior works analyzing QUIC's performance in video streaming applications. However, we have not covered studies that examine the impact of middleboxes on application QoE. We therefore review this line of work next.

Langley *et al.* [9] discussed *fallback* behavior of QUIC. They highlighted how the 1-bit change in the public field of QUIC resulted in some middleboxes allowing a few initial packets of QUIC but then subsequently blocking it and hence they face a destiny of blackhole. This is worse than middleboxes completely blocking QUIC; hence, several modern browsers consider fallback as a solution for this problem. The authors realized the pain points of sustainability of a protocol across various middleboxes through "*deployment*

TABLE 2.2: Comparison of prior work on video streaming over QUIC.

Work	**Root Cause Analysis**	**Network Bandwidth**	**Network RTT**	**Browsers**	**Explored QUIC Impl.**	**Video Duration**	**Application**
Langley et al.(2017) [9]	✓	–	✓	Chrome/ Chromium	✓	–	Video streaming and web browsing
Kakhki et al.(2017) [13]	✓	5–100 Mbps	✓	Chrome	✓	60 s	Web browsing and limited video streaming
Palmer et al.(2018) [19]	×	20 Mbps	×	Chrome	×	296 s	Video streaming
Seufert et al.(2019) [11]	×	1 Mbps	×	Chrome	×	180 s	YouTube video streaming
Shreedhar et al.(2021) [12]	✓	20–100 Mbps	✓	Chrome	×	180 s	Video streaming and web browsing
Zheng et al.(2021) [18]	✓	20–30 Mbps	✓	–	–	120 s	Video streaming
This Thesis	✓	64 Kbps – 1 Mbps	✓	Chrome/ Chromium and Firefox	✓	2700 s – 4500 s	Video Streaming

impossibility cycle". Hao Li *et al.* [7] highlight that updating the middleboxes for supporting emerging transport protocols is still difficult. For example, when QUIC or UDP is blocked, it impacts the QoE of users. They propose a Python-based domain-specific language called *Rubik* for programming the network stack of a middlebox. Several works from the literature have discussed the troubles of using a new protocol that legacy middleboxes cannot support. Li *et al.* [7] highlighted that QUIC is still not supported by a few middleboxes; as a result, it impacts the QoE of the application. They further propose a language for programming the middleboxes to support emerging protocols. The seminal paper from Google [9] also discusses the pain points of middlebox report that the issues arose when they modified they had changed the 1 bit field in the header of QUIC packets. This supported some middleboxes to allow the initial packets of QUIC but later blocked

them all, which created a destiny of black holes for the packets. However, several other works [37, 38, 39, 40, 41] in the literature have also highlighted the issues of QUIC due to the interference with the Internet middleboxes.

A similar fallback discussion was in the literature for IPv4 to IPv6 migration where the Happy Eyeball [42] algorithm is used to decide to fallback from IPv6 to IPv4. In this case, a connection is attempted using IPv6 first and then with IPv4. If IPV6 is successful, it is used for further communication, else IPv4 is used. A client caches this information to avoid *thrashing* the network with further attempts. Such caching information is flushed after a timeout interval.

Although the prior work has looked at the impact of middleboxes on applications, none of the papers considered whether application performance can be impacted unnecessarily, even without a middlebox.

2.5 QUIC under High-Speed Networks

In this section, we review prior work examining how application performance is affected when using QUIC as the transport protocol in high-speed networks.

Zhang et al. [16] show that QUIC underperforms in high-bandwidth settings with bitrate reductions of up to 9.8%. They found that QUIC's performance degrades due to significant receiver-side processing overhead. This overhead arises from excessive data packet handling at Layer 3 and above, as well as QUIC's reliance on user-space acknowledgments. Tanya et al. [12] highlight TCP and QUIC performance trade-offs based on file size and system resource usage. It was observed that while QUIC achieves higher throughput than TCP for smaller file sizes ($\leq$ 20 MB), TCP outperforms QUIC for larger file transfers ($>$ 20 MB up to 2 GB). As for large files, the overheads of QUIC outweigh the benefits of 0-RTT. Additionally, performance profiling during a 200 MB download from Google Drive revealed that QUIC consumes nearly twice the CPU resources compared to TCP to achieve similar throughput. Kempf et al. [43] enable reproducible QUIC benchmarks on dedicated hardware and high-speed links. Their evaluation of 10G networks reveals significant performance variation across QUIC client-server implementations. They identify packet I/O as the dominant bottleneck limiting QUIC throughput. Moreover, the study reveals that CPU architecture and core allocation significantly influence QUIC performance. Bi et al. [22] also highlight that in a high-bandwidth network, QUIC introduces considerable

computational overhead that shifts the system bottleneck from the network to the CPU. As a result, when CPU resources are constrained, HTTP/3 (using QUIC as transport protocol), despite being the successor to HTTP/1.1 (using TCP as transport protocol), can exhibit lower throughput than its predecessor. This performance degradation primarily arises from the CPU-intensive nature of key QUIC operations, including ACK processing, packet transmission, and data encryption. These tasks place a significant demand on the processor, making CPU efficiency a critical factor in achieving optimal performance with QUIC-based communication, particularly in resource-limited environments.

These prior studies clearly highlight the limitations of QUIC in high-speed networks. §. 2.3 highlights QUIC's advantages in poor or lossy networks. Thus, no single protocol consistently delivers the best performance across all conditions. In the next section, we review prior work proposing adaptive transport protocol switching frameworks.

2.6 Adaptive Transport protocol Switching

Zhang et al. [44] proposed *WiseTrans*, an adaptive transport protocol selection mechanism designed to enhance the performance of mobile web services. WiseTrans leverages machine learning techniques and historical network data to handle spatial heterogeneity and make protocol-switching decisions at the request level. In an extension of their work, Zhang et al. [45] introduced real-time experimental evaluation, allowing the system to compare the performance of transport protocols before switching. However, a major limitation of WiseTrans is its static nature; the machine learning model is trained on a fixed dataset and remains unchanged over time. This makes the system less effective in adapting to evolving network conditions and user behavior. Additionally, the training data is based on discrete network categories such as 2G, 3G, 4G, and WiFi, whereas real-world environments often involve more nuanced and fluctuating conditions, potentially limiting the model's generalizability. In a similar vein, Zhou et al. [46] developed FlexHTTP, an intelligent and scalable system for selecting the optimal HTTP version based on network conditions and web page structure. Unlike WiseTrans, FlexHTTP incorporates both global and local information to continuously update its machine learning classifier. Nevertheless, the system lacks extensive empirical validation, as it does not provide sufficient experimental data or statistical analysis to support its claims. In contrast to machine learning-based approaches, Ganji et al. [47] introduced Dynamic Transport Selection, a probabilistic framework that adaptively chooses the appropriate transport protocol for the

current network environment. While this approach avoids the limitations of static models, it still lacks integration with application-specific performance metrics. A key limitation shared by all these frameworks is their focus on general web service optimization rather than video streaming performance. None of the existing solutions considers the unique characteristics and challenges of adaptive video streaming, such as buffer dynamics, segment quality variations, and playback stability. Therefore, there is a clear need for an adaptive transport protocol selection mechanism tailored specifically for video streaming applications, one that can learn from evolving network conditions and dynamically update its decision-making strategy over time.

Table 2.3 summarizes prior adaptive transport protocol selection frameworks and highlights several important limitations that motivate the need for a new solution for video streaming. First, existing frameworks such as WiseTrans, FlexHTTP, and Dynamic Transport Selection are primarily designed for web services, where transport decisions impact page load time, rather than long-lived, bandwidth-intensive video sessions. These approaches overlook the unique challenges of video streaming, such as continuous bitrate adaptation, buffer dynamics, and rebuffering durations, which require fundamentally different decision strategies. Second, prior works rely heavily on static machine learning models or probabilistic rules that do not adapt in real time to evolving network conditions, thereby limiting their effectiveness in highly variable environments. Third, most frameworks evaluate their techniques under simplified or narrow network scenarios, typically focusing on deterministic bandwidth profiles or fixed RTT/loss settings, which do not reflect the complex and bursty patterns encountered in real networks. Due to these limitations, existing transport protocol selection mechanisms cannot be directly applied to video streaming and fail to address the challenges posed by modern browser-based QUIC deployments. Therefore, in this thesis we propose IntelSwitch and it fills this gap by designing a reinforcement learning–based adaptive framework specifically for video streaming, leveraging fine-grained QoE and network observations to make dynamic, context-aware transport protocol decisions. Its design accounts for both application-level behavior and browser-level interactions, enabling more robust and effective adaptation in practical streaming environments.

Overall, the existing literature highlights significant gaps in understanding QUIC's real-world behavior. Moreover, none of these studies capture the fine-grained, application-level and network-level interactions required to fully understand QUIC's performance in modern browser-based video streaming. To bridge this gap, the next chapter introduces

our H3B measurement tool and the comprehensive dataset we collected to support the analyses conducted throughout this thesis.

TABLE 2.3: Comparison of adaptive transport protocol selection frameworks

Framework	**Application Domain**	**Technique**	**Live Experiment**	**Network Scenario**
WiseTrans [16, 44]	Web services	Machine learning model for transport selection	Yes	2G, 3G, 4G, WiFi
FlexHTTP [46]	Web services	Machine learning-based dynamic protocol selection	No	1–25 Mbps bandwidth, 30–500 ms RTT, 0.01–1% loss rate
Dynamic Transport Selection [47]	Web services	Probabilistic transport selection framework	No	LTE and WiFi
IntelSwitch	Video Streaming	Reinforcement learning based framework	No	3% packet loss+50ms delay, LTE and 5G

Chapter 3

H3B Tool and Dataset

'Measure what is measurable, and make measurable what is not."
— Galileo Galilei

In this chapter, we describe the tool we developed to conduct a large-scale measurement campaign for video streaming across diverse geographical locations and bandwidth patterns. Using this tool, we collect a large-scale dataset containing network-application information from YouTube streaming sessions over two popular browsers, Chrome/Chromium and Firefox, that support both HTTP/3 and the legacy HTTP/2.

3.1 Introduction

HTTP/3 is the latest variant of the popular *Hypertext Transfer Protocol* (HTTP), which has recently been widely adopted by major Internet giants such as Google, Facebook, Cloudflare, Akamai, Apple, and many others. Unlike its predecessors, HTTP/3 uses QUIC [48, 49] as the underlying transport protocol. QUIC is expected to provide reduced latency and better application QoE (Quality of Experience) compared to TCP, the widely used transport protocol for the Internet, for the past three decades. The QUIC developers [9] and several other works [8, 13, 50, 51, 52] have shown the superiority of QUIC compared to TCP in terms of supporting better QoE with less stall for video streaming. YouTube is one of the most popular streaming media services on the Internet today, which supports QUIC. Despite this growing adoption, understanding QUIC's

real-world behavior remains challenging due to the complex interactions between network conditions, browser implementations, and application-level adaptation logic. Therefore, this chapter of the thesis presents an HTTP/3-supported browser dataset collection tool named H3B for YouTube video streaming. The tool accepts a video identifier and a predefined network bandwidth pattern as input, and produces two categories of output logs: (i) application-layer logs capturing QoE-related metrics, and (ii) network-layer logs detailing packet exchanges annotated with the corresponding transport protocol.

3.2 Contribution

In the above context, this chapter describes the following contributions of our work.

1. We collect synchronized application-layer and network-layer logs under diverse bandwidth patterns, including high, low, and very low bandwidth, as well as mobility scenarios, over both WiFi and cellular networks. We have collected data across five geographically distinct locations (Delhi, Bangalore, New York, Germany, and Singapore), enabling a comprehensive analysis of how network dynamics and geographic factors jointly impact YouTube video streaming performance and QoE.

2. We collect streaming data for 46 YouTube videos spanning news, entertainment, and educational content over both legacy HTTP/2 (TCP) and modern HTTP/3 (IETF QUIC), enabling a direct comparison of protocol behavior as well as an investigation into how video characteristics influence adaptive streaming behavior and overall QoE in real browser-based deployments.

3. The dataset was collected over an 18-month period for 5474 hours, allowing for the analysis of temporal variations in QoE and the assessment of the impact of browser updates and evolving QUIC implementations over time. The dataset is publicly available at `https://github.com/NKShukla/H3B`.

3.3 H3B Tool

In this section, we present the design of our tool H3B.

Modern browsers such as Chrome/Chromium and Firefox support video streaming over both legacy TCP-based HTTP/2 and QUIC-based HTTP/3 protocols. During a YouTube streaming session, the YouTube video player continuously requests audio and video segments from YouTube servers, while the browser internally manages transport protocol selection, connection establishment, HTTP request-response handling, and communication with the video player. As a result, several important transport- and browser-level behaviors remain hidden from standard application telemetry. Understanding how these behaviors influence video streaming QoE, we require a browser-centric measurement framework capable of jointly analyzing application- and network-level behavior. Motivated by this need, H3B is designed to enable the synchronized collection of browser-level application logs and packet-level network traces during YouTube streaming sessions. Figure 3.1 illustrates the overall architecture of H3B. The tool takes video ID and network bandwidth patterns as input and provides application-layer logs on application QoE and network-layer logs of packet exchanges as output. Details are as follows:

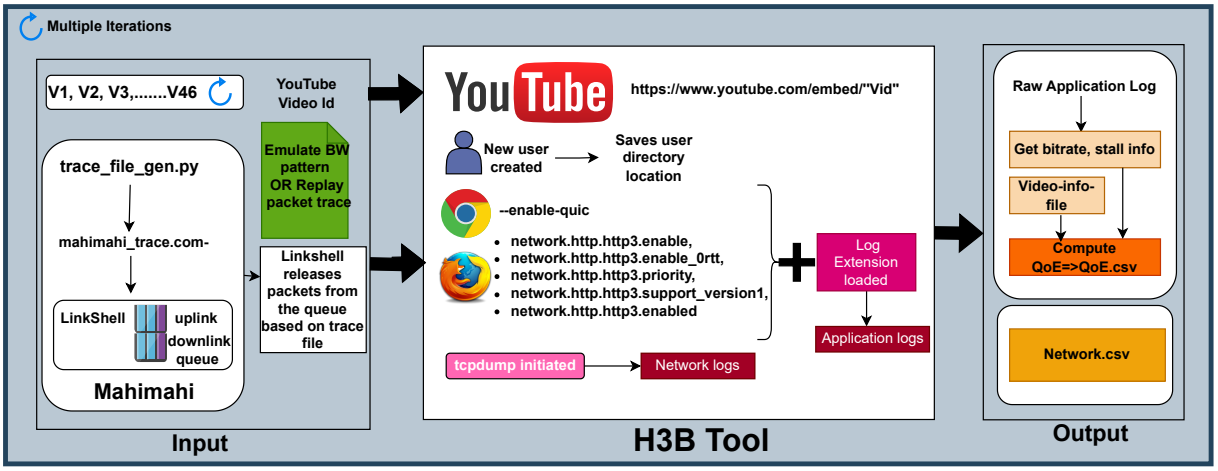


FIGURE 3.1: H3B architecture

3.3.1 Input to H3B

YouTube video selection: We first create a list of 46 YouTube videos, each lasting 45 minutes to 1 hour, as shown in Table 3.1. The genres of videos are News, Entertainment, Education, Indian talk shows, Comedy, Stanford online lectures, and British TV series. The minimum and maximum video quality of all the videos were 144p and 1080p. We use the YouTube developer's API to fetch necessary information about a particular video using

its unique identifier. We made HTTP GET request [1] with the video's unique identifier. It is important to note that this endpoint is no longer available, maybe due to changes in YouTube's policies. We obtain a mapping between itag, bitrate, and the video quality corresponding to a particular video using the YouTube developers API. Table 3.2 shows the mapping of itag to the corresponding bitrate and the quality label of a video. Multiple quality labels include 144p, 240p, 360p, 480p, 720p, and 1080p. YouTube supports both *constant bitrate* (CBR) and *variable bitrate* (VBR) encoding; thus, a same quality label can have multiple bitrates and hence multiple *itag*s. For example, Table 3.2 shows three different bitrates for each quality label and corresponding *itag*s.

TABLE 3.1: Details of the selected videos

Number of Videos	*46*
Video duration	*45 minutes - 1 hour*
Types of Videos	*News, Entertainment and Education videos*
Minimum Video Quality	*144p*
Maximum Video Quality	*1080p*

TABLE 3.2: Video-Info Table of two sample videos

Video ID —>	**-SI0HKTfHN4**																	
Itag	137	22	135	134	133	160	18	136	242	136	398	244	397	243	396	278	394	
Bitrate(Kbps)	4466585	743210	930845	654628	301679	121826	576066	2029085	244471	2029085	1497188	849429	737405	492803	399554	112110	94436	
Quality	1080p	720p	480p	360p	240p	144p	360p	720p	240p	720p	720p	480p	480p	360p	360p	144p	144p	
Video ID —>	**4QcHHal-pt4**																	
Itag	248	247	244	243	242	278	18	22	137	136	135	397	134	396	133	395	278	
Bitrate(Kbps)	2680531	1503593	756822	412207	224867	98699	512759	649740	4466452	1804135	1019629	706715	603781	387958	303545	185179	98699	
Quality	1080p	720p	480p	360p	240p	144p	360p	720p	1080p	720p	480p	480p	360p	360p	240p	240p	144p	

Network emulation: The purpose of such an H3B tool is to replay a given network behavior and stream YouTube videos on that network. While it is not possible to replay bandwidth patterns over an "in-the-wild" setup, a large number of recent studies [18, 53, 54, 55, 56, 57] have relied on benchmark network emulator frameworks like *Mahimahi* [58] to analyze the protocol performance in a realistic setup. Accordingly, to emulate a specific network bandwidth over the test setup, we use *Mahimahi* Link Shell (`mm-link`) emulation, which emulates network links using user-specified packet delivery trace files. Mahimahi maintains two queues: one for the uplink traffic and the second for the downlink traffic. Whenever packets arrive from the Mahimahi's `mm-link` or the Internet, they are placed directly into one of two packet queues, depending on whether they are uplink or downlink packets. Then it releases the packets based on the input packet-delivery trace file, we represent the trace file as *mahimahi_trace.com−*. So, each line in a packet-delivery trace file represents the time at which the packet of size MTU can be delivered. Also, `mm-link`

[1] $https://www.youtube.com/get\_video\_info?video\_id=\%s\&el=embedded\&ps=default\&eurl=\&gl=US\&hl=en\&html5=1\&c=TVHTML5\&cver=6.20180913"\%-SI0HKTfHN4$

wraps to the beginning of the input packet-delivery trace file on reaching its end. We write a Python script to generate a packet-delivery trace file that supports the corresponding network bandwidth. In addition to emulating bandwidth patterns, we support emulating any natural network packet trace collected using a packet capture tool, such as *Wireshark* or `tcpdump`. We create two types of network setups: (1) *semi-controlled setup*, where we replay a total of 161 packet traces. As part of our data collection, we recruited a few volunteers. We asked them to watch YouTube videos of their choice while they are mobile (more details in §. 4.3). We collected 56 traces from our volunteers. We replayed both our volunteer-collected traces and from a publicly available dataset [59]. This dataset was collected while streaming video over YouTube through WiFi on a laptop. The collected dataset contains data from both stationary and mobile user scenarios. From this, we used 105 mobility traces. We converted the packet traces to packet-delivery trace files using the mechanism used in [53]. (2) *controlled setup*: where we emulate various bandwidth patterns. We begin with a static bandwidth pattern to determine the minimum and maximum bandwidth at which the minimum and maximum video quality can be achieved. At < 128 Kbps, the lowest video quality $144p$ was observed throughout the video; similarly, at > 1500 Kbps, the quality level of $720p$ was observed. Each of the patterns initiates a loop from an initial bandwidth, makes a jump to the next bandwidth level, maintains them for a duration, and this goes up to the final bandwidth as mentioned in Table 3.3, and then the patterns repeat with the final bandwidth as the initial bandwidth, the initial bandwidth as the final bandwidth, and a negative value of the jump. Finally, from the individual bandwidth pattern, we generate emulated traces as follows, Say, at time T_i, a packet has been transmitted; then the transmission time for the next packet T_{i+1} is computed from the bandwidth at that time and the packet size (which equals to path MTU). For example, if the bandwidth is 640000 bps, then $T_{i+1} = T_i + ((1500 * 8)/640000)$. To ensure that the end-to-end Internet bandwidth follows this emulated bandwidth pattern and the backbone bandwidth does not impact the YouTube performance, we performed a sanity check using parallel streaming without applying any traffic shaping. Such a setup consistently achieves the maximum quality level, even when run on a different machine. Further, we observe that more than 80% of the total data was transferred from the CDN with IP address *180.149.59.12* for both QUIC-enabled and QUIC disabled browser setups. This IP belongs to NKN (National Knowledge Network) of the Delhi location that maintains a local cache of Google. This way, we ensured that no other parameters except the network play a role in the performance measurement.

TABLE 3.3: Bandwidth Patterns used for Network Emulation

Emulated Bandwidth Patterns	**Initial**	**Final**	**Jump**	**Jump Dur.**
Dynamic High (DH)	1152Kbps	896Kbps	-256	240 sec
Dynamic Low (DL)	640Kbps	128Kbps	-256	240 sec
Dynamic Very Low (DVL)	64Kbps	256Kbps	+64	60 sec
Semi-controlled (Real)	"in-the-wild" data and mobility traces from [59]			

3.3.2 H3B tool

Upon receiving an input, H3B creates a new user profile and configures the user data directory where application logs are stored. It also provides control over the transport protocol by allowing QUIC to be enabled or disabled during video streaming. These configurations are used to initialize and manage the subsequent streaming session. In Chrome/Chromium, the `--enable-quic` flag is employed, while in Firefox, a set of preferences including `network.http.http3.enable`, `network.http.http3.enable_0rtt`, `network.http.http3.priority`,`network.http.http3.support_version1`, and `network.http.http3.enabled` are adjusted accordingly. Notably, when we disable QUIC, the browser setup uses the legacy HTTP/2 instead of HTTP/3. In order to embed the YouTube video inside the browser, we created our I-Frame and appended the YouTube video id at the end of "`https://www.youtube.com/embed`", which is called an I-Frame source URL [60]. The autoplay feature was also set inside the I-frame to play the video automatically once the player was loaded. We collect logs at two layers: the application layer and the network layer. For the application layer, we download all the video-related information from the requests made by YouTube client, and the responses received. Since the connections were encrypted, we needed to log these requests/responses within the browser; we created our own application log extension. Inside the application log extension, we have used the `console.log` API for the logs. We defined four events when we collected the logs – `BEFORE_REQUEST` (the YouTube client prepares an HTTP Request), `SEND_HEADERS` (YouTube client sends the HTTP Request to the YouTube server, which contains various DASH parameters), `RESPONSE_STARTED` (YouTube client starts receiving the response from the YouTube server) and `COMPLETE` (YouTube client completes receiving the response, including the video segment data, from the server). These events log all the HTTP Request messages that have been sent from the client to the server. For this events log, we observe two different types of requests – a `videoplayback` request (request URL is like `https://r6---sn-o3o-qxal.googlevideo.com/videoplayback`) which is a HTTP

GET Request, and a `qoe` request (Request URL is like `https://www.youtube.com/api/stats/qoe?...`) which is a HTTP POST Request. The `videoplayback` request contains the details of the requested video segment, such as *the request timestamp* of the requested segment, *total bytes* in the segment, byte *range* of the segment data to be downloaded, *itag* value (tells the requested video and audio quality), *rbuf* (receiver buffer) in second, *rbuf* in bytes, *clen* (the maximum possible length of the downloaded segment), *dur* (duration of the downloaded segment), the *download speed*, and about the protocol it has used (either QUIC or TCP). Once the YouTube server receives this request, it sends the corresponding HTTP Response and the segment data as requested. On the contrary, the YouTube client periodically sends the `qoe` request to the server, on triggering of the event `streamingstat` that is triggered after a predefined number of frames are rendered over the client. This is a POST request that embeds the statistics about the video streaming, i.e., what amount of data has been rendered or played over the YouTube client. The `qoe` request embeds the following information: *request timestamp*, *itag* of the segment played, *health of buffer* that tells at real-time t for how much duration the video has been buffered, and *cmt* that tells at real-time t for how much duration the video has been played. We combine these two statistics to obtain the details of a video frame, i.e., when and at what quality a frame has been downloaded, and when the frame has been played finally. It can be noted that YouTube *itag* value encodes the bitrate in terms of quality labels, such as $144p$, $240p$, $360p$, $480p$, $720p$, etc. However, YouTube stores a mapping among *itag*, quality labels, and the corresponding bitrate in terms of a `video-info` file, which can be obtained through YouTube developers API. Indeed, for our analysis, we select the videos that contain such mapping in their `video-info` file. Interestingly, this covers both the *constant bitrate* (CBR) and *variable bitrate* (VBR) encoded videos, as the quality label to bitrate mapping can directly be inferred from a given *itag* value. Loading this log extension was easier in Chromium, but in the case of Firefox, it was a difficult task. Therefore, for Firefox, we use the command-line tool web-ext to run this extension with the `-verbose` flag, which prints the logs to the terminal.

For the network layer logs, we use the packet capture tool `tcpdump` to collect the network packet captures (pcap). Upon completion of video streaming, the application and network logs are stored in the local file directory, and the `tcpdump` process is terminated along with the user profile.

3.3.3 Design Rationale and Contributions of H3B

Existing tools and frameworks like Wrapper app[20] and YoMo app[21] collect application- and network-level logs for YouTube streaming applications, but they are mainly designed for application-level QoE analysis in mobile streaming environments and provide limited visibility into browser-level transport behavior. H3B is specifically designed for browser-centric analysis of modern QUIC-enabled browsers supporting both TCP and QUIC. Building H3B required addressing several practical challenges. One of the key challenges was the heterogeneity across modern browsers. Different browsers expose different mechanisms for enabling or disabling QUIC, configuring HTTP/3 behavior, collecting browser logs, and supporting browser extensions. As a result, instrumentation techniques and experimental configurations that worked for one browser did not necessarily generalize to others. H3B, therefore, required browser-specific exploration and configuration to ensure consistent measurements across widely used browsers such as Chrome/Chromium and Firefox. Another important challenge was collecting detailed application-level logs beyond the limited information exposed through YouTube's "Stats for Nerds." To address this, we designed a custom browser extension tailored for YouTube streaming applications to capture fine-grained browser-level events synchronized with packet-level network traces. This enables detailed analysis of browser-controlled mechanisms such as connection racing, protocol fallback between QUIC and TCP, and transport protocol switching behavior. In addition, we automated the entire experimentation pipeline to systematically collect streaming sessions over both TCP and QUIC under identical network conditions. To ensure independence across experiments and eliminate caching effects, H3B creates a fresh browser profile for every streaming session. These capabilities collectively distinguish H3B from existing measurement frameworks and make it suitable for systematic analysis of browser-level transport protocol behavior and its impact on video streaming QoE in modern HTTP/3-enabled browsers.

3.3.4 Output of H3B

H3B generates an application log and the corresponding network log at its output.

Application log structure: The application logs provide two types of information. (1) *Video-related information while streaming a YouTube video*: The timestamp of the requested segment, total bytes of the segment, the itag value (tells the audio and video

quality), the duration of the requested segment, and the protocol (TCP or QUIC) it uses for the segment request. (2) *Statistics about video streaming*: Information like the amount of video data that has been rendered and has been played, the quality of the segment stored in the buffer, buffer health (tell at any time t how much amount of video is buffered in the buffer), and the playback duration in terms of how much video has been played. The sample application log description is shown below, and the details of important parameters are in Table 3.4.


```
"52": {
    "type": "videoplayback",
    "request_ts": 6.583864990234375,
    "complete_ts": 13.9705791015625,
    "total_time": 7.386714111328125,
    "total_bytes": 174508,
    "complete_range": [
        0,
        174508
    ],
    "complete_itag": 397,
    "complete_rbuf_sec": 0.0,
    "complete_rbuf": 0,
    "complete_rn": 1,
    "complete_clen": 150413396,
    "complete_dur": 2624.64,
    "kbytes/second": 23070.876465714333
},
"58": {
    "type": "streamingstats",
    "request_ts": 10.3623898925781,
    "itag": 397,
    "buffer_health": "10.027:0.00",
    "cmt": "10.027:0.000",
    "bwe": "10.027:130000",
    "vps": "10.027:B"
},
```


QoE metric parameters extracted from application log used for Evaluation: We convert the raw application log into a JSON file by only extracting the required

TABLE 3.4: Application log description

Application log parameter	Description
request_ts	the request timestamp of the requested segment
complete_ts	the complete timestamp of the requested segment
total_time	the total timestamp from segment request to segment request gets complete
total_bytes	the total bytes of the segment
range	bytes of the segment data to be downloaded
itag	the requested video segment quality
rbuf	the receiver buffer in seconds
clen	the maximum possible length of the requested segment
dur	duration of the downloaded segment
buffer_health	at real-time t for how much duration the video has been buffered
cmt	at real-time t how much duration of the video has been played

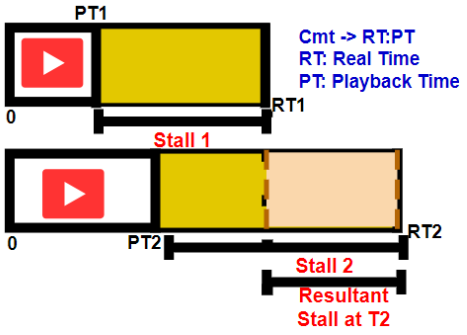


FIGURE 3.2: The figure shows how the stalls are computed from the *cmt* parameter of streaming stats we obtained in application log

features. We characterize the QoE via three performance counters: the average bitrate of a video (Bitrate), the average bitrate variation (V_BRate), and the average re-buffering or stalling (Stall) that the user experiences during a video playback. We compute these metrics using the *streaming-stat* information. These metrics are captured multiple times during a YouTube video session. Note that the HTTP response contains the information corresponding to a requested segment. To compute bitrate, we extract the *itag* value inside the streaming stat and then map it to the video information file. We calculate the bitrate variation by determining the difference between the previous and current bitrates. To compute stall, we utilize the *cmt* parameter of the streaming statistic. We subtract the playback timestamp from the real timestamp, as shown in Fig. 3.2, which yields the cumulative stall at any given time. We then subtract the stall computed for the previous instance of the streaming stat from the current instance to obtain the temporal distribution of the stall. We then compute the moving average of all three parameters each time the new streaming stat log appears in the application log.

Network log structure:

The network logs are packet captures (pcap) obtained using tcpdump. Packet captures contain detailed information about a packet such as timestamp, source, and destination IP address, port number, application protocol (HTTP), transport protocol (TCP or QUIC or UDP), fields specific to TCP (SYN, FIN, RST, etc.), packet length, sequence number, acknowledgment number (for TCP, for QUIC one has to decrypt the packet header), packet type (TCP data/ACK, QUIC handshake/initial/payload, etc.), whether retransmitted, etc. Since we are interested in correlating the number of bytes transferred over two transport protocols, TCP and QUIC, with application-layer QoE, we extract five fields and convert them into a csv format. The structure of the network.csv is shown below:

```
Timestamp,  Source IP,  Destination IP, protocol, length
03:04:42, 192.168.29.48, 142.251.12.188, QUIC,     559
```

3.4 Dataset Description

We launched a measurement campaign using H3B over 5 different geographical locations. The locations are: Delhi, Bangalore, New York, Germany, and Singapore. Such a measurement was conducted using two different web browsers Chrome/Chromium and Firefox. We stream once over QUIC-enabled and once over QUIC-disabled. One of the testbeds is set up inside our campus premises, and the rest are set up using Digital Ocean machines. We emulate 5 different bandwidth patterns, namely Dynamic High (DH): a good bandwidth, Dynamic Low (DL): a poor bandwidth, and Dynamic Very Low (DVL): a very poor bandwidth. Real: real packet captures under mobility over WiFi and over cellular network.

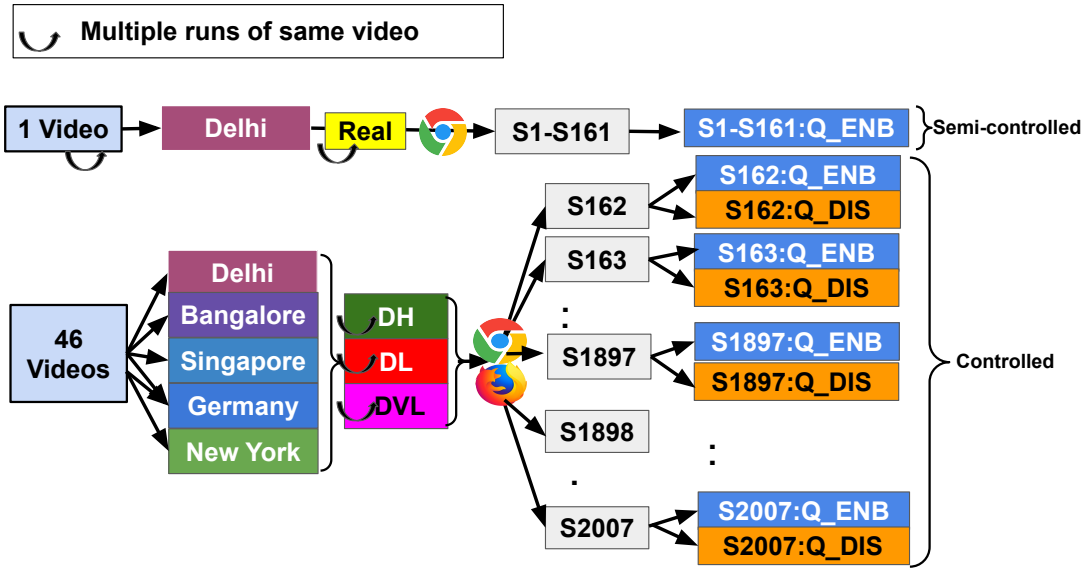
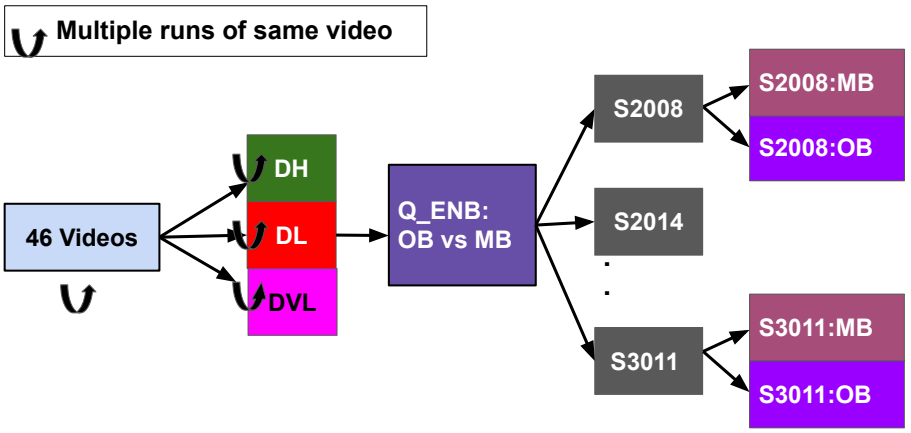
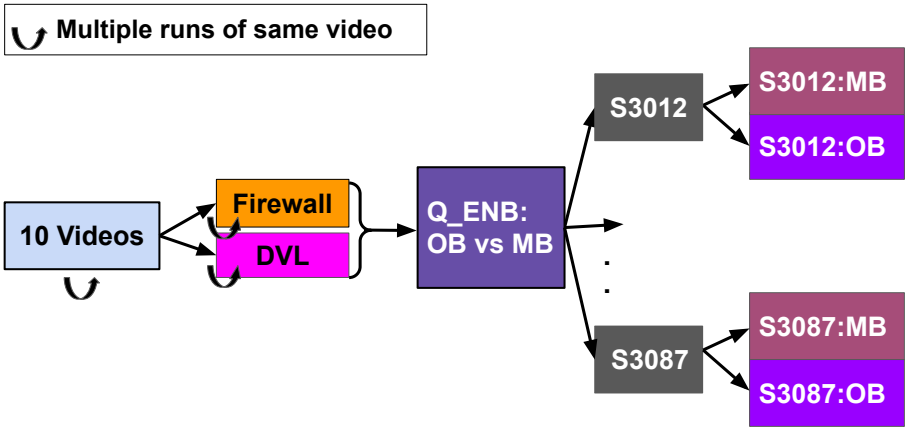


FIGURE 3.3: Dataset Organization across 5 locations, 2 browsers, and 4 bandwidth patterns

Fig. 3.3 represents our dataset. For a semi-controlled setup, we streamed a single video multiple times over a QUIC-enabled (Q_ENB) Chrome/Chromium browser setup. We obtain a dataset of a total of 161 YouTube streaming sessions, named as *S1, S2,....., and S161*. For a controlled setup, we run 46 videos of different genres over Chrome/Chromium and Firefox multiple times under each bandwidth pattern. Finally, we obtain datasets of total 1846 steaming session pairs that we name as *S162, S163, ..., S2007* once over QUIC-enabled (Q_ENB) browser and then over QUIC-disabled (Q_DIS) browser (summarized in Table 3.5). Fig. 3.4 (a) shows the dataset of 1004 steaming session pairs of static modification in the Chromium browser, named as $S2008$, $S2009$,..., $S3011$. Further, Fig. 3.4 (b) shows the dataset of 76 video sessions of the poor network and firewall named as $S3012$, $S3013$, ..., $S3087$. Our dataset is available at `https://github.com/NKShukla/H3B`(Access:July 2026).



(a)



(b)

FIGURE 3.4: Dataset organization of YouTube over (a) QUIC-enabled *statically* modified Chromium browser (MB) and original Chromium browser (OB), (b) QUIC-enabled *Dynamic-Racing* Chromium browser (MB) and original Chromium browser (OB)

In the following chapters, we conduct an in-depth analysis of the collected dataset to understand how QUIC behaves across browsers, network conditions, and geographical regions. We conduct an extensive comparison between streaming sessions with QUIC enabled and those with disabled, examining key QoE metrics, including average bitrate, bitrate variation, stall duration, and protocol switching frequency.

Table 3.5: Dataset details

Total sessions	6013
Total duration	5474 hours
Q_DIS duration	1877 hours
Q_ENB duration	3770 hours
DH duration	278 hours
DL duration	1712 hours
DVL duration	1650 hours
Firefox duration	142 hours
Semi-controlled duration	161 hours
Delhi duration	3922 hours
Bangalore duration	490 hours
Singapore duration	253 hours
Germany duration	459 hours
New York duration	218 hours
Static solution	1453 hours
Dynamic-Racing	78 hours

3.5 Summary of the Chapter

In this chapter of the thesis, we presented H3B, a browser-based toolbox for collecting fine-grained application-layer and network-layer logs for YouTube video streaming over HTTP/3-enabled browsers. A key feature of H3B is its ability to emulate arbitrary network bandwidth patterns in a controlled manner, enabling systematic and repeatable experimentation. Using H3B, we conducted a large-scale measurement campaign across five geographically diverse locations and five distinct bandwidth patterns. This effort resulted in a comprehensive dataset comprising 5474 hours of streaming data across 6013 YouTube sessions, capturing both application-level QoE metrics and detailed network-level behavior.

The resulting dataset provides a valuable foundation for developing intelligent models that integrate network and application behavior in modern HTTP/3 browsers while accounting for backward compatibility with legacy HTTP/2. More broadly, this dataset enables several important research directions, including dynamic tuning of protocol and application parameters based on network conditions, intelligent network–application co-design for improving QoE, and learning-driven prediction and optimization of YouTube streaming performance. In particular, the diversity of network conditions, geographical

locations, and streaming scenarios captured in our dataset makes it well-suited for training and evaluating data-driven models that aim to improve video streaming QoE under challenging and realistic Internet conditions.

Hence, building on the dataset and measurement infrastructure introduced in this chapter, the next chapter leverages this large-scale, in-the-wild data to conduct a detailed analysis of browser behavior during HTTP/3 video streaming.

Chapter 4

Analysis of Streaming Media Performance over Popular QUIC-enabled Browsers

'The important thing is not to stop questioning."
— Albert Einstein

Modern browsers such as Chrome/Chromium and Firefox implement fallback mechanisms to preserve connectivity when QUIC experiences connection establishment failures or persistent degradation. As part of this behavior, browsers may switch traffic from QUIC to TCP through a mechanism commonly referred to as connection racing discussed in Chapter 2. While such mechanisms are primarily designed to handle scenarios where UDP traffic is blocked or rate-limited by middleboxes, their behavior and impact under poor or variable network conditions remain largely unexplored. In this chapter, we investigate how browser-level connection racing influences video streaming QoE in QUIC-enabled browsers. We first conduct a small in-the-wild pilot study to determine whether protocol switching due to connection racing is common in practical YouTube video streaming. Motivated by the observations, we then utilize the semi-controlled and controlled dataset collected using H3B tool in Chapter 3 to systematically analyze the frequency, causes, and QoE impact of connection racing under diverse network conditions. Building on these insights, we further propose browser-side mechanisms for adaptively managing connection racing behavior in order to improve streaming performance.

4.1 Introduction

In recent years, major Internet service providers, including Google, Facebook, Akamai, Cloudflare, and Apple, have increasingly migrated their services to HTTP/3. While this transition promises performance improvements, browser-level adoption of HTTP/3 has faced a key deployment challenge. HTTP/3 is built on QUIC, which operates over UDP, and many legacy middleboxes still block or rate-limit UDP traffic due to long-standing security policies [5, 6, 7]. To preserve backward compatibility in such environments, modern browsers such as Google Chrome and Mozilla Firefox employ a mechanism known as connection racing, in which both TCP and QUIC connections are initiated in parallel and the protocol that completes successfully is selected to serve the application.

Although connection racing is designed to handle failures caused by middleboxes, it can also lead to frequent protocol switching and unintended performance side effects during long-lived sessions such as video streaming. In this chapter, we investigate how and why protocol switching occurs in practice, analyze the conditions that trigger connection racing, and evaluate its impact on application-level QoE. Through detailed measurements and browser-level analysis, we aim to understand whether protocol switching is always necessary and how browser implementation choices influence the performance of HTTP/3-enabled applications.

4.2 Contribution

This chapter of the thesis has the follwoing contributions:

1. We analyzed how the use of an HTTP/3-enabled browser impacts the performance of YouTube compared to the legacy HTTP/2 browsers. The analysis is done over 6013 video sessions combining more than 5474 streaming hours (228 days) over two popular browsers, including (i) emulation over different bandwidth patterns, (ii) in-the-wild collected data, and (iii) over real network traces. We observe that enabling QUIC inside the browser does not consistently improve the QoE; instead, YouTube suffers over QUIC-enabled in the browser when network quality fluctuates or the bandwidth is low.

2. We explore the code flow of the Chromium (the open-source implementation of Google Chrome) and Firefox browsers' implementation of the connection racing mechanism. We validate our observation that streaming applications suffer in a poor network due to the browser's implementation of connection racing. We made necessary modifications to the Chromium browser to forcibly stop connection racing if a protocol is affected by a middlebox or a poor network. We observe that the *Modified Browser* (when the data is transmitted over a pure QUIC stream) outperforms the *Original Browser* (with the connection racing option) for more than 60% cases.

3. Next, we implement a dynamic solution for deciding when to enable connection racing. We made this implementation and data open-sourced[1]. We evaluate our solution in poor network conditions and the presence of UDP rate-limiting firewalls. We observe that it correctly enables connection racing in the case of a firewall and disables it in the case of a poor network. Our solution improves the QoE compared to the original browser.

4.3 Motivation: Frequency of Protocol Switching In-the-Wild during QUIC-TCP Connection Racing

We start by performing a pilot measurement "*in-the-wild*" in low and middle-income countries to analyze how frequent protocol switching happens and whether we should care about it while designing a streaming media application. For this measurement, we recruited 10 volunteers from 8 different cities over four different countries, namely Delhi, Bangalore, Kanpur, Howrah, and Aligarh from India, Nairobi from Kenya, Riyadh of South Arabia, and Lahore. We asked the volunteers to watch any video of their choice using the YouTube application (with QUIC-enabled) on their phones while walking or traveling. We instructed the volunteers to use their mobile data for such streaming in two different modes. First, in the incognito mode inside the YouTube app, ensure the video is not played from the cache. Second, without incognito mode. We asked them to collect a packet trace while streaming the videos for around $10-15$ minutes using a packet capture application named `PcapDroid`. To ensure that packets from no other mobile applications are captured except YouTube, we use YouTube as the target app option in `PcapDroid`. We collected a total of 56 such packet traces to analyze whether the presence of TCP bytes

[1] `https://github.com/sapna2504/Chromium-Modification-V2` (Access: July 2026)

is common across various locations when QUIC is already enabled inside the YouTube app.

We quantify connection racing ($\mathcal{R}Qvs.T_v$ i.e., Racing between QUIC and TCP) for a Q_ENB streaming session (v) as a ratio between bytes transferred only over TCP ($\mathbb{T}_v$) to the total bytes transferred over both QUIC ($\mathbb{Q}_v$) and TCP ($\mathbb{T}_v$). Notably, we consider only the bytes of packets that contain application data and ignore control packets.

$\mathcal{R}Qvs.T_v = \frac{\mathbb{T}_v}{(\mathbb{T}_v+\mathbb{Q}_v)}$.

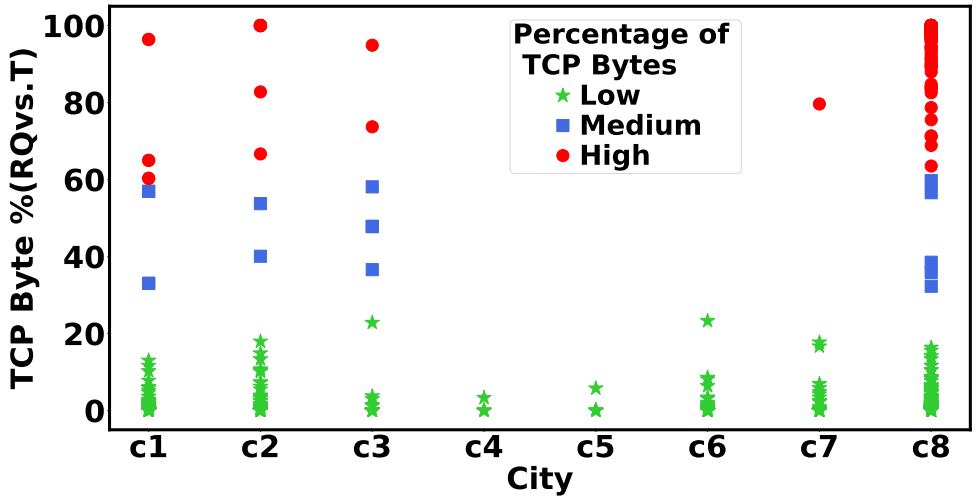


FIGURE 4.1: Percentage of TCP bytes ($\mathcal{R}Qvs.T_v$) due to connection Racing over different cities where C1: Delhi, C2: Bangalore, C3: Kanpur, C4: Howrah and C5: Aligarh are of India, C6: Nairobi of Kenya, C7: Riyadh of Saudi Arabia and C8: Lahore of Pakistan. Percentage of TCP bytes ($\mathcal{R}Qvs.T_v$) : Low ($0-30\%$), medium ($30-60\%$), and high ($60-100\%$)

Fig. 4.1 shows the percentage of TCP bytes due to connection racing in different cities across the four countries. All the logs from different cities are collected while traveling from home to the workplace or vice versa. In the network logs, we found successful QUIC connection establishments. Hence, we assume that no middleboxes were blocking UDP. As a result, such presence of TCP bytes refers to the impact of connection racing. Therefore, we conclude that unnecessary connection racing is common in scenarios with poor or varying network bandwidth, as is typically experienced during mobility. The volunteers reported poor application performance, characterized by low video quality and frequent rebuffering, while watching YouTube videos. Besides poor network conditions impacting ABR (Adaptive Bitrate) decisions, we wondered whether unnecessary connection racing also contributes to degradation in users' QoE (Quality of Experience). While in-the-wild measurements show that connection racing frequently occurs in practical mobile YouTube video streaming scenarios, they are not sufficient to systematically analyze its impact

on application performance. Therefore, to perform a detailed and controlled analysis of connection racing behavior. We therefore utilize the synchronized application- and network-level dataset collected using the H3B tool presented in Chapter 3.

Using the controlled dataset collected through the H3B tool, we first analyze the impact of different network configurations and browser behaviors on application QoE under QUIC-enabled (Q_ENB) and QUIC-disabled (Q_DIS) streaming sessions. We then investigate the relationship between $\mathcal{R}Qvs.T_v$, and the additional degradation in streaming performance.

4.4 Application Performance versus Connection Racing: How This Affects Streaming QoE

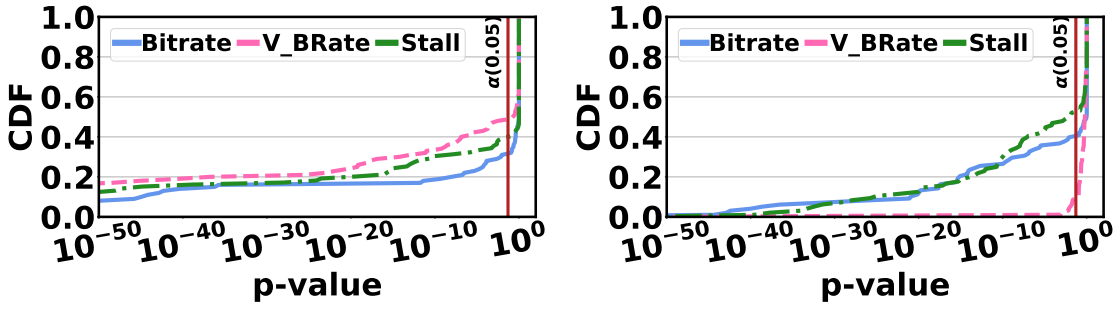
In this section, using the controlled dataset collected through the H3B tool, we analyze the impact of different network configurations and browser behaviors on application QoE under QUIC-enabled and QUIC-disabled streaming sessions. We then investigate whether increased connection racing, quantified using $\mathcal{R}Qvs.T_v$, is associated with additional degradation in streaming performance.

4.4.1 Impact of Connection Racing on Streaming QoE

Using the controlled dataset collected with the H3B tool, we compare the YouTube performance of two browser setups: QUIC-enabled (Q_ENB) and QUIC-disabled (Q_DIS) for Chrome/Chromium and Mozilla Firefox. We compute the three metrics using the *streaming-stat* information discussed in Chapter 3. We compute the average by taking all the samples from the beginning to the current instance of the streaming stat. Consequently, we obtain a moving average of these three metrics. This way, we obtain the instantaneous moving average value for each QoE metric for each video streaming session. Note that averaging the three metrics to compute a single value for the entire session is not a suitable approach, as it fails to capture instantaneous changes. For statistical analysis, we perform hypothesis testing over all the 1846 streaming session pairs ($S162 - S2007$) to find out in what percentage of sessions YouTube over Q_ENB browsers perform (a) better than, (b) statistically similar, and (c) worse than the Q_DIS browsers. Hypothesis testing [61] is a statistical method used to determine whether an observed difference between two sets of measurements is statistically significant or whether it could have occurred due to

random variation in the data. The null hypothesis assumes that there is no statistically significant difference between the QoE observed over Q_ENB and Q_DIS browser setups. In other words, any observed QoE variation is assumed to arise due to normal variability in streaming behavior under identical network conditions. To test this hypothesis, we perform a Welch's t-test [62] for each streaming-session pair. Welch's t-test is chosen because it does not assume equal variances between the two samples. The test computes a p-value, which represents the probability of observing a difference at least as extreme as the measured one under the assumption that the null hypothesis is true. We use a significance level of $\alpha = 0.05$. If the p-value exceeds 0.05, we fail to reject the null hypothesis and consider the two browser setups to perform similarly. Otherwise, the null hypothesis is rejected. Since we do not assume a priori that either browser setup will consistently outperform the other, we use a two-tailed test [63]. When the null hypothesis is rejected, the Q_ENB browser performs significantly better or worse than the Q_DIS browser for the corresponding QoE metric.

Streaming Quality vs Browser Setup: For Chrome/Chromium, the null hypothesis



(a) Chrome/Chromium (b) Mozilla Firefox

FIGURE 4.2: Hypothesis test result for various QoE metrics

was accepted for 15%, 30%, and 16% cases, respectively, in terms of average bitrate, average bitrate variation, and average stall. For Firefox, the null hypothesis was accepted in 23%, 52%, and 28% of the cases. For the cases where the null hypothesis was rejected, we performed the two-tailed test. Fig. 4.2(a) and (b) show the CDF of p-values observed over the two-tailed test for the cases when YouTube over the Q_ENB browser did not perform better than the Q_DIS one for Chrome/Chromium and Firefox, respectively. We observe that YouTube over Q_ENB browsers suffers for about 32% (for Chrome) & 41% (for Firefox) cases in terms of average playback bitrate and 50% (for Chrome) & 9% (for Firefox) for average bitrate variation, compared to the disabled one. Furthermore, Q_ENB ones experience higher stalls compared to Q_DIS ones for 40% (Chrome) and 52%

(Firefox) of the instances. This raises the question of why YouTube's performance over Q_ENB browsers suffers in terms of all the QoE metrics.

Impact of Bandwidth Patterns: We perform hypothesis tests across data collected over different bandwidths for each QoE performance metric. Fig.4.3 (a), (b), and (c) show the result for the dynamic-high (DH), dynamic-low (DL), and dynamic-very-low (DVL) bandwidth patterns. For DH, the null hypothesis was accepted for 45%, 12%, and 84% cases, for DL, 22%, 24%, and 12% cases, and for DVL, 3%, 41%, and 8% cases for average bitrate, average bitrate variation, and average stall, respectively. Furthermore, for the cases where the null hypothesis was rejected for DH, we observe that YouTube performs poorer than Q_DIS browsers in terms of average bitrate, average bitrate variation, and stall rates for 19%, 17%, and 11% cases, respectively, combined across both browsers. For DL, the corresponding percentages are 41%, 32%, and 53%, respectively. For DVL, the corresponding percentages are 50%, 29%, and 48%, respectively. In DVL, most videos were played at the minimum quality, i.e., at 144p; hence, the instances of quality drops are fewer.

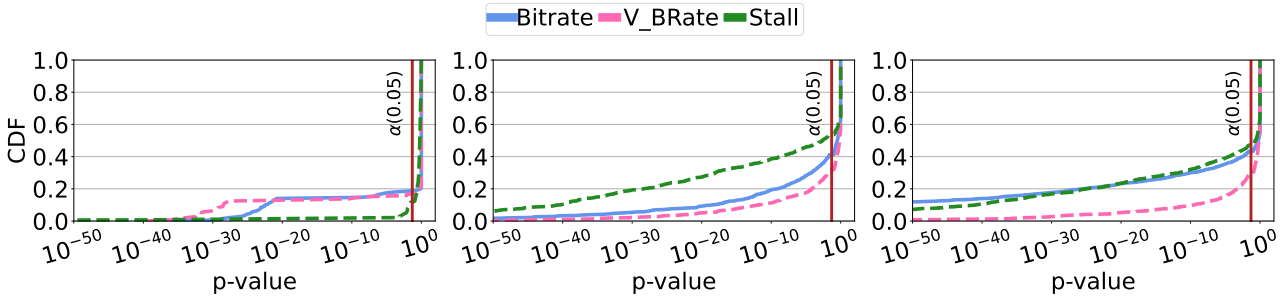


FIGURE 4.3: Hypothesis test results for different bandwidth patterns: (a) DH, (b) DL, (c) DVL

Impact of Geographical Locations: To answer this, we test the hypothesis using data collected in five different locations: Delhi, Bangalore, New York, Germany, and Singapore. We observe that the null hypothesis was accepted for 15%, 12%, 20%, 19% and 13% respectively for average bitrate, for 26%, 31%, 27%, 34% and 40% respectively for average bitrate variation and for 10%, 18%, 27%, 16% and 20% respectively for average stall. For the cases where the null hypothesis was rejected, we performed a two-tailed test. Fig. 4.4 (a) shows the CDF plot of the hypothesis-testing result for the average bitrate for the cases where YouTube over the Q_ENB browser did not perform well. We observe for Delhi, Bangalore, New York, Germany, and Singapore, the Q_ENB browser performs worse than the disabled one in about 49%, 36%, 37%, 37%, and 46% cases, respectively. In terms of average bitrate variation (shown in Fig. 4.4 (b)), the similar percentages are

26%, 28%, 30%, 32% and 24% cases, respectively. In terms of stall (shown in Fig. 4.4 (c)) the percentages are 49%, 43%, 43%, 46% and 43% cases, respectively. Hence, our observations remain consistent across various locations as well.

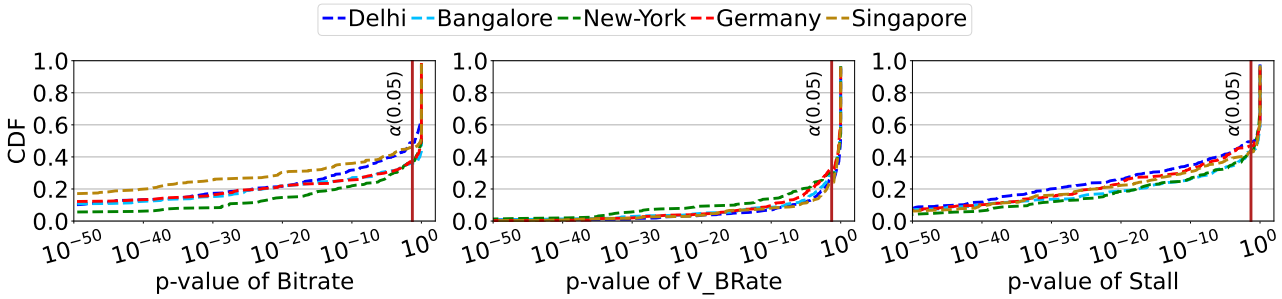


FIGURE 4.4: Hypothesis test results for different geographical locations

We also analyzed whether our observations remain consistent across video genres and the entire data collection duration of 14 months, from September 2021 to December 2022. We find that our observations of YouTube's performance suffer more from Q_ENB browsers than Q_DIS browsers, and this remains consistent across these two verticals as well. While these observations indicate that Q_ENB browsers can degrade streaming performance under poor and variable network conditions, they do not directly explain the underlying cause of this degradation. Therefore, we next investigate whether increased connection racing in a Q_ENB browser, quantified using $\mathcal{R}Qvs.T_v$, is associated with degradation in application QoE.

4.4.2 Application QoE vs Connection Racing

To study the temporal pattern of the video's QoE with $\mathcal{R}Qvs.T_v$, i.e., percentage of TCP bytes due to connection racing, we need to bring the application and network logs within a common time frame. The streaming stats provide information in an asynchronous manner. Therefore, we define two derived metrics from the application logs: quality drop duration (B) and stall duration (S). We compute all these metrics for a sample window size of 30 (seconds) throughout the video session. To compute the quality drop (B) duration, we first encode different video quality levels using a mapping as follows: 144p $\rightarrow$ 1, 240p $\rightarrow$ 2, 360p $\rightarrow$ 3, 480p $\rightarrow$ 4, and 720p $\rightarrow$ 5. The quality drop duration $B = (e(q_t^{prev}) - e(q_t)) \times t_{dur}(q))$ where $e(q_t^{prev})$ refers to the previous quality level, $e(q_t)$ refers to the current quality level obtained from the streaming stats, and $t_{dur}(q)$ is the

duration for which the quality remained dropped in a 30-sec window. The stall duration S is computed as the sum of all stalls experienced within the 30-sec time window.

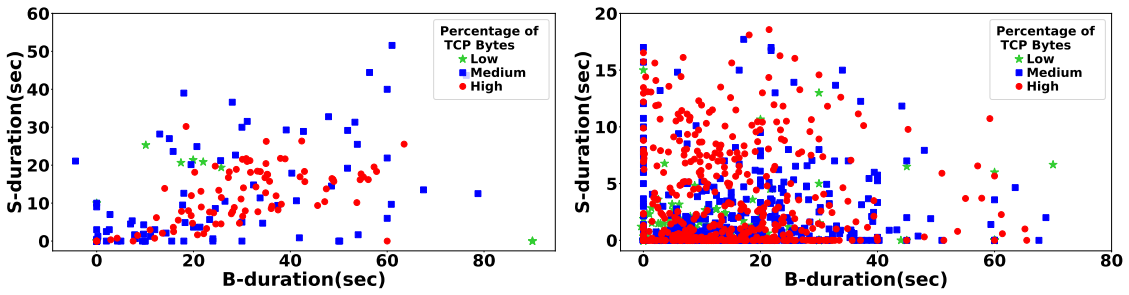


FIGURE 4.5: Scatter plot of Quality drop (B) duration and Stall duration (S) (seconds) at different levels of percentage of TCP bytes due to connection racing (low ($< 30\%$), medium ($30 - 60\%$) and high ($> 60\%$)) in (a) semi-controlled (b) controlled setup

We categorize the percentage of TCP bytes due to connection racing into three levels – low ($0 - 30\%$), medium ($30 - 60\%$), and high ($60 - 100\%$). Fig. 4.5(a), (b) show a scatterplot of quality drop (B) duration and stall (S) duration for three different levels of TCP bytes percentage present due to connection racing in a semi-controlled and controlled setup, respectively. Under both setups, we observe that as the presence of TCP bytes increases from low to high, the quality drop and stalling duration also increase. For example, in the semi-controlled setup, the median quality drop duration increases from 11 seconds to 25 seconds, while the median stall duration increases from 1 second to 8 seconds as the percentage of TCP bytes increases from low to high levels. Similarly, in the controlled setup, the median quality drop duration increases from 6 seconds to 13 seconds, and the stall duration increases from 0.3 seconds to 1.3 seconds. If the videos were streamed at the lowest quality (as is done in DVL), the stalling instances are observed more frequently. Otherwise, both quality drop and stalling are experienced when the network is poor(DL)/varying (Real). Such an impact on video QoE in terms of stalling and quality drop can be an effect of connection racing. In case of a poor network, the videos are streamed over the lowest possible quality in both Q_ENB and Q_DIS browser setups, but it might be possible that connection racing adds extra stalling in the case of Q_ENB browser. These observations indicate a correlation between increased connection racing and degradation in streaming QoE. However, since poor and variable network conditions can themselves negatively affect adaptive bitrate decisions and application performance, Fig. 4.5 alone does not establish causality between connection racing and QoE degradation. Instead, these observations motivate a deeper controlled investigation into

whether browser-level connection racing contributes additionally to application performance degradation beyond the impact of adverse network conditions alone. Therefore, in the subsequent sections, we perform controlled experiments to isolate and analyze the specific impact of connection racing behavior on video streaming QoE.

Takeaway 1: *We compared the application QoE obtained over a QUIC-enabled browser with that of a QUIC-disabled browser. We observe that application QoE suffers more from the former than the latter. Further, these findings are statistically consistent across browsers, bandwidth patterns, locations, time, and video genres. We then observe that higher connection switching occurrences between QUIC and TCP are associated with poorer application QoE.*

4.5 Correlation between Application QoE and Connection Racing Instances

From the above section, we found that connection racing and QoE parameters such as average bitrate, average bitrate variation, and average stall are correlated. To quantify this correlation analytically through mathematical foundations, we perform statistical correlation analysis on the QoE parameters and the captured connection switching occurrences from the Q_ENB browser streaming sessions. From the hypothesis testing, as reported in §. 4.4 (Fig. 4.2), we consider the cases where YouTube over Q_DIS browser performs better than the Q_ENB video playback session and analyze whether connection racing was one of the primary causes behind the poor performance of YouTube over Q_DIS. Modeling correlation is particularly challenging in this case because we have to work with three different time axes – the *real-time* (t) (the time of the global clock when a video frame is rendered), the *playback time* ($\tau(f_t)$) (the relative time of a particular video frame with respect to the video start time) and the *download time* ($\sigma(f_t)$) (the time when a video frame has been downloaded).

Let t be the global clock time. Let f_t be the video frame that has been rendered over the YouTube client at time t. We assume that $\tau(f_t)$ is the playback time for the frame f_t, i.e., the time when frame f_t is rendered/played, and $\sigma(f_t)$ is the download time for the frame f_t, i.e., the time required to download frame f_t. It can be noted that $\tau(f_t) \geq t$ as the frame f_t can be rendered either at its original playback time $\tau(f_t)$ (if there is no rebuffering or video stall) or after that (if there is a stall before rendering the frame).

Also, $\sigma(f_t) < t$, as f_t needs to be downloaded before it is rendered. Further, there would be a rebuffering if $\sigma(f_t) > \tau(f_t)$, i.e., the video frame f_t is downloaded after its original playback time.

Let $\mathcal{B}_v(t)$ and $\mathcal{S}_v(t)$ denote the perceived bitrate and video rebuffering at time t during a video playback session v. Here t is the real time (in ms) of the system. Let $e(q)$ be the encoded quality level at which the video is being played; for a video quality level q, we encode it using the mapping $q \rightarrow e(q)$ as discussed in §. 4.4.2 (144p $\rightarrow$ 1, 240p $\rightarrow$ 2, 360p $\rightarrow$ 3, 480p $\rightarrow$ 4, and 720p $\rightarrow$ 5). Let q_t be the quality level at which the video frame f_t is being played, q_t^{prev} be the quality level at which the video was being played just before it switched to the quality level q_t (so, there is a quality switch $q_t^{prev} \rightarrow q_t$), and $t_{dur}(q_t)$ be the duration of playback after it switched to q_t from q_t^{prev}. Then, we model $\mathcal{B}_v(t)$ as the time series, $\mathcal{B}_v(t) = (e(q_t^{prev}) - e(q_t)) \times t_{dur}(q_t)$. Similarly, we model $\mathcal{S}_v(t) = t - \tau(f_t)$, following Fig. 3.2 (in §. 3.3.4) of Chapter 3.

Following this, we compute the cross-correlation between the two time-series data $\mathcal{RQ}vs.T_v(t)$ and $\mathcal{B}_v(t)$ to study the correlation between connection racing and QoE drop events.

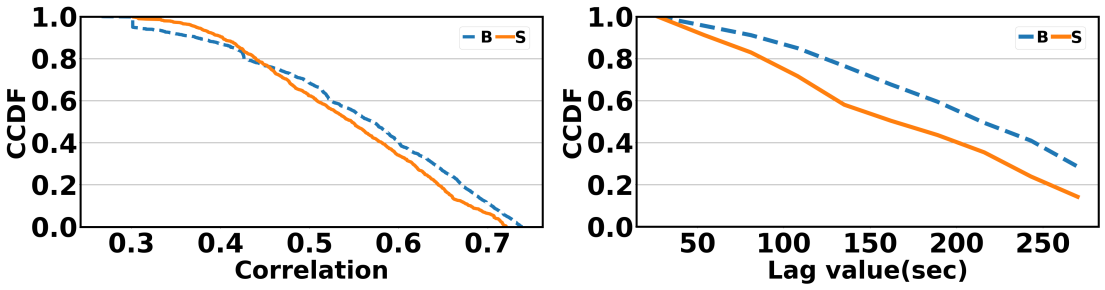
$$(\mathcal{RQ}vs.T_v * \mathcal{B}_v)(\mu_b) = \int_{-\infty}^{\infty} \overline{\mathcal{RQ}vs.T_v(t)} \mathcal{B}_v(t + \mu_b) dt \tag{4.1}$$

where $\overline{\mathcal{RQ}vs.T_v(t)}$ is the complex conjugate of $\mathcal{RQ}vs.T_v(t)$ and μ_b is the corresponding lag. We compute the μ_b for which we obtain the maximum cross-correlation (μ_b^{max}). Similarly, we compute the cross-correlation between $\mathcal{RQ}vs.T_v(t)$ and $\mathcal{S}_v(t)$ as follows.

$$(\mathcal{RQ}vs.T_v * \mathcal{S}_v)(\mu_s) = \int_{-\infty}^{\infty} \overline{\mathcal{RQ}vs.T_v(t)} \mathcal{S}_v(t + \mu_s) dt \tag{4.2}$$

where μ_s is the corresponding lag. Similar to the above, we compute the μ_s for which we obtain the maximum cross-correlation (μ_s^{max}). We next plot the CCDF distributions of correlation and the lag values in Fig. 4.6.

Fig. 4.6(a) shows the correlation between connection racing ($\mathcal{RQ}vs.T_v(t)$) and quality drop ($\mathcal{B}_v(t)$) as well as stalling ($\mathcal{S}_v(t)$). Similarly, Fig. 4.6(b) shows the lag values μ_b^{max} and μ_s^{max}. We observe that about 50% of videos positively correlate with a 0.55 or higher correlation coefficient for both cases. Further, we observe from Fig. 4.6(b) that for both $\mathcal{B}_v(t)$ and $\mathcal{S}_v(t)$, the lag values across different streaming sessions are distributed within



(a) (b)

FIGURE 4.6: (a) Cross-correlation between $\mathcal{RQ}$ vs.T and $\mathcal{B}$ & $\mathcal{RQ}$ vs. T and $\mathcal{S}$ (b) Lag value for quality drop and stalling

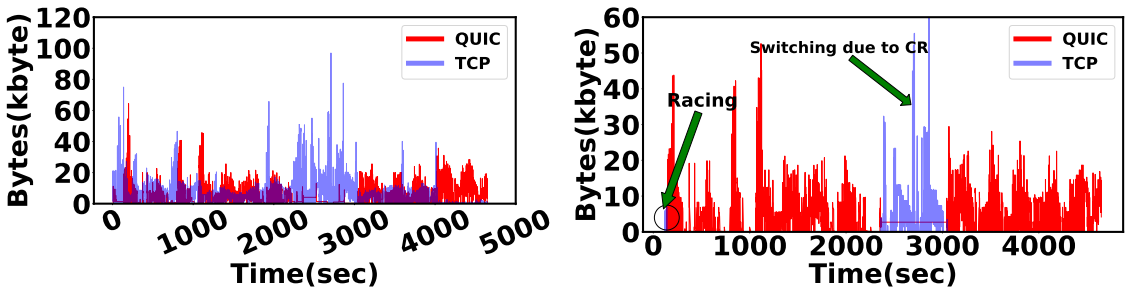
250sec. Hence, we conclude that connection racing and quality drop events are correlated, and there is a lag or time difference between connection racing events and quality drop events. We next investigate why connection racing affects video streaming QoE.

4.6 Delving Into the Depth: Why Connection Racing Affects Streaming QoE

In this section of the thesis, we conduct a root-cause analysis to understand why connection racing is triggered even in the absence of middleboxes that block or rate-limit UDP traffic. Although connection racing is primarily designed as a fallback mechanism for handling QUIC connectivity failures caused by middleboxes, our observations reveal that browsers initiate racing under additional conditions as well.

4.6.1 Temporal Analysis of Connection Racing Events and Corresponding YouTube QoE

Next, we present a few representative streaming sessions to illustrate the browser-level connection-racing behavior observed in our collected data. We then perform a temporal analysis to examine the impact on YouTube QoE when connection racing occurs. Fig. 4.7(a) shows the bytes transferred over both TCP and QUIC across all the server IPs from which a Q_ENB browser received the streaming data (multiple YouTube backend servers can process client requests).Fig. 4.7(b) shows one sample server from which the client received the maximum amount of data. Note that the presence of a small number of



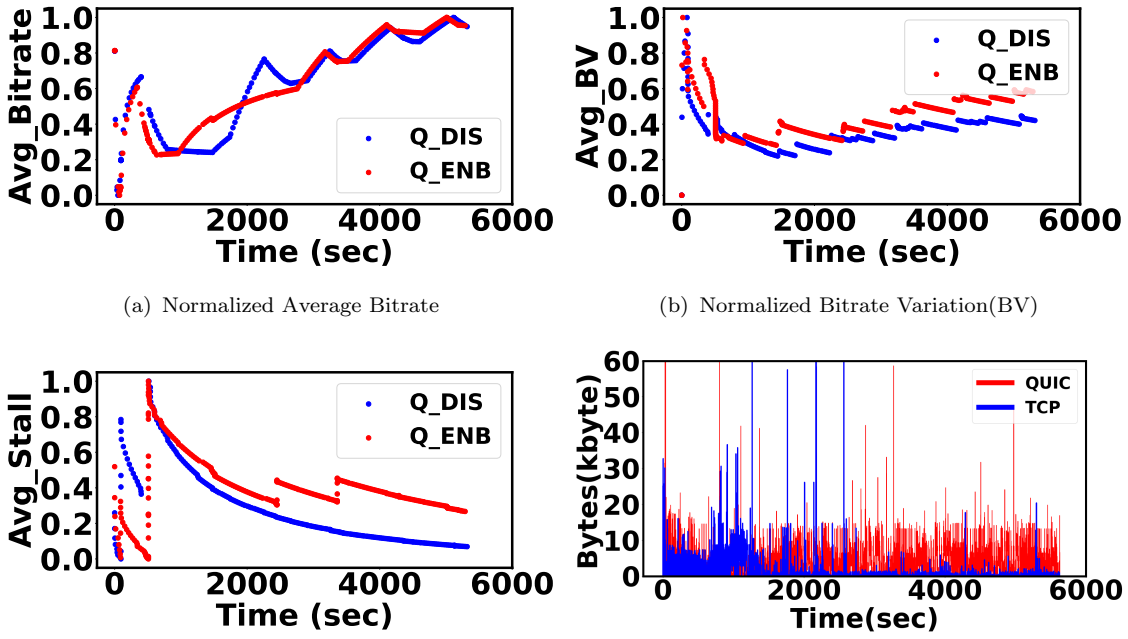
(a) (b)

FIGURE 4.7: Connection Racing observed over Chrome Browser in terms of bytes transferred over individual protocols, for a YouTube streaming over Q_ENB browser, (a) Across all IPs from where the client received streaming data, (b) For a single IP on which 89% of total bytes were transferred.

TCP packets during the initial phase is expected due to the browser's connection racing mechanism. If QUIC successfully wins the race, subsequent requests are expected to continue primarily over QUIC, resulting in only a limited presence of TCP traffic. However, the figure shows a significant amount of TCP traffic even after successful QUIC connection establishment, indicating repeated protocol switching events between QUIC and TCP.

Specifically, the session initially starts with TCP, then switches to QUIC after QUIC wins the race, and later reverts back to TCP before switching again to QUIC. Fig. 4.7(b) shows that the first two HTTP requests are transmitted over TCP, after which subsequent requests are transferred over QUIC. Later, around 2200 seconds, an error occurs either in an existing QUIC stream or during the creation of a new stream. Consequently, instead of continuing over the already established QUIC connection, the browser starts transferring data over TCP while waiting for QUIC to recover. The key observation here is that the fallback to TCP occurs even though the network conditions indicate a poor or variable network, rather than explicit UDP blocking or QUIC rate-limiting. While fallback to TCP is beneficial when QUIC traffic is genuinely blocked or rate-limited by middleboxes, we argue that in this scenario, the browser misinterprets temporary QUIC degradation caused by poor network conditions as a signal requiring protocol fallback. Since QUIC is itself designed to operate efficiently under lossy and variable networks, such unnecessary protocol switching introduces additional instability and negatively impacts application QoE.

Further we analyze the instantaneous change in bitrate, variation in bitrate, and stall with respect to the extent of connection racing. We plot the perceived bitrate, variation in bitrate, and stall values over time for two representative pairs of video streaming sessions



(a) Normalized Average Bitrate

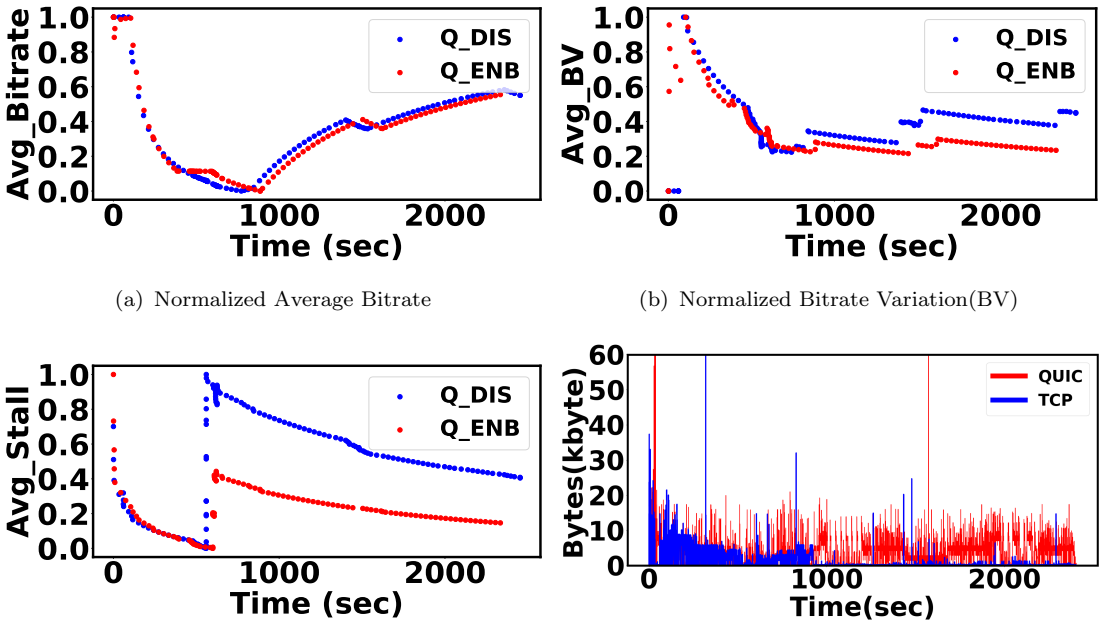
(b) Normalized Bitrate Variation(BV)

(c) Normalized Stall Duration

(d) Connection Racing over Q_ENB Browser

FIGURE 4.8: Temporal performance for Scenario S1: YouTube performs better over a Q_DIS browser

of sample video streaming sessions across two different cases: (S1) YouTube performs better over a Q_DIS browser, and (S2) YouTube performs better over a Q_ENB browser. For S1, as shown in Fig. 4.8(a), the normalized bitrate for the YouTube streaming over a Q_DIS browser is better (with $p < 0.05$) than that over the Q_ENB one under similar traffic shaping. Similarly, for normalized bitrate variation (Fig. 4.8(b)) and stalling (Fig. 4.8(c)), YouTube performs better over the Q_DIS browser. In the second case (S2), we observe that YouTube performs better over the Q_ENB browser (with $p < 0.05$) for all the three QoE parameters – average bitrate (Fig. 4.9(a)), bitrate variation (Fig. 4.9(b)) and stall duration (Fig. 4.9(c)). To explore the reason behind this, we plot the temporal distributions of the QUIC and TCP bytes over the Q_ENB browser for the above two scenarios. Notably, Q_DIS browsers carry only the TCP bytes. Interestingly, we observe that TCP traffic is less in S2 (Fig. 4.9(d)) than in S1 (Fig. 4.8(d)), indicating less amount of protocol switching occurrences for S2 compared to S1. We also observe fewer fluctuations in the QoE performance counters in S2 compared to S1 when using the Q_ENB browser. This temporal analysis indicates that excessive connection switching between QUIC and TCP during their racing, particularly due to poor network bandwidth, introduces network instability, affecting the streaming QoE.



(a) Normalized Average Bitrate

(b) Normalized Bitrate Variation(BV)

(c) Normalized Stall Duration

(d) Connection Racing over Q_ENB Browser

FIGURE 4.9: Temporal performance for Scenario S2: YouTube performs better over a Q_ENB browser

This temporal analysis shows that frequent switching between QUIC and TCP during connection racing, especially under poor network bandwidth, can introduce instability and degrade streaming QoE. Next, we analyze the temporal patterns of connection racing, along with key network parameters, to understand how these conditions drive instability within the browser itself.

4.6.2 Temporal Analysis of Connection Racing Events and Network Parameters

Next, we analyze the temporal patterns of connection racing together with key network parameters to understand how poor and variable network conditions influence browser protocol behavior. Since RTT and throughput are commonly used by browsers to estimate network quality, we naturally expect poor network conditions to lead to increased RTT values, temporary QUIC degradation, and occasional QUIC-related errors. However, our primary goal is not to show that such degradation is unexpected, but rather to investigate how the browser reacts to these transient conditions. In particular, we study whether temporary RTT spikes and QUIC-related errors cause the browser to interpret

the QUIC path as blocked or rate-limited, thereby triggering repeated connection racing and fallback to TCP. Since QUIC is designed to operate under lossy and variable network conditions, we expect QUIC communication to continue despite temporary RTT inflation. Therefore, the key question is whether the browser unnecessarily initiates protocol fallback even when QUIC connectivity still exists. The network parameters, such as RTT and throughput, can thus be utilized to understand the browser's decision-making behavior for triggering connection racing. We found that the browser monitors RTT in two ways: (1) by using NQE (Network Quality Estimator) service within Chromium that provides network quality estimation [64, 65] and (2) by monitoring RTT values used by the congestion control algorithm. Since throughput computations are performed over a time window, throughput values can provide stale estimates in cases of varying network bandwidth (which we are emulating). Hence, we use RTT. We try to correlate application errors with network parameters. We log all the QUIC-related errors while streaming a video over the Q_ENB browser under a poor bandwidth pattern (DVL). We extract the RTT values used by the QUIC congestion control algorithm. We instrument the browser to get these values.

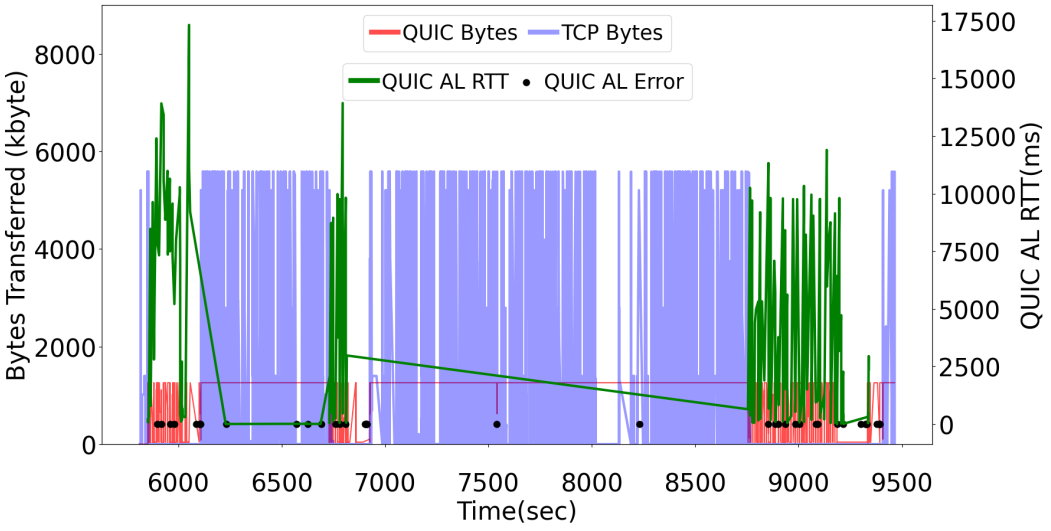


FIGURE 4.10: Chromium: Temporal analysis for Application events and network RTT: QUIC application layer(AL) errors, and RTT with bytes transferred from a single IP: *172.217.138.105* over TCP and QUIC over a QUIC-enabled session

Fig. 4.10 shows an enlarged view of the application and network layer event of a sample Q_ENB session. We show application-layer events in terms of QUIC errors and QUIC RTT. The network layer events are shown in terms of bytes transferred over two protocols. We zoom in from 6000 sec to 9500 sec for an IP address: 172.217.138.105. We

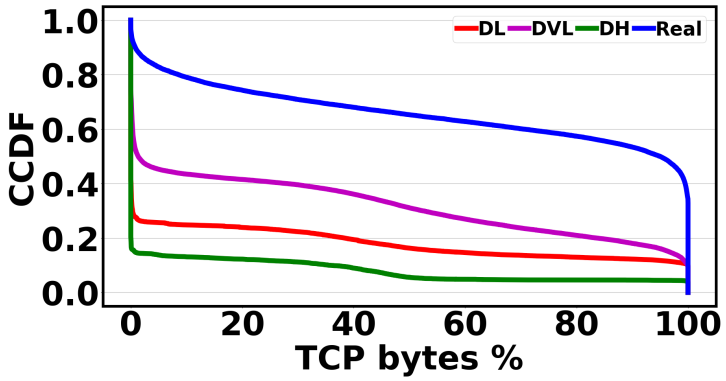
observe that whenever the QUIC RTT was above 10000 ms, or there were sudden spikes in QUIC RTT values, the QUIC errors such as *QUIC_TOO_MANY_RTOS* (error code: 85), *QUIC_NETWORK_IDLE_TIMEOUT* (error code: 25), *QUIC_STREAM_CANCELLED* occurred. Recall that whenever a QUIC-related error occurs, the alternate job is marked as broken and the main job is used to transfer the data (Sec . 2.1 in Chapter 2). The alternate job again tries to establish a connection after an exponential amount of time.

We observe that whenever QUIC errors occur in the application layer, with some lag, the data transfer starts over TCP. Further, the extent of connection racing varies with network RTT. (1) If the RTT fluctuates significantly, some of the QUIC connections were successful using the exponential connection retry attempt. In these cases, the extent of connection racing would be lesser (low - medium). (2) If the RTT remains continuously high, QUIC connection retry attempts might fail more frequently (at 7539 sec in Fig. 4.10). This, in turn, increases the exponential retry duration for QUIC establishment. Consequently, it increases the extent of connection racing (high). We validated the same hypothesis for 10 such use cases.

Hence, to summarize, the extent of connection racing will depend on whether the QUIC RTT remains consistently high or varies. If the RTT is varying which means the RTT is sometimes less than δ ms and sometime more than that then there will be fewer QUIC retry connection establishment failures, resulting in low- to medium-level connection racing. Otherwise, if QUIC RTT is more than a threshold δ ms, then there will be more QUIC connection establishment retry failures, leading to more connection racing. In the sample examples, δ is $9000-10000$ ms. After examining such individual examples, we perform an analysis on an aggregate basis to investigate how the network impacts connection racing instances.

4.6.3 Characterizing Connection Racing Across Networks

Fig. 4.11 shows the complementary CDF (CCDF) of connection racing ($\mathcal{R}Qvs.T_v$) at various network scenarios collected across 2007 YouTube sessions over Q_ENB browsers for both semi-controlled and controlled experiments ($161 + 1846$). For *DH*, there is minimal presence of TCP traffic for 80% of the cases. This is expected, as in high-bandwidth scenarios, there are fewer connection failures with QUIC.



(a)

FIGURE 4.11: Extent of connection racing: Percentage of bytes transferred over TCP across all YouTube over QUIC-Enabled Browser (Q_ENB) sessions for various network scenarios

For *DL*, *DVL* and *Real*, 40% or more bytes are transferred over TCP for 20%, 40% and 70% times, respectively. Hence, we observed that Q_ENB browsers tend to be involved in more connection switching events in a low- or variable-bandwidth network. This observation is also significant when compared to real network traces. Interestingly, the relative amount of connection switching events correlates with the number of cases where YouTube performance over Q_ENB browsers suffers more than the disabled ones.

We have the following takeaways.

Takeaway 2: *Connection Racing is expected when the network RTT is high or varying. Browsers cannot distinguish between a middlebox and a poor network, as both lead to QUIC connection failure/suffering. Hence, browsers engage in unnecessary connection racing.*

Takeaway 3: *Connection racing interacts poorly with long-lived video streaming applications due to frequent connection switching between QUIC and TCP, particularly when the network quality is not good. Further, there is a lag between the connection switching events between QUIC and TCP during their racing and the application QoE drop events.*

Because of the above reasons, YouTube over the Q_ENB browser faces additional stalls rather than experiencing a reduction in the connection setup latency using the 1-RTT or 0-RTT connection establishment process.

Motivated by this observation, we decided to disable connection racing within the browser, as we have confirmed that no middlebox is blocking or rate-limiting UDP traffic. The intuition is that eliminating unnecessary racing should improve application QoE, as our analysis shows that unnecessary protocol switching is a major contributor to poor performance in long-lived video streaming sessions. In the next section, we instrument the Chromium source code to implement this modification and evaluate its impact on QoE.

4.7 YouTube over a Pure QUIC Session

We modify the Chromium source[2] to manually disable the connection racing procedure and enable data transmission over a pure QUIC session. We have made this open-source, which can be accessed from the following link – `https://github.com/sapna2504/Chromium-Modification-V2` (Accessed: July 2026). We perform similar experiments under identical network and traffic conditions using both the original Chromium browser and the modified one to understand whether a pure QUIC session helps improve application performance in scenarios where the application performs poorly over the original Chromium browser.

4.7.1 Disable Connection Racing

We manually disable the connection racing option by modifying the open-source Chromium browser version *103.0.5025.0* (Developer Build) (64-bit). We make the following modifications to the connection racing implementation. (1) We stop the frequent fluctuations of the Chromium browser by delaying the `main` job by a significant value (30 sec). We empirically observed that this time lag is sufficient for the `alternate` job to bind with the request, even in low-bandwidth conditions. (2) When an error occurs, such as handshake failure, high packet losses in QUIC, etc., the `alternate` job is marked as broken and waits for the `main` job to succeed. To stop this, we remove all the functions that can mark the `alternate` job as broken. This prevents the calling of the `main` job for sending HTTP data. (3) We marked QUIC to be the only valid `Alt-svc` available so that HTTP2/SPDY can't become an `alternate` job even when the server advertises them as `Alt-svc` in the HTTP response header. (4) We set `retry_without_alt_svc_on_quic_errors` flag to be

[2] `https://www.chromium.org/Home/` (Accessed: July 2026)

false so that `alternate` job would remain available even in case of errors in QUIC connection. (5) If the `alternate` job is still not able to make a successful connection before the `main` job resumes and succeeds (which was delayed by 30 sec), we reset both the jobs and create the connection again with the `alternate` job.

Dataset: Fig. 3.4(a) shows the dataset of 1004 steaming session pairs, named as *S*2008, *S*2009, ..., *S*3011 collected using H3B tool. Each pair contains one session over YouTube with a QUIC-enabled original browser (OB) and QUIC-enabled modified browser (MB). The total duration is $\simeq$ 1453 hours ($\simeq$ 61 days), of which the original browser is 738 hours, and the modified browser is 715 hours.

4.7.2 Results

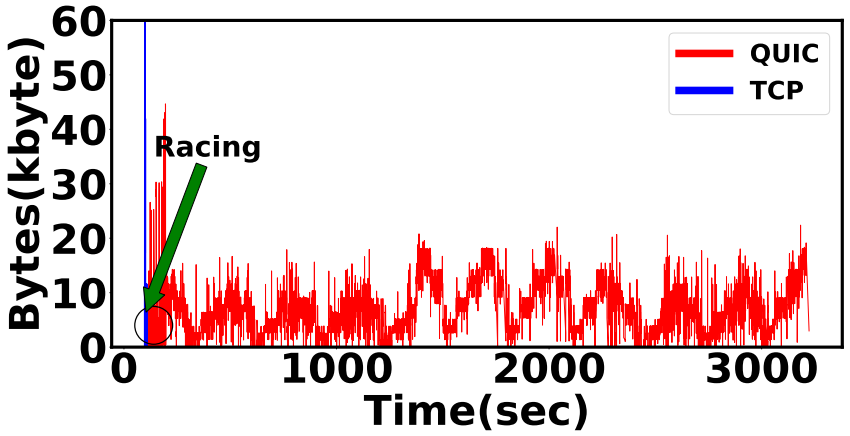


FIGURE 4.12: Results of Chromium browser; Number of bytes transferred from a single IP: *180.149.59.14* over TCP and QUIC over a QUIC-enabled modified browser (MB)

Fig. 4.12 shows the negligible presence of TCP traffic except in the beginning, which is only used to investigate QUIC support. All the modifications we made restricted the browser from using the main job. Hence, the browser enables QUIC to utilize its full potential and prevent unnecessary connection racing. We then examine sample video sessions to observe how the modified browser affects application QoE compared to the original case. Fig. 4.13(a, b, and c) shows the distribution of QoE metrics, i.e., average bitrate, average bitrate variation, and average stall for 6 sample streaming pairs: S1 to S6. We observe that the modified browser outperforms the original one in all QoE metrics for all streaming pairs except for S3. This is because we stop the frequent protocol switching

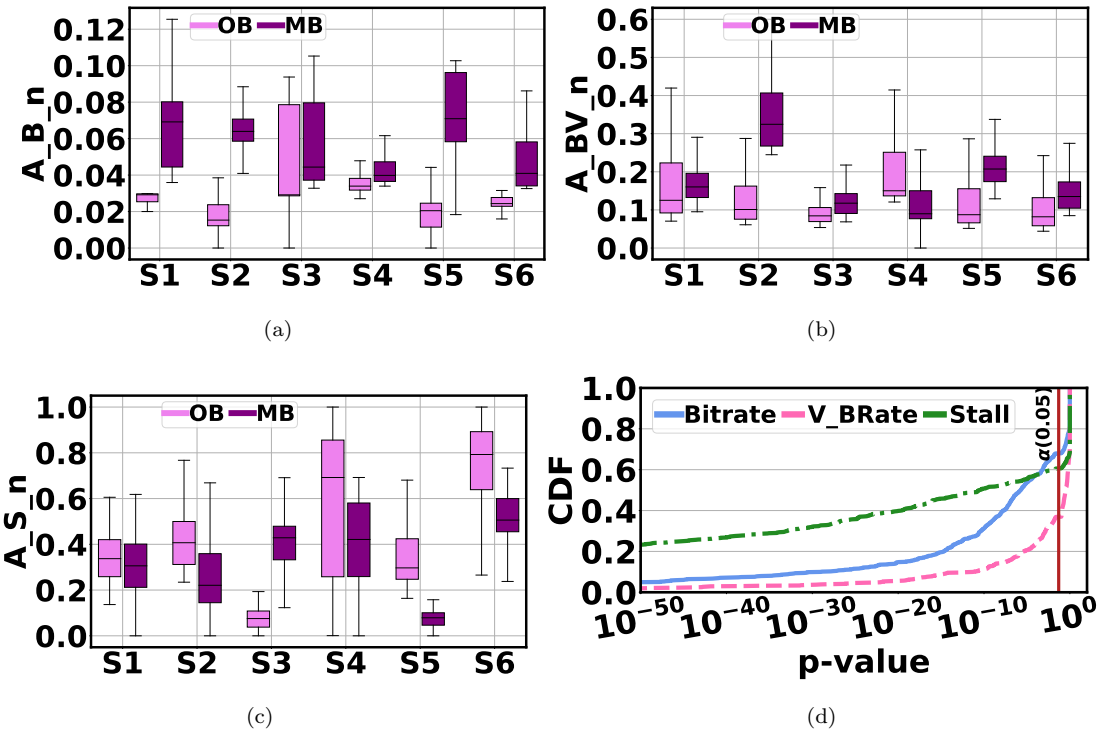


FIGURE 4.13: Normalized (a) average bitrate, (b) average bitrate variation, and (c) average stall distribution of 6 sample streaming pairs for original vs modified Chromium browser (d) Hypothesis testing on original vs modified Chromium browser

and allow streaming only over one protocol, QUIC. Stopping frequent protocol switching helps QUIC to utilize its true potential.

Fig. 4.13 (d) shows the result of hypothesis testing on all 1004 streaming sessions. The null hypothesis was accepted for 12%, 33%, and 8% cases for average bitrate, average bitrate variation, and average stall. For the rejected null hypothesis cases, we observe that YouTube over QUIC-enabled *modified* browser provides a better average bitrate compared to YouTube over QUIC-enabled *original* browser for 67% cases. Further, it experiences fewer stalls compared to QUIC-enabled original browser for 61% times and average bitrate variation 37% times when both *modified* and *original* QUIC-enabled browsers were streamed under similar network conditions.

This proves that unnecessary connection racing in a poor network is one of the primary reasons for poor application QoE for media streaming over modern browsers that support QUIC.

Next, we present two complementary solution approaches aimed at improving video

streaming QoE by optimizing how the browser handles connection racing. The first approach focuses on making connection racing itself adaptive, enabling the browser to decide dynamically when racing should be triggered based on real-time network conditions. The second approach aims to mitigate the adverse effects of protocol switching when it occurs, ensuring that the transition between QUIC and TCP does not disrupt playback. Together, these solutions address both the decision-making and performance consequences of connection racing, offering a more robust transport behavior for long-lived video streaming sessions.

4.8 Solutions to Improve Video Streaming Performance through Optimized Connection Racing

In this section, we present two solution approaches for improving application QoE over HTTP/3. These mechanisms are motivated by our key findings that (1) unnecessary connection racing negatively impacts application QoE, and (2) protocol switching leads to a reset of the congestion state, further degrading performance. First, we introduce our design of an intelligent, browser-side connection racing mechanism that dynamically decides when racing should occur. Next, we present our congestion state transfer solution, which preserves congestion information across protocol switches to minimize the impact.

4.8.1 *Dynamic-Racing*: Intelligent Connection Racing Mechanism

We develop a dynamic solution named *Dynamic-Racing* to make a connection racing decision. Our solution is based on our observations that connection racing is highly correlated with the RTT. We perform experiments under a poor network and a UDP rate-limiting firewall using both the original Chromium browser and the modified one to understand whether our solution helps improve application performance in scenarios where the original Chromium browser suffers.

4.8.1.1 *Dynamic-Racing*

At the browser side, we aim to distinguish between firewall blocking/rate-limiting of QUIC packets vs a poor network causing QUIC packet losses. The key idea in our solution is the following: *we monitor the RTT of QUIC and TCP at the browser. If comparable, it indicates that the network is providing the same treatment to both protocols, and we should not allow connection racing. On the other hand, if QUIC's RTT is not comparable to TCP's and is higher, it indicates that the network is not providing the same treatment to both. Specifically, QUIC is facing additional challenges; hence, we allow connection racing in that case.*

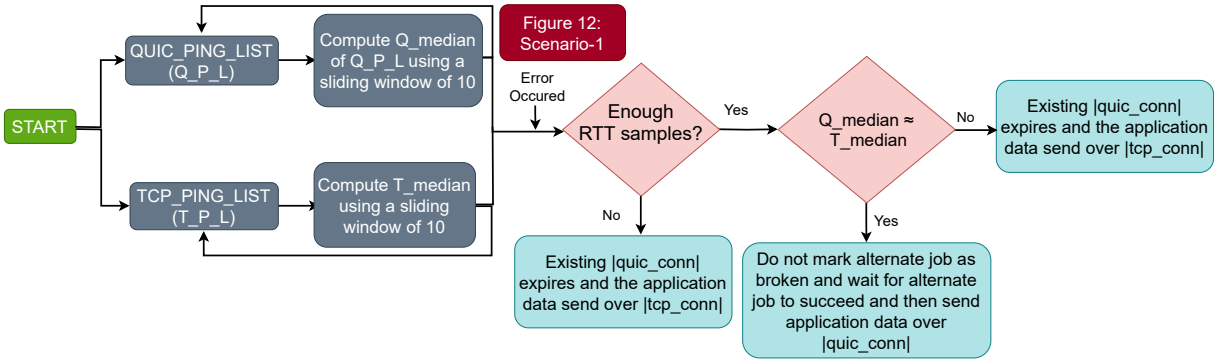


FIGURE 4.14: Dynamic connection racing solution workflow

Fig.4.14 shows the design of our solution. We modify the Chromium source[3] to implement the dynamic solution. We have also made this open-source, which can be accessed from the following link – `https://github.com/sapna2504/Chromium-Modification-V2` (Accessed: July 2026). We monitor the RTT of QUIC and TCP using NQE (Network Quality Estimator) [64]. NQE transmits QUIC frames to obtain QUIC RTT value and monitors the existing TCP sockets to obtain TCP RTT. As the mechanisms used for obtaining RTTs differ (active for QUIC and passive for TCP) and are not directed to the same server, such ping values are not comparable. Therefore, we develop an active probing mechanism for both TCP and QUIC. We perform such pinging once every 15 sec to reduce overhead due to pinging in a poor network. Whenever the QUIC connection is established, the browser sends the first ping to perform a connection migration check. We utilize this first ping to set further alarms (every 15 seconds) for future pings. The QUIC RTT was computed by subtracting the packet sent time from the received time (it provides accurate network RTT estimation [9]). Similarly, we utilized the HTTP/2 pinging

[3] `https://www.chromium.org/Home/` (Accessed: July 2026)

mechanism for TCP, which transmits TCP pings every 15 seconds using *HttpStreamFactory*. We enable this HTTP/2 pinging mechanism only for the sessions attached to the YouTube server by checking a hostname. It allows enabling and disabling the alt-svc for any request. We disable alt-svc QUIC, and therefore all the pings are sent over TCP at regular intervals of 15 seconds. We also ensure that both TCP and QUIC pings are sent to the same server. We, therefore, first check the hostname and if it is "www.youtube.com", then we send the TCP ping for that server. We set the server IP using the environment variable *ping_addr*. We use this environment variable in the QUIC pinging mechanism before sending a QUIC ping.

We compute the median of QUIC and TCP RTT values for a sliding window of size 10. If the median values are similar, we announce that the network is providing the same treatment to both. Hence, we don't allow connection racing. We achieve the same by setting the environment variable *SHOULD-FALLBACK* to false. Otherwise, if the median RTT value of QUIC is X times higher than TCP, we announce that QUIC is facing additional challenges. Hence, we allow connection racing and we set the same environment variable to true to allow the browser to behave like the original browser. Note that X value depends on the rate limit applied on UDP. Empirically in our case, we found that a value of $X \geq 3$ works good. Note that such a comparison of pings is event-driven, i.e., the moment the browser faces any stream error and/or timeout(s), we compare the RTT values. Until we have enough RTT samples to make a decision, we allow the browser to behave as it would have otherwise.

4.8.1.2 Dataset & Results

Dataset: We stream YouTube videos under a poor network (DVL) and UDP rate-limiting firewall using our H3B tool. Note that we have rate-limited the UDP packets at port 443 to a rate of 64 kbps using qdisc. For the poor network (DVL) case, the total duration is $\simeq 56$ hours, of which the original browser is 28 hours, and the modified browser is also 28 hours. For firewall the total duration is $\simeq 22$ hours, of which the original browser is 11 hours, and the modified browser is also of same 11 hours. Fig 3.4(b) in Chapter 3 shows the dataset of 76 video session pairs of poor network and firewall, named $S3012, S3013, S3014, ...S3087$.

Comparison of RTT under poor network vs UDP rate-limiting firewall Fig. 4.15 shows the RTT distribution of QUIC and TCP under a poor network and a rate-limiting

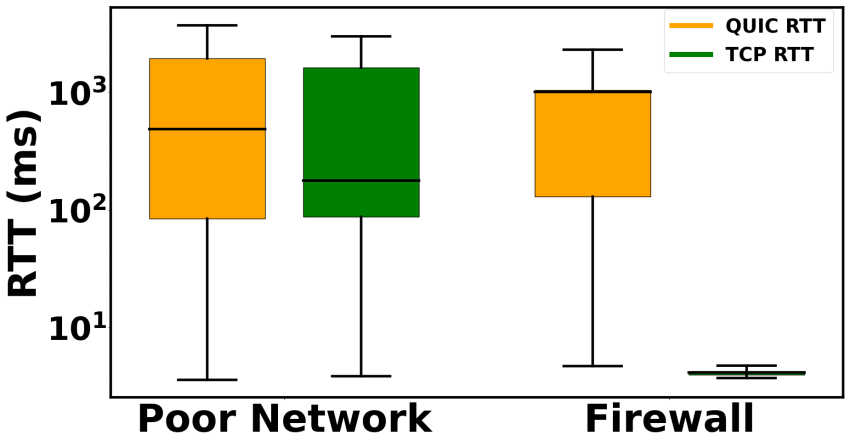


FIGURE 4.15: QUIC and TCP RTT under poor network with 170 QUIC RTT samples and 140 TCP RTT samples, and rate-limit UDP:443 firewall with 171 QUIC RTT samples and 143 TCP RTT samples.

firewall. We observe that QUIC and TCP RTT values are similar in poor network conditions. However, the former's RTT values are higher than the latter for firewalls. We have utilized this key observation to develop our solution.

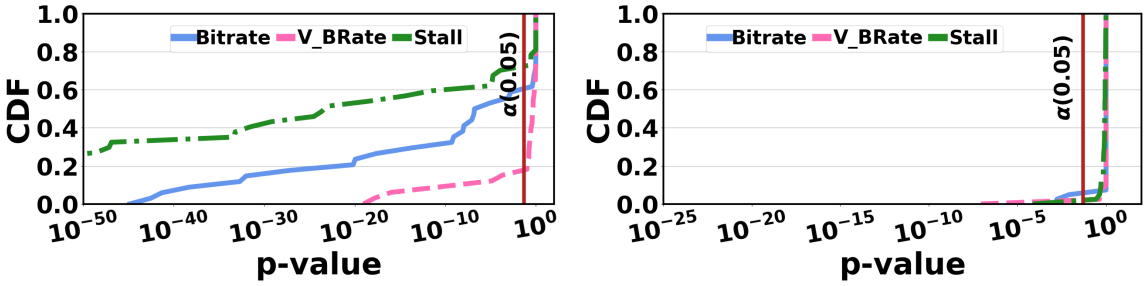


FIGURE 4.16: (a) Poor Network: Hypothesis test result for average bitrate, average bitrate variation, and average stall of 38 sample streaming pairs for original vs modified browser (b) Firewall: Hypothesis test result for average bitrate, average bitrate variation, and average stall of 38 sample streaming pairs for original Browser vs modified Chromium browser

Comparison of QoE Fig. 4.16(a) and (b) show the result of hypothesis testing for a poor network and a UDP rate-limited firewall, respectively. For a poor network, the null hypothesis was accepted for 16%, 54%, and 7.8% cases for average bitrate, average bitrate variation, and average stall, respectively. For the rejected null hypothesis cases, YouTube over QUIC-enabled *modified* browser provides a better average bitrate compared to YouTube over QUIC-enabled *original* browser for 58%, lesser average bitrate variation for 17% cases, and lesser average stall 71% of cases. Thus, our solution was able to detect the reason for packet losses as the poor network and did not allow connection racing,

unlike the original browser. Hence, provides better QoE. In the presence of a firewall, the null hypothesis was accepted for 90%, 97%, and 97% cases for average bitrate, average bitrate variation, and average stall, respectively. This indicates that both OB and MB provide similar QoE in the case of the firewall, as in this case, our solution detected a firewall by comparing the RTT and allowed connection racing to happen like the original browser.

Next, we examine the relationship between congestion state and protocol switching, highlighting how congestion information is handled during a protocol switch and the consequences that arise when a connection transitions from one transport protocol to another. Building on these insights, we then present a mechanism that mitigates the performance degradation caused by protocol switching.

4.8.2 Protocol Switching vs. Congestion State

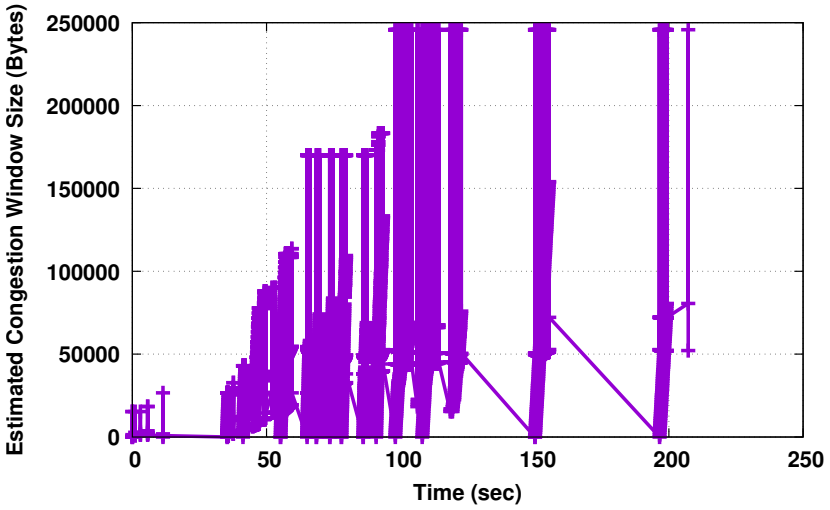


FIGURE 4.17: HTTP/3: Estimated congestion window size for an existing TCP connection with IP *172.217.139.200* and port *20246*; as observed during analysis

Whenever a protocol switch occurs from QUIC to TCP, application data is transferred via the new protocol. However, the congestion state that the QUIC protocol achieved is not shared with TCP. This is understandable, as QUIC is implemented in user space, and TCP is implemented in kernel space. There is no communication between the user space and kernel space. Hence, the application incurs a lag as the data transfer over QUIC halts and restarts over TCP. Suppose, at time t, QUIC has reached a particular congestion state cw. Now, when the browser chooses to fall back to TCP and data transfer starts with

TCP, it will not start from *cw*. During connection racing, the browser may initiate a new TCP connection, or it may resume an existing TCP connection. If it is a new connection, there will be significant lag, as a connection must first be established through a 3-way handshake procedure. Next, TCP will start with a slow start phase and take its own time to reach a congestion state, similar to the one that QUIC achieved. When it resumes an old TCP connection, it will either start from the previous congestion state, where past data transfer had finished over TCP, or begin with a slow start phase again, depending on when the browser chooses to return to this connection. But in either case, it will not be the same as where QUIC had reached (as the network is dynamic).

Fig. 4.17 shows the estimated congestion window size when the browser reuses an existing TCP connection. We obtain the estimated congestion window by using *tcptrace* [66]. We can observe that the TCP connection was initially used for data transfer for about 100 sec. Then, the data transfer occurs over QUIC; the browser switches back to TCP again at 150 seconds. At that time, the congestion window is reset, starting from the beginning again. Then again, switching happens at 200 seconds, and the congestion window is reset at 200 seconds. Thus, during a protocol switch, TCP requires additional time to reach the congestion window level at which the QUIC connection previously operated. Since any new TCP connection begins in the slow start phase, this transition introduces further delay. During this period, the application is unable to utilize the available network capacity effectively, resulting in degraded streaming performance. Next, we propose a server-side solution that leverages this observation to minimize the impact of protocol switching caused by connection racing whenever it occurs. We emphasize that this mechanism operates entirely on the server side.

4.8.3 Minimizing the Impact of Protocol Switching through eBPF-based Congestion State Transfer

As we have observed in Sec. 4.8.2, during protocol switching the congestion state is not passed from QUIC to TCP and vice versa. As a result, the application suffers. To address this issue, we design an eBPF-based solution to transfer the congestion state on occasions of fallback. We aim to reduce the time duration for which the application suffers. We now discuss the eBPF-based solution in detail.

4.8.3.1 Solution Design

In the case of QUIC to TCP protocol switching, the congestion state of QUIC is not transferred to TCP, as QUIC is implemented in the user space, and TCP is implemented in the kernel space, since there is no communication between the user space and kernel space. Hence, to reduce the impact of protocol switching, we transfer the congestion state from one protocol to another using eBPF (Extended Berkeley Packet Filter) on the server side. A key feature of eBPF is its use of eBPF maps, which serve as shared data structures accessible from both user space and kernel space. This bi-directional sharing capability enables the QUIC implementation in user space to write congestion-state information (e.g., cwnd) into an eBPF map, while an eBPF program running in the kernel can subsequently retrieve and apply this information to a TCP connection.

Our solution works as follows: when a protocol switch from QUIC to TCP occurs, an eBPF program is attached to the TCP socket. Depending on the scenario, the TCP connection may be newly initiated (active open) or may attach to an existing passive connection. Upon activation, the eBPF program reads the congestion window value previously stored in the eBPF map by the user-space QUIC process and updates the kernel TCP socket with this value. In doing so, TCP begins transmission with a congestion window that closely matches the last known QUIC state, thereby avoiding the delay imposed by slow start and enabling a smoother transition between transport protocols.

4.8.3.2 Analysis

To evaluate this mechanism, we deploy a DASH player integrated with the Chromium browser as the client and an OpenLiteSpeed server configured both with and without our eBPF-based congestion-state transfer solution. The DASH player manages adaptive video streaming at the client side, while the OpenLiteSpeed server is responsible for delivering video segments under controlled network conditions. This setup allows us to systematically observe, quantify, and compare the impact of congestion-state transfer on QoE during QUIC-to-TCP protocol switching.

For our experiments, we select a test video of a big buck bunny [67]. We encode the video at a wide range of bitrates with a segment duration of 1 second: 4, 2, and 1 Mbps, along with an additional representation at 500 Kbps. We conduct video streaming experiments both with and without the eBPF-based congestion-state transfer mechanism, referring to

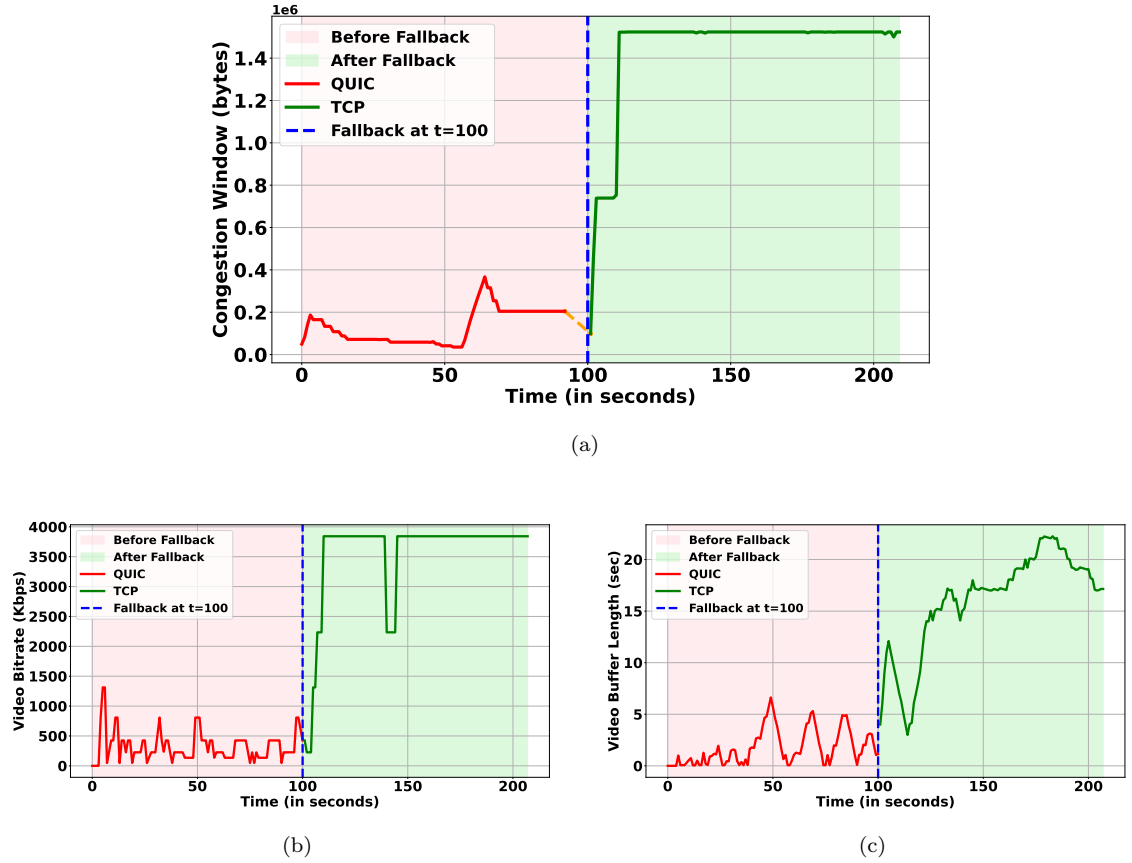
these configurations as *W-eBPF* and *W/O-eBPF*, respectively. We log both QUIC and TCP congestion values at the server side: QUIC congestion information is recorded in user space through our instrumented QUIC stack, while TCP congestion window values are captured in the kernel via our eBPF program. The resulting measurements are written to two separate files (quic_readings.log and tcp_readings.log), and later processed for analysis. We also collect application-level metrics such as stall events and bitrate selections from the DASH player. This is achieved by instrumenting the `main.js` JSON file within the DASH player, enabling it to record these metrics at 10 ms intervals and store them in a CSV file for subsequent analysis. For our experiment, we emulated a scenario in which UDP traffic is rate-limited by a firewall while TCP remains unaffected. To trigger a controlled fallback event, we completely blocked UDP on port 443 at the 100-second mark, forcing the browser to switch from QUIC to TCP. This setup enables us to evaluate the effectiveness of our eBPF-based congestion window transfer mechanism during a real protocol switch.

Figure 4.19 illustrates the behavior of the congestion window during a QUIC-to-TCP protocol switch when the eBPF-based congestion-state transfer mechanism is enabled, along with its resulting impact on bitrate selection and video buffer evolution. When compared to the W/O-eBPF results 4.18, a clear performance improvement is observed.

Without an eBPF-based solution, TCP initiates slow start and takes a substantial amount of time to rebuild its congestion window, resulting in a temporary drop in throughput. This reduced throughput forces the DASH player to select lower bitrates and, in many cases, leads to noticeable buffer depletion. As a result, the application experiences degraded QoE in the form of lower video quality or increased risk of rebuffering.

In contrast, with the eBPF-based congestion-state transfer enabled, the TCP connection inherits the congestion window previously maintained by QUIC. Because TCP does not need to re-establish its sending rate from scratch, it immediately begins transmission at a rate that closely matches the pre-switch network capacity. This behavior is reflected in the congestion window graph, where TCP rapidly stabilizes within a few milliseconds. Consequently, the bitrate graph shows that the DASH player quickly selects and sustains higher quality video representations, avoiding the prolonged downward bitrate shift observed in the W/O-eBPF case. Similarly, the buffer-length graph experiences no reduction in buffer occupancy, indicating smoother playback and a reduced likelihood of rebuffering.

Overall, these results demonstrate that transferring the congestion state across protocols significantly reduces the performance penalty typically associated with switching from



(a)

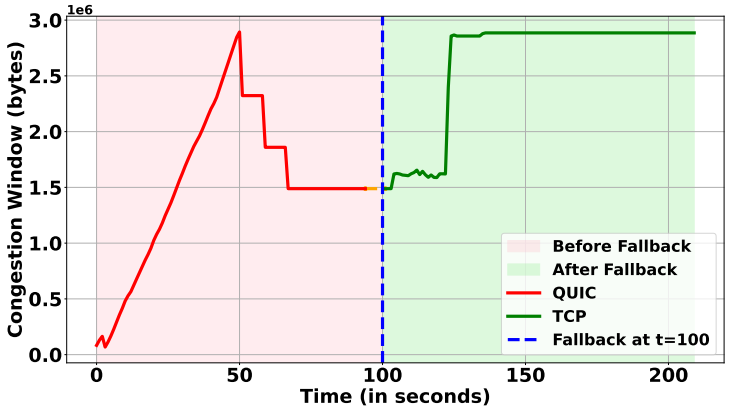
(b)

(c)

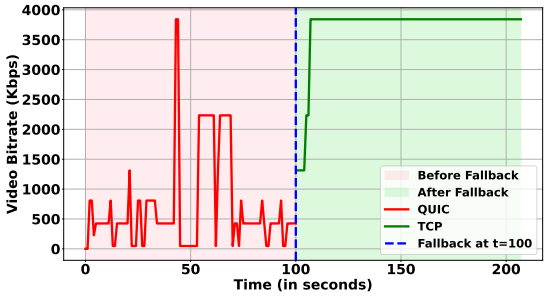
FIGURE 4.18: Protocol switching under the W/O-eBPF configuration, showing (a) congestion window, (b) bitrate, and (c) buffer length vs time (seconds) graph

QUIC to TCP. By eliminating the need for TCP to re-enter slow start and rapidly rebuilding the congestion state, the eBPF-based approach ensures a smoother transition between protocols, resulting in consistently higher bitrates, improved buffer stability, and a clear improvement in overall video QoE.

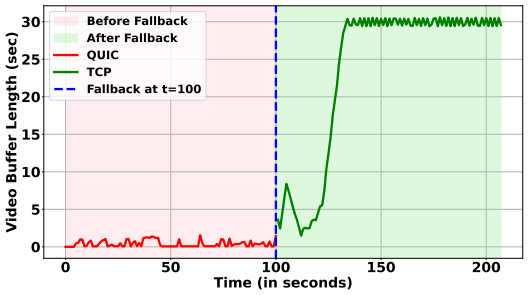
The results from our congestion-state transfer experiments reinforce the importance of protocol switching; when QUIC is impaired, such as in the presence of UDP rate limiting, switching to TCP is essential for maintaining connectivity and QoE. By transferring congestion state across protocols, we further minimize the performance penalty of such switching events, ensuring a smoother transition during QUIC to TCP fallback.



(a)



(b)



(c)

FIGURE 4.19: W-eBPF Protocol Switching: (a) Congestion Window, (b) Bitrate, and (c) Buffer Length vs. Time (seconds) graph

4.9 Summary of the Chapter

In this chapter of the thesis, during an HTTP/3 session, we observed repeated protocol switching due to connection racing, which negatively impacts application performance. This fluctuation between protocols is alarming for the streaming session. Finally, our analysis of more than 5474 streaming hours of video data can provide a reliable indication of the impact of connection racing on YouTube's performance over QUIC-enabled browsers from an application perspective. As application performance is affected by the connection racing implementation in the browser, which is independent of the application running on top of it, we believe similar observations will apply to other streaming media services as well if they support HTTP/3 on top of the existing browser implementations.

An interesting observation is that the browser does not differentiate between failures caused by a middlebox and those due to network congestion, which necessitates a thorough

investigation to support stable QUIC performance over the Internet. The sheer variance in the real world makes it impossible to measure precisely. Although we have attempted to emulate the real world to understand what happens beneath the surface when an application like YouTube streaming suffers over a QUIC-enabled browser, our observations also have some limitations, as follows. **Connection racing might not be the only cause:** Our analysis indicates that connection racing is a prime reason behind the poor performance of YouTube streaming over a QUIC-enabled browser. However, there are instances where QUIC-enabled sessions perform poorly even when there is no protocol switching, such as in the case of a high-bandwidth network. Furthermore, even when connection racing is manually turned off after modifying the browser, the original browser still outperforms the modified browser in about 40% of instances. Therefore, further research is required to determine the other reasons that affect the QUIC performance.

Our findings reveal that suboptimal streaming performance is not always a consequence of inherent protocol limitations; external factors such as the browser's connection racing implementation, as demonstrated in this chapter, can significantly influence how a transport protocol performs under real network conditions. The findings suggest that QoE impairments frequently originate from implementation-level interactions, particularly within the browser, rather than from fundamental protocol limitations. Moreover, even when switching is optimized, no single transport protocol consistently delivers superior performance across all environments. Prior studies show that QUIC tends to excel in lossy or unstable networks, while TCP remains more robust in high-speed settings. This diversity in behavior highlights the need for a more intelligent, adaptive mechanism that can dynamically select the appropriate protocol based on real-time network and system conditions, rather than relying on static defaults. Motivated by these insights, the next chapter introduces an adaptive transport protocol switching framework designed to optimize video streaming QoE across heterogeneous and rapidly changing network environments.

Chapter 5

Learning to Switch: Adaptive Transport Protocols for High-Performance Video Streaming

'It is not the strongest of the species that survives, nor the most intelligent, but the one most responsive to change."

— Charles Darwin

Several prior works [9, 12, 13, 19] have shown that QUIC performs better than TCP in poor and lossy networks due to its ability to avoid head-of-line (HoL) blocking and recover faster from losses. On the other hand, prior studies [13, 14] have also reported several scenarios in which QUIC performs worse than TCP. For example, QUIC suffers under packet reordering and jitter due to premature loss detection, which limits congestion window growth and prevents efficient bandwidth utilization. Similarly, QUIC performs poorly for large object transfers and multiple concurrent object downloads compared to TCP. These observations indicate that no single transport protocol consistently performs best across all network and system conditions. In the previous chapter, we further showed that even in scenarios where QUIC is expected to perform better, its performance can still degrade due to browser-level implementation behavior. In particular, we observed that browser-level connection-racing mechanisms unnecessarily trigger protocol fallback by misinterpreting temporary network degradation as QUIC rate limiting or blocking. As a result, even when QUIC connectivity persists, the browser repeatedly switches between QUIC and TCP, negatively impacting video streaming QoE. To address this issue, we

introduced a browser-side intelligent mechanism that enables protocol switching only when necessary. Moreover, with the evolution of modern high-bandwidth networks such as 5G and Wi-Fi 6/7, prior studies [11, 15, 16, 17] have reported scenarios in which TCP outperforms QUIC due to additional host-side processing overheads associated with QUIC. Therefore, our findings, together with observations from prior studies, collectively indicate that no single transport protocol consistently performs best across all network and system conditions. Therefore, rather than relying only on static protocol selection or browser-specific optimizations, there is a need for an adaptive framework that can dynamically decide when to switch transport protocols based on real-time network and system behavior in order to improve overall video streaming performance.

Therefore, in this chapter, we first motivate the need for adaptive transport protocol switching by highlighting that no single transport protocol consistently performs best across diverse and dynamic network conditions. We then present IntelSwitch, an adaptive framework designed to intelligently switch between TCP and QUIC for video streaming applications based on real-time network and system behavior.

5.1 Introduction

The proliferation of high-speed access technologies, such as Wi-Fi 6/7 and 5G, offering bandwidths ranging from hundreds of Mbps to multiple Gbps, is reshaping the Internet delivery landscape. Indeed, industry forecasts [1] predict that by 2028, the majority of global Internet traffic will traverse 5G networks. These developments highlight an urgent need for transport protocols that can not only withstand lossy and variable conditions but also fully exploit the capacity and stability of next-generation high-speed infrastructures.

Despite the widespread adoption of QUIC, it faces challenges in high-speed networks. Studies [12, 16, 22] show that host-side processing overhead in its user-space implementation limits throughput, with Zhang *et al.* [16] reporting up to 45% reduction and a 9% bitrate drop for video streaming compared to TCP. Bi *et al.* [22] further highlight server-side overhead under concurrent HTTP/3 sessions, while QUIC's reliance on UDP exposes it to ISP rate-limiting, further degrading performance. In contrast, QUIC excels in lossy or variable networks: Google's seminal study [9] showed 15–18% lower buffering on YouTube, and subsequent works [12, 13, 68] confirm its superiority over TCP in poor

or fluctuating conditions. These results underscore a key limitation: no single transport protocol consistently achieves optimal QoE across all environments.

Motivated by this observation, this chapter first establishes the need for adaptive transport protocol switching for video streaming by experimentally comparing the performance of TCP and QUIC across a range of realistic network scenarios. We then discuss the key challenges in designing such an adaptive mechanism, including handling dynamic network conditions, accurately estimating real-time network metrics, accounting for the intrinsic characteristics of video streaming applications, managing protocol switching overhead, addressing the complexity of modifying the Chromium browser, and ensuring deployability without requiring server-side changes. Building on these insights, we introduce IntelSwitch, an adaptive transport protocol switching framework, describe its system architecture and implementation, and detail the experimental setup used for its training and evaluation. Finally, we define the evaluation metrics and present comprehensive results demonstrating the effectiveness of adaptive protocol switching in improving video streaming QoE.

5.2 Contribution

This chapter of the thesis makes the following contributions:

1. We demonstrate that the QoE of video streaming applications can significantly degrade on high-speed links when QUIC is used as the transport protocol, primarily due to client-side processing overhead. Through a series of pilot experiments, we observe that no single static transport protocol is optimal across all network conditions.

2. We propose IntelSwitch, a deep reinforcement learning (DRL) based framework that dynamically selects a suitable transport protocol (TCP or QUIC) for each video segment based on network conditions and application QoE. We implement IntelSwitch by extending both the `dash.js` player and the Chromium browser to enable adaptive protocol switching. We contributed to about 140 and 503 lines of code to `dash.js` and the Chromium browser.

3. We conduct extensive evaluations of IntelSwitch under 9 diverse network patterns covering varied types of networks, from controlled to real 5G networks. We compare

its performance against static protocol choices (TCP and QUIC) and a Heuristic baseline. IntelSwitch improves the application QoE by upto 47% over only TCP, 114% over only QUIC and 58% over the Heuristic baseline.

5.3 Motivation

Video streaming performance is highly sensitive to the choice of transport protocol, with QUIC (HTTP/3) and TCP (HTTP/2) each offering distinct strengths depending on the network environment. This asymmetry creates an important challenge: neither protocol alone consistently ensures the best Quality of Experience (QoE), motivating the need for adaptive protocol switching.

QUIC's strengths in challenging networks: QUIC was designed to improve performance in unreliable or lossy environments, which are common in mobile and wireless settings. By enabling faster connection establishment (0-RTT), multiplexing streams to avoid head-of-line (HoL) blocking, and handling losses more efficiently than TCP, QUIC significantly reduces rebuffering events and increases average playback bitrates. Multiple studies confirm its superiority over TCP in such scenarios [9, 12, 13, 33, 68, 69].

QUIC's challenges in high-speed networks: In contrast, QUIC struggles in modern high-speed, low-latency environments such as WiFi 6/7 and 5G. Its user-space implementation of ACK processing and encryption introduces substantial computational overhead, limiting throughput [16]. At gigabit speeds, packet I/O becomes the dominant bottleneck [12, 43], with QUIC consuming nearly twice the CPU resources of TCP to deliver comparable throughput [12, 22]. As a result, TCP often outperforms QUIC on well-provisioned links.

Dynamic trade-offs across conditions: These contrasting performance profiles imply that a static choice of transport protocol is inherently suboptimal. TCP is better suited for high-bandwidth, low-latency environments where CPU overheads dominate, while QUIC excels in lossy, high-latency, and mobile networks. However, real-world conditions are highly dynamic, with performance shifting due to user mobility, fluctuating network load, and varying host resources. This variability motivates the need for an intelligent, adaptive framework that can switch between TCP and QUIC based on real-time network and QoE metrics, ensuring consistently high streaming performance across heterogeneous environments.

Further, to better understand the practical implications of protocol choice in video streaming, we first conduct a pilot study evaluating the performance of TCP and QUIC across a range of high-bitrate scenarios. This analysis reveals the specific conditions under which each protocol provides advantages or suffers performance degradation, thereby motivating and shaping the adaptive mechanism introduced later in this chapter.

5.3.1 Pilot Study

Motivated by prior findings, we investigate the performance gaps between TCP and QUIC in high-bitrate video streaming. To this end, we conduct motivation experiments under diverse network conditions, ranging from high-bandwidth to lossy environments. The setup consists of three components: (1) a client running a DASH player with BOLA as the ABR algorithm, (2) a server hosting video segments, and (3) a network emulator (mahimahi [70]). We use a video from the FFmpeg filter testing set [71], encoded at bitrates from 500 Kbps to 700 Mbps, and evaluate across five networks (1) high-bandwidth Ethernet (1 Gbps), (2–3) variable high-bandwidth 5G mid-band drive and mmWave walk traces [72], (4) variable low-bandwidth 4G walk trace [73], and (5) poor-bandwidth with 5% packet loss and 50 ms delay emulated using `tc netem` [74].

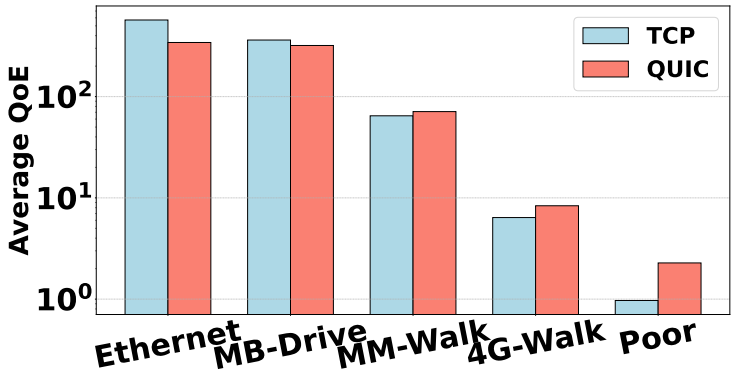


Figure 5.1: Average QoE for TCP and QUIC across different network scenarios

Fig. 5.1 shows the average QoE of TCP and QUIC across these scenarios. On Ethernet, TCP reaches the maximum 700 Mbps bitrate, while QUIC is capped around 350 Mbps. In the 5G mid-band drive (MB-Drive), TCP again outperforms QUIC, particularly above 500 Mbps, consistent with prior work attributing QUIC's limits to client-side overhead [16]. CPU pressure measurements (Fig. 5.2) further support this, despite lower throughput; QUIC exhibits equal or higher CPU pressure than TCP, confirming earlier findings that

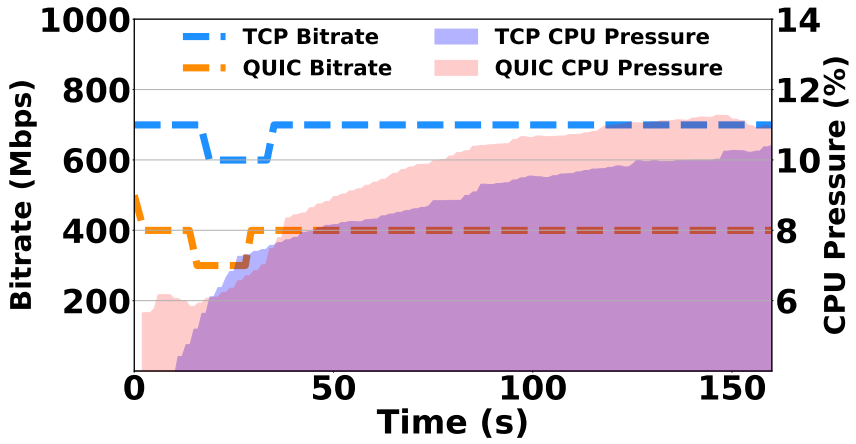


FIGURE 5.2: Bitrate (Mbps) vs Time (seconds) of TCP and QUIC while streaming a video at 1Gbps bandwidth and the corresponding CPU pressure (%)

user-space processing bottlenecks hinder QUIC at high bandwidths [12, 16, 22]. Conversely, in poor or lossy networks (poor-bandwidth and 4G-walk), QUIC outperforms TCP due to its ability to avoid head-of-line (HoL) blocking and recover faster from packet losses. HoL blocking in TCP occurs when the delivery of subsequent packets is delayed because TCP enforces in-order delivery of all bytes within a connection. As a result, the loss of a single packet can temporarily block the delivery of later packets, even if those packets have already arrived successfully at the receiver. In the context of video streaming applications such as YouTube, such blocking can delay the delivery of video or audio segments to the player, potentially increasing playback stalls and reducing streaming quality under lossy network conditions. QUIC mitigates this issue by supporting independent streams at the transport layer, allowing unaffected streams to continue progressing even when packet loss occurs in another stream. In moderately fluctuating yet provisioned networks (e.g., stable 4G/5G), the benefits of QUIC are less pronounced.

These results highlight a clear trade-off: TCP dominates in high-speed environments, while QUIC excels in lossy conditions. This motivates the need for an adaptive switching mechanism that selects the most suitable transport protocol based on real-time network and host conditions. Realizing such a mechanism requires coordinated changes to both the video player and browser infrastructure. Before presenting our system design, we first outline the key challenges that make optimal transport protocol selection in real-world environments a non-trivial problem.

5.4 Design Challenges

Deciding the optimal transport protocol in real-world network environments is a non-trivial problem due to several key challenges:

Dynamic network conditions make video streaming performance highly sensitive to fluctuations in bandwidth, RTT, and packet loss, particularly in mobile or wireless environments. Existing approaches that rely on coarse network categories (e.g., 2G/3G/4G/WiFi) or static supervised models fail to capture temporal dependencies and unseen states, underscoring the need for continuous, adaptive learning from real-time conditions.

Handling video streaming characteristics introduces additional complexity, as adaptive video streaming involves intricate buffer dynamics, segment quality variations, and playback stability considerations that directly influence user experience. This necessitates an adaptive transport protocol selection mechanism capable of learning from evolving network and playback contexts to optimize long-term QoE.

Overhead of protocol switching presents another difficulty: switching protocols midstream often requires new handshakes, potentially causing playback delays or interruptions that degrade QoE, thereby demanding efficient mechanisms to minimize disruptions.

Moreover, **accurate network metric estimation** remains a significant hurdle since browsers operate at the application layer and lack direct access to precise network metrics such as RTT, bandwidth capacity, and loss rate. Instead, they rely on noisy or delayed indirect estimates, complicating real-time assessment of network states and their effects on playback performance.

The **complexity of Chromium browser modification** further amplifies the challenge, as implementing dynamic protocol switching necessitates intricate code changes across multiple layers of Chromium's large, multi-layered architecture, where errors can adversely impact QoE.

Finally, **deployment without server modifications** is essential, as practical solutions must function entirely on the client side without requiring server-side changes, relying solely on browser-level data and interactions.

Next, we discuss the design of IntelSwitch to address these challenges.

5.5 INTELSWITCH: System Overview and Implementation Details

In this section, we discuss the IntelSwitch system overview and the implementation details of IntelSwitch and its components.

5.5.1 IntelSwitch: System Overview

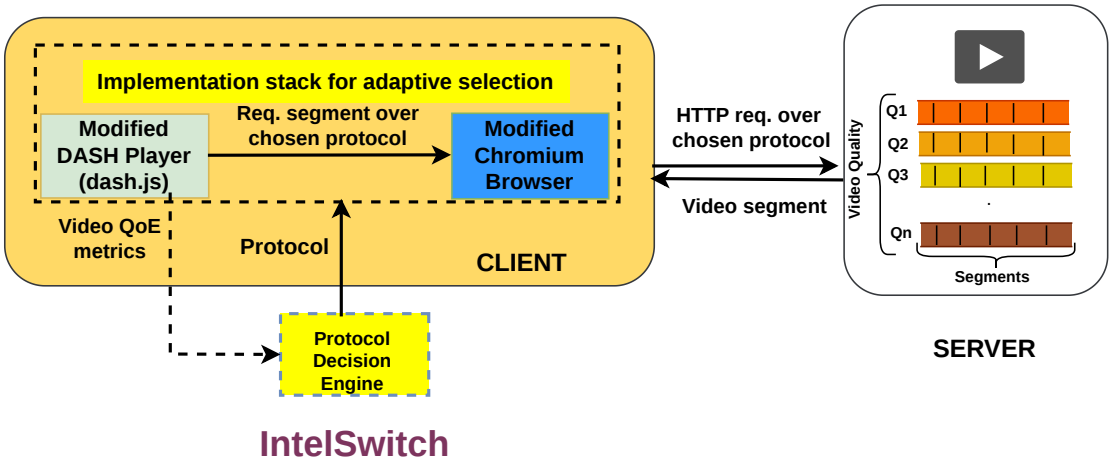


Figure 5.3: A schematic view of IntelSwitch for video streaming application. The coloured box indicates a client-only solution. Two major building blocks: 1) Protocol decision engine, 2) Implementation stack.

We design IntelSwitch, a fully client-side system that dynamically selects the transport protocol during a video streaming session to optimize user Quality of Experience (QoE). Fig. 5.3 provides a schematic overview of IntelSwitch. IntelSwitch introduces two major components: (1) A **protocol decision engine** that selects the best transport protocol adaptively located outside the client in another machine, (2) an **implementation stack that enables adaptive transport selection**. It modifies the video player and browser stack to allow adaptive protocol switching during a video stream. IntelSwitch works as follows: **(Step 1)** the video player sends video QoE metrics to the decision engine. **(Step 2)** The decision engine then chooses the best protocol and sends it back to the video player. **(Step 3)** The modified player then sends the video segment request to the browser while specifying the transport protocol. **(Step 4)** The modified browser finally sends the appropriate HTTP request to the video streaming server over the chosen protocol.

To overcome the challenges discussed in § 5.4, we carefully designed each component of IntelSwitch:

1. Adaptive protocol selection under dynamic conditions: We design *Protocol Decision Engine*, which leverages a deep reinforcement model for adaptive decision-making. Our engine continuously learns from real-time network and playback metrics collected during the streaming session. This enables the decision engine to capture complex temporal relationships between network behavior (bandwidth, RTT, loss rate) and application-level QoE factors (buffer level, bitrate, dropped frames), overcoming the limitations of coarse categorization and supervised learning.

2. Handling video streaming specifics: IntelSwitch modifies DASH player to track playback-relevant metrics such as buffer dynamics, segment quality variations, and playback stability in real time. IntelSwitch works at a segment level where the decision engine selects the right protocol for each segment. This allows the system to balance bitrate and rebuffering risk when making protocol decisions. It avoids purely reactive choices and instead optimizes long-term QoE stability.

3. Overcoming protocol switching overhead: Switching transport protocols midstream introduces connection re-establishment delays, potentially degrading playback. To address this, IntelSwitch proactively initiates both TCP and QUIC connections at session start, assigning audio requests to TCP and video requests to QUIC. Later during the session, when IntelSwitch switches the video segment protocol from QUIC to TCP or vice versa, audio is sent over the other protocol. This dual-path approach ensures that both transport paths are persistently available, allowing seamless switching of video segments between protocols without requiring costly connection setups during playback.

4. Bridging the visibility gap in network metrics: As the browser operates at the application layer without direct access to low-level network statistics, we extract indirect but meaningful signals from dash.js. These include throughput estimates, buffer occupancy, rebuffering events, and bitrate history. The decision engine is designed to process these noisy, incomplete measurements intelligently, providing robust transport protocol decisions despite limited visibility into the true network state.

5. Seamless integration with browser architecture: IntelSwitch modifies Chromium Browser to accept a custom protocol field from the DASH player and creates network requests over TCP or QUIC accordingly. By carefully modifying the network stack, we

avoid disrupting default behavior and prevent concurrency or stability issues, ensuring correct and efficient request handling throughout the session.

6. Minimal deployment hurdle: By implementing all changes at the client side, i.e., within dash.js and Chromium, IntelSwitch avoids requiring any modifications to video streaming servers. This makes the solution practical for real-world deployment across diverse network environments and devices.

Overall, IntelSwitch combines adaptive learning, careful engineering of protocol handling, and smart metric usage to address the core challenges of adaptive video streaming over fluctuating networks. Below, we first discuss our protocol decision engine, followed by the implementation stack of the adaptive protocol switching system by modifying the DASH player and Chromium browser.

5.5.2 Protocol Decision Engine

We design *Protocol Decision Engine*, a Deep Reinforcement Learning (DRL) based adaptive transport protocol selection mechanism. This is agnostic to the underlying video player or the browser, which works on the decision given by the engine.

5.5.2.1 Need for DRL-based protocol decision engine

The performance of video streaming applications is influenced by a wide range of factors, including variable network conditions, video parameters, and the choice of underlying transport protocol. These factors interact in complex ways, making video performance highly unpredictable. A fixed, rule-based decision engine struggles to capture this variability, as it cannot adapt to all possible scenarios. Prior work [44] has explored supervised learning–based decision engines, which rely on labeled data to indicate the preferred protocol under given network conditions. However, such labeling depends on prior knowledge or arbitrary thresholds, limiting both accuracy and scalability. Furthermore, supervised models can only generalize to conditions similar to their training data and often fail in highly dynamic or unseen environments. Reinforcement learning does not rely on predefined labels. Instead, it learns through direct interaction with the environment by taking actions (e.g., choosing TCP or QUIC) and observing the corresponding reward (QoE, stability, or resource efficiency). Therefore, unlike prior works, we adopt deep reinforcement

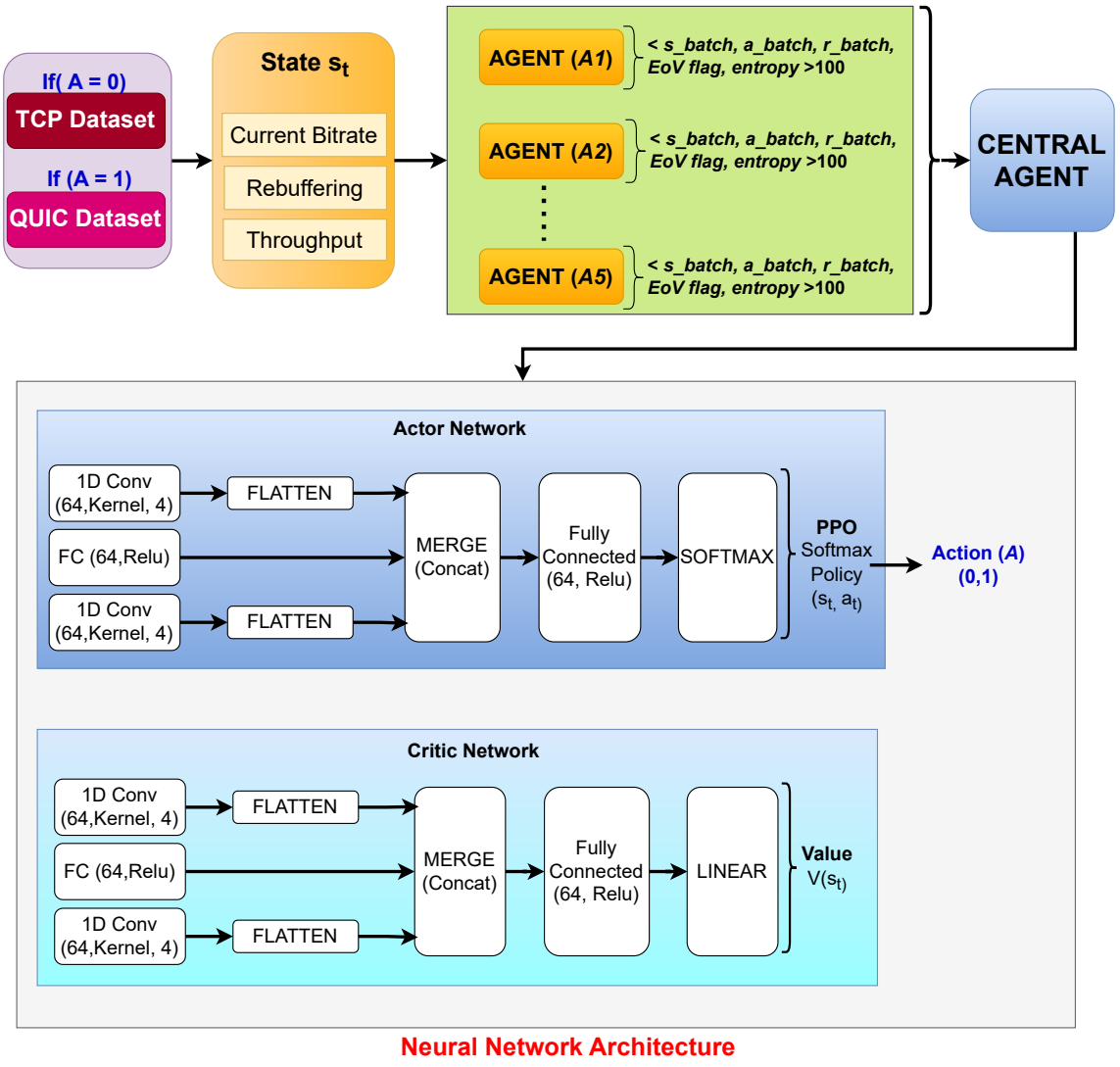


Figure 5.4: Protocol Decision Engine

learning for protocol selection instead of supervised learning. While supervised learning may achieve comparable performance in certain scenarios, it requires pre-labeled data and struggles to generalize across diverse environments. In contrast, reinforcement learning eliminates the need for explicit labeling and offers a higher likelihood of selecting the appropriate protocol under previously unseen or dynamic conditions. Next, we discuss how our protocol decision engine is designed using DRL.

5.5.2.2 Problem Formulation

At each time step t, the decision engine predicts the most suitable transport protocol for a client streaming video in a given network environment. The decision depends on both (1) network parameters, such as throughput, and (2) QoE-related parameters, including current bitrate, bitrate variation, and rebuffering events. Thus, the decision process requires continuous feedback evaluation and adaptive action selection.

Given the stochastic nature of Internet traffic, we formulate protocol selection as a Markov Decision Process (MDP), where an agent interacts with the environment, selects actions, and receives rewards. A finite MDP is defined by a tuple (S, A, P, R), where: (1) S is the state space (all possible environment states), (2) A is the set of discrete actions, (3) P defines the state transition probabilities $P(s'|s, a)$, and (4) R is the reward function mapping state-action pairs to scalar rewards. In our case, the agent observes a state vector at each time t, representing network and QoE conditions, and selects an action choosing between HTTP/2 or HTTP/3 based on action probabilities. We solve this MDP using an actor–critic network(A3C). The actor-critic reinforcement learning architecture is well-suited for dynamic sequential environments with delayed rewards. In A3C, the actor network learns the transport-selection policy, while the critic network estimates long-term value functions that stabilize policy learning and guide future decisions. The asynchronous training mechanism further improves exploration and training stability across diverse network conditions by allowing multiple agents to learn in parallel from different network experiences. Such asynchronous parallel learning is particularly useful for adaptive video streaming applications where different users may simultaneously experience highly heterogeneous network conditions. Note that the RL-based ABR algorithm Pensieve [53] also used an A3C network for bitrate-selection. In adaptive video streaming, the system's behavior is tightly coupled to the choice of transport protocol. For instance, selecting QUIC over TCP can result in different throughput, latency, and buffer dynamics. To adapt effectively to such changes, PPO [75] is particularly well-suited, as it is an on-policy, policy-gradient reinforcement learning algorithm designed to improve stability and sample efficiency while ensuring reliable policy updates. Thus, we replace vanilla A3C's update rule with PPO as it provides more stable and efficient policy updates while retaining the same architecture. This allows us to leverage the advantages of Pensieve's design while benefiting from PPO's stability.

Table 5.1: Protocol decision engine states information

Abbreviation	Definition
CB (Current Bitrate)	Bitrate of the most recently played video segment.
Rebuf (Rebuffering)	Duration of the most recent rebuffering event.
T (Throughput)	Measured throughput experienced by the application.

5.5.2.3 Decision Engine Offline Training

To the best of our knowledge, no existing dataset captures video streaming over both TCP and QUIC across diverse conditions, including high-bandwidth environments such as 5G mid-band, mmWave, and Ethernet. To address this, we collected a dataset spanning: 1 Gbps Ethernet, 5G-mmWave, 5G mid-band (walking and driving scenarios, from [72]), 4G network walking trace from [73], and a poor network from [53] and simulated poor-bandwidth with 5% packet loss and 50 ms delay emulated using `tc netem` [74]. For each setting, we recorded sessions using both TCP and QUIC. This dataset serves as the basis for offline training of our decision engine. Fig 5.4 shows the overall flow of the reinforcement learning (RL)-based protocol decision engine with the collected dataset. In our RL-based protocol decision engine for video streaming, we carefully select the most relevant state features. Table 5.1 summarizes the chosen state features.

Input State: At each time step t, the agent extracts the state vector $s_t = \langle \vec{\mathbf{CB}}, \vec{\mathbf{Rebuf}}, \vec{\mathbf{T}} \rangle$ of a segment k from the collected TCP and QUIC dataset.

Action Space: The action space A is the best protocol for the next segments for each client. The decision engine policy π maps the state to a discrete action $A = \{0, 1\}$, where $0 = \text{HTTP/2}$ and $1 = \text{HTTP/3}$. The policy π maps s_t to an action.

Reward: The reward is defined as a function of the segment's current bitrate, the rebuffering duration, and the throughput experienced at time t. The R_t is defined as:

$$\begin{aligned} R_t &= b_t - r_t \\ b_t &= \log_2(CB_t) \\ r_t &= \log_2(1 + (\beta \cdot \text{Rebuf}_t)) \end{aligned} \tag{5.1}$$

Here, b_t denotes the bitrate of the segment at time t, while r_t represents the rebuffering experienced during the download of that segment of bitrate b. Rebuffering is penalized with β set to the maximum available bitrate (Mbps). We use logarithmic scaling to balance the

reward magnitudes across bandwidth extremes, preventing bias towards high-throughput scenarios (e.g., Ethernet) and enabling consistent learning across heterogeneous conditions.

5.5.2.4 Neural Network Architecture

We employ an A3C framework that uses multiple agents running in parallel to explore the environment and update a shared global model asynchronously. In our implementation, we define separate neural networks for the actor and the critic, both taking the same input state vector: The actor outputs the probability distribution over available protocols, while the critic estimates the expected reward. In our case, both networks share a hybrid architecture, combining fully connected (FC) layers and one-dimensional convolutional (Conv1D) layers. Each input state has 3 feature channels where (1) *Scalar feature* (e.g., rebuffering) → FC layers (64 neurons, ReLU). (2) *Temporal features* (e.g., bitrate, throughput histories) → Conv1D layers (64 filters, kernel size = 4). The outputs are concatenated into one vector and passed to another fully connected layer. The actor takes the output of this fully connected layer as input to the softmax layer and outputs action probabilities. The critic produces a scalar value $V(s_t)$, representing the estimated value of the current state. Once an action is selected, the agent computes the reward. Then it sends the `<state, action, reward>` tuple to a central agent.

The overall procedure for transport protocol switching is summarized in Algorithm 1, and the mathematical description of the algorithm is discussed in the next subsection.

5.5.2.5 PPO-Based Protocol Decision Engine Training:

To enhance and speed up the training, we use multiple agents in our A3C architecture. At each timestamp t, each agent interacts with the environment and collects trajectories, from which policy and value gradients are computed by following the procedure below:

Collection of Trajectories Each agent periodically sends the collected trajectories to the central agent in the form of batches $(s_t, a_t, r_t, e_v, e_r)$, where s_t denotes the observed states, a_t the chosen actions, r_t the corresponding rewards, e_v an end-of-video indicator, and e_r the entropy record.

Algorithm 1 PPO-Based Protocol Decision Engine

Initialize: Actor network π_θ, Critic network V_ϕ, entropy weight $\beta \leftarrow 0.001$
Set learning rates $\alpha_\theta = 0.0003$, $\alpha_\phi = 0.001$
Set epoch counter $k \leftarrow 0$ and training batch = 100
while training not converged **do**
    Step 1: Collect Trajectories
    for each agent i in parallel **do**
        Run policy π_θ in environment
        for each timestep t **do**
            Observe state s_t of TCP and QUIC
            Sample action $a_t \sim \pi_\theta(a|s_t)$
            Receive reward r_t
            Store tuple $(s_t, a_t, r_t, e_v, e_r)$
        Send trajectory $\tau_i = \{(s_t, a_t, r_t, e_v, e_r)\}$ to central agent
    Step 2: Compute Returns and Advantages
    for each trajectory in batch $D_k = \{\tau_i\}$ **do**
        for each timestep t (in reverse) **do**
            Compute TD-error: $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$
            Compute GAE: $A_t = \delta_t + \gamma\lambda A_{t+1}$
            Compute return: $\hat{R}_t = A_t + V(s_t)$
    Step 3: Optimize Policy and Value Network
    for each timestep t **do**
        Compute probability ratio: $r_t = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$
        Compute clipped policy objective L_{clip} using r_t and advantage $\hat{A}_t$
        Compute entropy $H(\pi_\theta)$ from action distribution
        Actor loss: $L_{\text{actor}} = -(L_{\text{clip}} + \beta \cdot H(\pi_\theta))$
        Critic loss: $L_{\text{critic}} = (V_\phi(s_t) - \hat{R}_t)^2$
    Step 4: Update Network Parameters
    Update actor: $\theta \leftarrow \theta - \alpha_\theta \nabla_\theta L_{\text{actor}}$
    Update critic: $\phi \leftarrow \phi - \alpha_\phi \nabla_\phi L_{\text{critic}}$
    Update each agent with the updated actor and critic
    Step 5: Logging
    Log average reward, TD loss, and entropy
    Increment epoch: $k \leftarrow k + 1$

Cumulative Return and Advantage Computation Upon receiving trajectory batches $\tau = (s_0, a_0, r_0, e_{v0}, e_{r0} \ldots)$ from all agents, the central agent computes the cumulative returns as:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{2}$$

where δ_t is the temporal difference error. While δ_t provides a one-step estimate of advantage, relying solely on single-step temporal difference error introduces bias, whereas using full Monte Carlo returns can lead to high variance. Hence, we employ Generalized Advantage Estimation (GAE), which introduces a tunable trade-off between bias and variance. The above computed δ_t is used in GAE advantage computation. Here, γ is the discount factor controlling how much future rewards influence current estimates. $V(s_t)$ is the critic predicted network value and $V(s_{t+1})$ is the predicted value sequence forward by one timestep. $V(s_{t+1})$ is 0 if the trajectory terminated. We recursively compute the GAE advantage A_t, which balances bias and variance using the parameter λ. While the discount factor γ reduces the influence of distant rewards, λ decides how much of the future advantage should be included. To apply this recursion, the computation begins from the last timestep of the trajectory and proceeds backward through time. This balance allows GAE to produce smoother and more reliable estimates.

$$A_t = \delta_t + \gamma\lambda A_{t+1} \tag{3}$$

The critic's final return target, R_t is obtained by combining the estimated advantage with the predicted network value.

$$\hat{R}_t = A_t + V(s_t) \tag{4}$$

Optimization of Policy and Value Network The estimated advantages obtained via GAE serve as inputs to the PPO update rule. PPO uses a clipped surrogate objective to stabilize training by preventing large policy updates. The clipped surrogate objective is defined as:

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min \left(R_t(\theta)\hat{A}_t,\ \text{clip}(R_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right]. \tag{4}$$

where $R_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the ratio of the probability of selecting an action in the new policy to the old policy. $\hat{A}_t$ is the estimated advantage at time step t, and ϵ is the clipping parameter. This objective compares two terms inside the minimum function. The first term, $R_t(\theta)\hat{A}_t$, is the standard policy gradient update that encourages actions with positive advantage and discourages actions with negative advantage. The second term $\text{clip}((R_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)$ prevents the policy from changing too aggressively in a single update. If the new policy moderately increases the probability of an action,

PPO permits the update. However, if the increase is too large, PPO clips the update to prevent overfitting and ensure training stability. The actor is trained using PPO's clipped surrogate loss with $\epsilon = 0.2$, which constrains the policy update by clipping the probability ratio to the range $[1-\epsilon, 1+\epsilon]$, ensuring stable updates and preventing destructive policy shifts. We further enhance the clipped surrogate objective by adding an entropy bonus. The purpose of this term is to encourage the agent to maintain sufficient exploration during training, rather than becoming stuck with a single action. Therefore, the entropy of the softmax output is computed as:

$$\mathcal{H}(\pi_\theta) = -\sum_a \pi_\theta(a \mid s_t) \log \pi_\theta(a \mid s_t). \tag{5}$$

where the summation runs over all possible actions a. Here, $\pi_\theta(a \mid s_t)$ denotes the probability of selecting action a under the current policy π with parameters θ and the negative sign ensures that entropy is non-negative. Further, combining the clipped surrogate objective with the entropy regularization term yields the final actor loss. Therefore, the total objective optimized by the actor becomes:

$$\mathcal{L}_{\text{actor}} = -(L^{\text{CLIP}}(\theta) + \beta \cdot \mathbb{E}_t \left[\mathcal{H}(\pi_\theta)\right]) \tag{6}$$

While the actor is updated to improve the policy and encourages exploration, the critic is updated to minimize the prediction error of state values. Specifically, we use AdamOptimizer to minimize both actor and critic losses. For the actor, it minimizes the negative PPO objective plus entropy, which helps to improve the policy while encouraging exploration. For the critic, it minimizes the mean squared error between the estimated return and the critic's value, helping it predict returns more accurately. The critic network is updated by minimizing the mean squared error (MSE) between the predicted value $V_\phi(s_t)$ and the GAE return target $\hat{R}_t$. The MSE is averaged over the batch to ensure stable gradients during training.

$$\mathcal{L}_{\text{critic}} = \mathbb{E}_t \left[\left(V_\phi(s_t) - \hat{R}_t \right)^2 \right] \tag{7}$$

Updation of Network Parameters After computing the actor and critic losses, the corresponding gradients are sent to the central agent. The central agent aggregates gradients across agents and applies updates to the shared global actor and critic networks. These

gradients are then applied sequentially to the shared global networks. Mathematically, for each agent i , the actor parameters θ and critic parameters ϕ are updated as:

$$\theta \leftarrow \theta - \alpha_\theta \nabla_\theta \mathcal{L}^i_{\text{actor}} \tag{8}$$

$$\phi \leftarrow \phi - \alpha_\phi \nabla_\phi \mathcal{L}^i_{\text{critic}} \tag{9}$$

where α_θ and α_ϕ are the actor and critic learning rates. Moreover, $\mathcal{L}^i_{\text{actor}}$ is the gradient of the actor loss with respect to the actor's parameters θ, and $\mathcal{L}^i_{\text{critic}}$ is the gradient of the critic loss with respect to the critic's parameters ϕ.

5.5.2.6 Offline Training Justification

We rely on offline training using real-world measurements instead of live training for several reasons. First, concurrent multi-agent training in real systems (e.g., high-bandwidth setups) would artificially constrain bandwidth, distorting protocol behavior. Second, live training requires substantial infrastructure for traffic generation, QoE computation, and real-time decision making, making it computationally infeasible. Offline training avoids these issues while still capturing protocol dynamics across diverse environments.

5.5.3 Implementation Stack for Adaptive Transport Selection

We now present the modified video player and the browser that respects the decision chosen by the engine and appropriately forms the media request (video and audio) to be sent to the video streaming server. As a video player, we use `dash.js`, an open-source MPEG-DASH player available on GitHub[1]. We modify the Chromium browser, an open-source web browser maintained by the Chromium Project community, whose complete source code is publicly accessible[2].

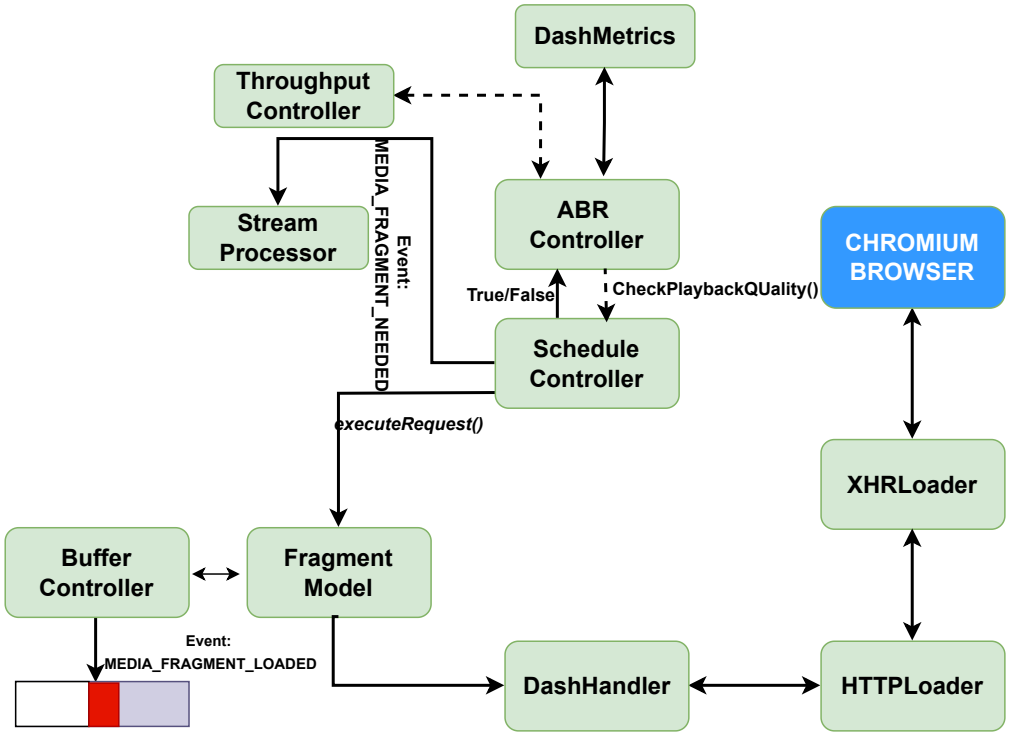


FIGURE 5.5: DASH Workflow

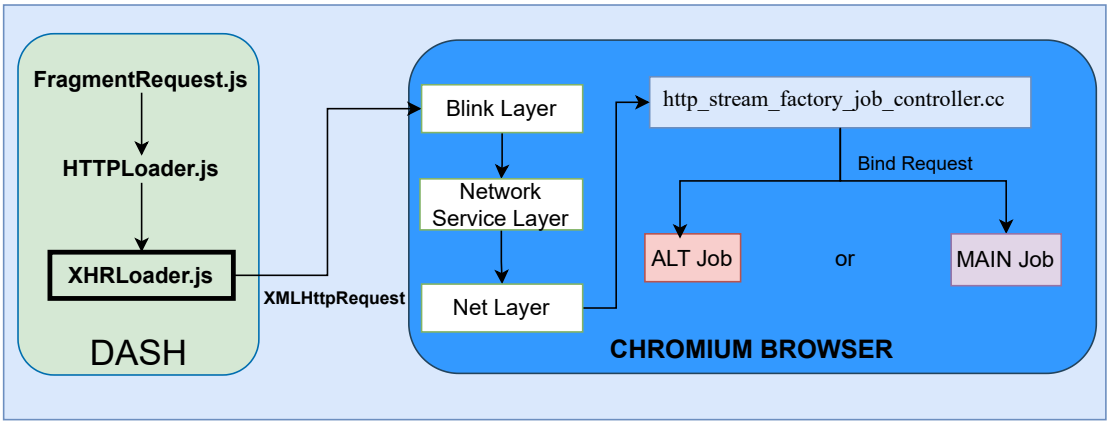


FIGURE 5.6: Chromium Media Request WorkFlow

5.5.3.1 Background

DASH player: Dynamic Adaptive Streaming over HTTP (DASH) is a standard for delivering video in segments of varying quality, allowing playback to adapt to changing network conditions. The server stores video segments at multiple bitrates, while the client's DASH player requests segments at the most suitable quality.
Fig. 5.5 illustrates the request flow. The *ScheduleController* queries the *ABRController* to determine whether the segment quality should change. If so, *ABRController* selects the appropriate quality based on throughput, buffer occupancy, and related parameters, and instructs *ScheduleController* to schedule the new segment; otherwise, it requests the current quality. This triggers the MEDIA_FRAGMENT_NEEDED event, prompting *StreamProcessor* to generate a segment request. The request is added to the execution queue by *FragmentModel* and passed to *HTTPLoader*, which uses *XHRLoader* or *FetchLoader* to fetch the segment from the server. Once retrieved, the segment flows back through *FragmentModel* to *BufferController*, which appends it to the media buffer for playback. Importantly, the DASH video player runs on top of a web browser, making browser integration essential for protocol-aware streaming.

Chromium browser: When the player requests a video segment, it initiates the request through the browser, which is responsible for converting it into an actual HTTP request to the target server. Specifically, the DASH player uses XMLHttpRequest (XHR) to initiate these segment requests. XMLHttpRequest is a JavaScript API for creating an HTTP request to send a network request from the browser to the server. Fig. 5.6 shows the workflow of a media request from the DASH player to the browser. The XHR calls are first processed by the browser's rendering engine, known as the *Blink layer*. Inside Blink, the request is transformed into the browser's internal request format. Once constructed, the request is passed to the renderer's loader component, which is responsible for handling resource fetching from the network. The request is then sent through the Chromium *mojom* system, Chromium's interprocess communication (IPC) mechanism, which relays it from the renderer process to the browser process. The browser process then constructs the final network request and selects the appropriate transport protocol: HTTP/2 over TCP, or HTTP/3 over QUIC. The browser manages two types of jobs for this purpose: the *main_job*, which uses TCP, and the *alt_job*, which uses QUIC. Additionally, other alternative protocols, such as SPDY, may also be considered if supported by the server. The

[1]https://github.com/Dash-Industry-Forum/dash.js
[2]https://source.chromium.org/

job binding is protocol-dependent: if the server supports QUIC or another alternate service, the browser binds the request to the *alt_job*; otherwise, it falls back to the *main_job*. The decision logic for this binding is handled by the *http_stream_factory_job_controller.cc* component in Chromium.

5.5.3.2 Modified DASH Player

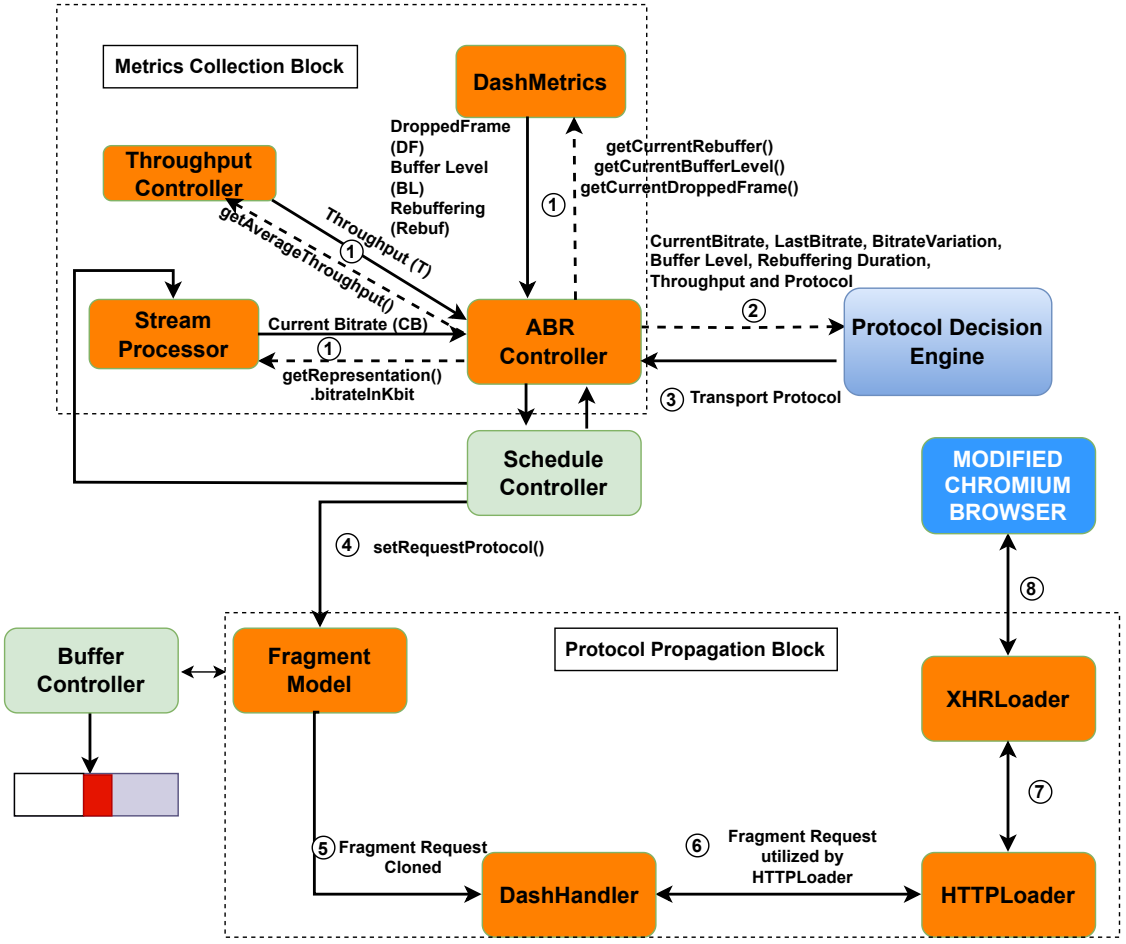


FIGURE 5.7: Workflow of Modified DASH, an orange block indicates the modified code block.

While the standard dash.js player is agnostic to the underlying transport protocol, adapting it for protocol-aware streaming required several non-trivial changes. We first highlight the challenges in extending dash.js to support per-request protocol control and QoE-driven feedback, and then describe our solution design.

Key Challenges

- **Separation of functional layers:** The dash.js architecture cleanly separates metadata parsing (`FragmentRequest.js`), request orchestration (`HTTPLoader.js`), and transport execution (`XHRLoader.js`). Introducing protocol-awareness meant that decisions from the transport selection engine had to be injected consistently across all these layers without breaking modularity. Any inconsistency could cause the protocol intent to be lost before reaching Chromium.
- **Maintaining ABR independence:** Existing Adaptive Bitrate (ABR) logic is central to dash.js and operates orthogonally to the transport layer. Integrating protocol switching without entangling it with bitrate adaptation was essential to preserve stability. A naïve coupling of the two risks could destabilize adaptation decisions.
- **Feedback-loop integration:** The decision engine relies on continuous learning from the network and QoE-level feedback. However, dash.js does not natively expose fine-grained metrics in a structured manner. Capturing sufficient metrics (startup delay, rebuffer events, segment download times, etc.) while ensuring the reporting path remained lightweight required careful instrumentation.
- **Overhead minimization:** Additional metadata propagation and metric reporting could interfere with real-time playback if implemented synchronously or with excessive logging. Ensuring that the modifications did not introduce delays or CPU pressure was a key design constraint.

Solution Design: To address these challenges, we extended dash.js as below:

- **Protocol propagation block:** Protocol intent is embedded as metadata at the point of request creation and propagated end-to-end across `FragmentRequest.js`, `HTTPLoader.js`, and `XHRLoader.js`. This ensures that each segment request carries explicit protocol information that Chromium can interpret deterministically.
- **Metrics Collection Block:** QoE metrics are collected from existing monitoring hooks in dash.js (e.g., buffer events, throughput estimators) with minimal new instrumentation. These are exported asynchronously to the decision engine, preventing reporting from blocking playback or interfering with ABR logic.

Resulting Architecture Fig. 5.7 shows the workflow of modified DASH. It works as follows:

(Step 1) The ABR controller gets the QoE and network metrics from DashMetrics, Stream Processor, and Throughput Controller. **(Step 2)** The DASH player invokes the decision engine prior to issuing each segment request. **(Step 3)** The protocol decision engine responds with the preferred protocol. **(Step 4)** The preferred protocol is attached as metadata to the `FragmentRequest`. **(Steps 5, 6, 7, 8)** This metadata propagates through `DASHHandler`, `HTTPLoader` and `XHRLoader`, ensuring Chromium receives the correct per-request preference.

This architecture enables the DASH player to act as a protocol-aware application, coordinating bitrate adaptation, per-request transport control, and QoE feedback in a modular and non-intrusive manner.

5.5.3.3 Modified Chromium Browser

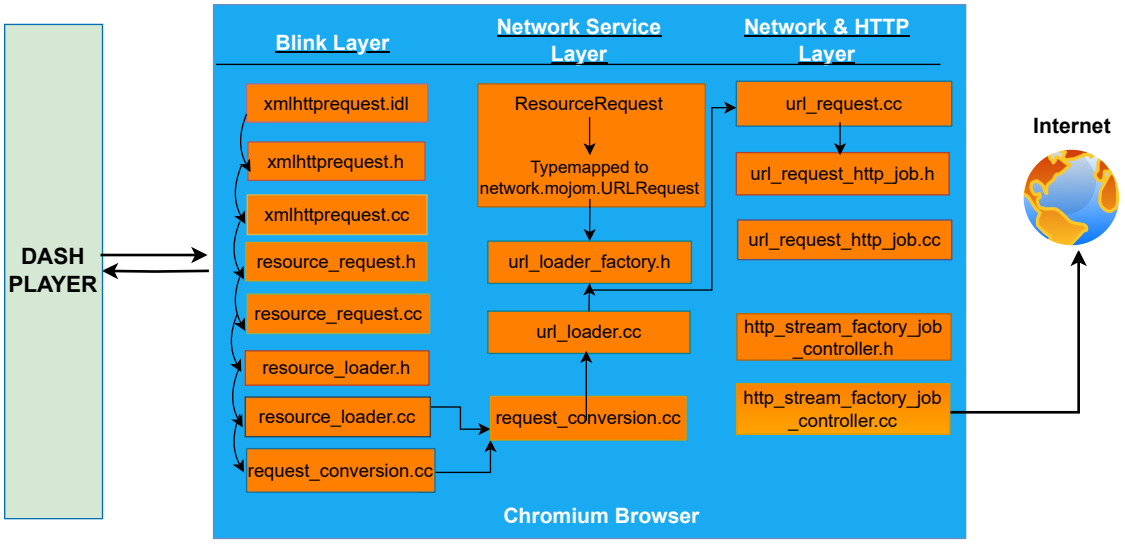


FIGURE 5.8: Workflow of Modified Chromium, an orange bar indicates a modified code block.

Integrating application-driven protocol selection into Chromium was non-trivial, as its networking stack is heavily optimized for opportunistic connection establishment and speculative racing. We first outline the technical hurdles that arise in adapting Chromium for per-request transport control, and then describe our solution design.

Key Challenges Several architectural constraints complicated our modifications.

- **Per-request protocol signaling:** Chromium's default design makes protocol choices at connection setup, using ALPN negotiation or cached Alternative Services. These mechanisms operate at the session level, rather than at the granularity of individual HTTP requests. Associating transport preferences with each DASH segment request, therefore, required introducing new request-level flags, while avoiding concurrency hazards if these flags were stored in global state.
- **Deep cross-layer propagation:** HTTP requests traverse multiple layers: Blink (renderer), the network service (broker), and the net stack (transport). Adding a new field (`preferredProtocol`) required modifications across these layers and propagation through IPC boundaries. Furthermore, this must be accomplished while maintaining stability and backward compatibility with existing request flows.
- **Job orchestration complexity:** The `HttpStreamFactory::JobController` maintains multiple parallel jobs (main, alternate, DNS-based ALPN) and resolves them via speculative racing. The controller aggressively cancels jobs once one succeeds, which is efficient for static protocol choices but incompatible with explicit protocol directives. Preventing premature job cancellation while still leveraging Chromium's fallback paths required restructuring the job orchestration logic.
- **Job binding consistency:** Chromium's assumption of a single bound job per request meant that adding explicit preferences risked leaving jobs unbound or incorrectly reused. Without careful handling, this could create orphaned connections or state inconsistencies in the event-driven machinery of the net stack.

Solution Design To address these challenges, we restructured the request pipeline, enabling explicit and reliable protocol selection.

- **End-to-end visibility of protocol preference:** Each DASH segment request now carries an explicit protocol intent from its origin (the DASH player) through Blink, into the network service, and finally into the net stack. This ensures that the choice of TCP vs. QUIC is treated as primary request metadata, rather than being inferred indirectly through cached hints or ALPN negotiation.
- **Deterministic yet fallback-safe orchestration:** The DASH-provided protocol preference drives connection establishment, replacing speculative racing for the common case. However, Chromium's robust fallback logic (e.g., retrying TCP if QUIC fails) remains intact to preserve reliability in the face of network variability.

- **Minimal intrusion into abstractions:** To reduce regression risks, modifications were confined to interface boundaries between major subsystems (Blink → network service → net stack). This avoided invasive changes inside tightly coupled state machines and preserved Chromium's maintainability across upstream updates.
- **Separation of concerns:** The DASH player provides policy by selecting the transport protocol via its decision engine. Chromium's net stack implements a mechanism for executing this preference within its connection management mechanism. This clear separation prevents policy logic from leaking into low-level networking code, retaining modularity and clarity.

Resulting Architecture Fig. 5.8 shows the modified pipeline. It operates as follows: (Step 1) The DASH player annotates each request with a protocol hint, (Step 2) Blink transparently forwards this metadata, (Step 3) The Network Service Layer carries the hint into the net stack, and (Step 4) Finally, the Network and HTTP Layer (`HttpStreamFactory`) instantiates and binds jobs deterministically in accordance with the preference. (Step 5) Finally, the HTTP request with the selected transport protocol is sent to the server. Through this architecture, protocol selection becomes application-aware, per-request, and explicitly enforced, while retaining Chromium's robust handling of failures and maintaining compatibility with its event-driven connection management system.

5.5.4 Implementation of IntelSwitch

We implement IntelSwitch by modifying the DASH player and Chromium browser. We utilize the DASH player, version 5.0.0, and the Chromium browser, version 126.0.1666.0, for protocol-aware playback.

5.5.4.1 Modifications to DASH Player

The dash.js source code is extended to make protocol-specific requests, collect QoE and network metrics, and finally, it is integrated with the protocol decision engine.

- For making protocol-specific requests: In `FragmentRequest.js`, we add a new field *protocol* to segment metadata (alongside existing fields such as `media_type`,

`quality`, `bandwidth`, and `URL`). In `HTTPLoader.js`, which constructs HTTP requests, we incorporate the protocol field. If no protocol is set, HTTP/3 is used as the default. For audio, the protocol is always set as the alternative of the corresponding video protocol. The modified request is forwarded to `XHRLoader.js`, which issues the actual HTTP request.

- For QoE and network metric collection: We extend `ABRController.js`, specifically the function `_onFragmentLoadProgress`, to invoke a new method `getNextSegmentProtocol`. This method gathers metrics including: Bitrate (from `StreamProcessor.js`), Buffer level, rebuffering time, and dropped frames (from `DashMetrics.js`), Throughput (from `ThroughputController.js`). These metrics are transmitted via asynchronous POST requests to the local decision engine.

- Finally, we integrate the protocol decision engine. The decision engine returns protocol choice (0 = HTTP/2, 1 = HTTP/3). The response updates the protocol field of the in-progress fragment request via `FragmentModel.getRequest`. Then, we update `DashHandler.js` to propagate the modified protocol field before forwarding to `HTTPLoader.js`. Next, the Chromium browser issues the HTTP request using the updated protocol field. This implementation ensures that protocol switching decisions are seamlessly integrated into the DASH workflow while maintaining compatibility with existing modules such as ABR and throughput estimation. Next, we discuss the corresponding modifications in Chromium to support protocol-aware requests.

A total of 8 files were modified, with about 140 lines of code modifications.

5.5.4.2 Modifications to Chromium Browser

We describe the end-to-end flow of how the protocol preference is transmitted from the DASH player to the application net-layer connection establishment in Chromium's network stack.

- **Step 1: DASH Player to Blink (Protocol Injection)** The DASH player issues segment requests via the standard `XMLHttpRequest` API. We extended the Blink layer (rendering engine of Chromium) by: (1) Adding a new `protocol` attribute to `xmlhttprequest.idl`. We modified `xmlhttprequest.h` and `xmlhttprequest.cc`

to read the protocol field associated with each request. We then use this protocol field to set the `protocolMode_`. In mojom, `protocolMode_` is defined as an enum in the network.mojom file. It represents a transport protocol preference for a network request and is serialized across Mojo IPC between the renderer, the browser, and the network service.

- **Step 2: Blink to Network Service (Request Propagation)**
  The `network::ResourceRequest` is a mojo-serializable representation of a network request. The ResourceRequest instance carries the protocol_mode value along with other attributes of a request. It is forwarded from the browser process via Mojo to the network service. We then modified `resource_request.h` and `HttpRequestInfo` to include the `preferredProtocol_` flag, ensuring the protocol preference persists through the out-of-process boundary into the network stack. For `protocolMode_` value *khttp3*, `preferredProtocol_` flag is set to True, and for *khttp2* and *khttp1* it is set to False.

- **Step 3: Net Stack Integration (Protocol Awareness)**
  In `HttpStreamFactory::JobController`, we introduced `activeProtocol` to reference `preferredProtocol_`, allowing job orchestration to access the chosen transport protocol.

- **Step 4: Job Creation Logic (`DoCreateJobs`)** We modified the default job creation behavior to: (1) Always create both `main_job_` (HTTP/2 over TCP) and `alternative_job_` (QUIC). (2) We remove the call to `ClearInappropriateJobs()` to prevent premature job cancellation. We set `main_job_is_blocked_` based on `activeProtocol`: true when HTTP/3 is preferred. We explicitly initialize `alternative_job_` with `kProtoQUIC`. This enables deterministic job creation driven by DASH's protocol decision.

- **Step 5: Job Binding Semantics (`BindJob`)** We extended `BindJob(Job* job)` to enforce protocol consistency. We discard jobs that do not match `activeProtocol` by resetting `bound_job_` and `job_bound_`. We prevent fallback to unintended protocols, ensuring fidelity to DASH's protocol choice.

- **Step 6: Fallback Handling** Our deterministic design preserves Chromium's fallback mechanism: if QUIC fails (e.g., due to a handshake timeout), the HTTP/2 job can still be used, ensuring robustness.

Table 5.2: Protocol Decision Engine Training/Testing parameters

Parameter	**Value**
Discount factor	$\gamma = 0.99$
Advantage estimator	$\lambda = 0.95$
Actor and critic learning rate	$\alpha_\theta = 0.0003$, $\alpha_\phi = 0.001$
PPO clip parameter	$\epsilon = 0.2$
Batch size	$B_s = 100$
Number of epochs	4000
Number of agents	5

A total of 29 files were modified, and these changes amount to about 503 lines of code spread across Blink, Network Service, and Net stack.

5.5.4.3 Implementation Details of Protocol Decision Engine

IntelSwitch neural network architecture was implemented in Python 3.10.12 with TensorFlow 1.15.0 and TFLearn 0.3.2, running on an NVIDIA L40S GPU with CUDA 12.2, cuDNN 8, and driver version 535.247.01. For protocol decision engine training, we write 1146 lines of code. Since training and testing parameters are crucial for optimizing any neural network architecture, the parameters are summarized in Table 5.2.

5.6 Evaluation Setup and Results

In this section, we describe the evaluation setup used to assess the performance of IntelSwitch. We begin by outlining the network traces used for training and testing the adaptive protocol switching model, which cover a wide spectrum of bandwidth patterns, variability levels, and real-world network conditions. Next, we present the baseline approaches against which IntelSwitch is compared, enabling a fair assessment of its performance relative to existing strategies. Finally, we define the evaluation metrics used in our analysis, followed by a discussion of the results.

5.6.1 Experimental Setup:

We selected one sample video (*testsrc2*) from FFmpeg [71], originally 120 seconds long, and extended its duration to 300 seconds. The video is encoded at a wide range of bitrates with 1-second segment durations: 700, 600, 500, 400, 300, 200, 100, 80, 40, 20, 10, 8, 4, 2, and 1 Mbps, along with an additional representation at 500 Kbps. We adopt this high-bitrate encoding and 1-second chunk size based on the motivation from [72], which demonstrates the effectiveness of such settings for 5G networks. The encoded bitrate files are packaged into a DASH manifest using GPAC MP4Box, with the video framerate fixed at 24 fps throughout the process. We develop a custom video streaming client-server setup using OpenLiteSpeed server, which supports both TCP (via HTTP/2) and QUIC (via HTTP/3), utilizing the lsquic implementation of QUIC [76]. The server machine is equipped with an AMD EPYC 7543 32-core CPU and an NVIDIA L40S GPU, running Ubuntu 20.04. The client is a desktop system with an Intel Core i5-9400 6-core CPU, also running Ubuntu 20.04. All video segments, along with the MPD file, are hosted on the server. The client runs a modified dash.js (version 5.0.0) player with a modified Chromium browser (version 126.0.1666.0) for protocol-aware playback. We use BOLA as the underlying ABR algorithm. The RL server is collocated with our video server. We perform both replay and real-time evaluation of IntelSwitch, described as follows:

5.6.1.1 Replay-Based Evaluation

For this, IntelSwitch begins with QUIC as the default protocol for the first segment. For subsequent segments, the protocol is determined by the model's predicted action: if the model selects TCP, the corresponding sample is taken from the TCP dataset; if it selects QUIC, the sample is taken from the QUIC dataset.

5.6.1.2 Real-Time Adaptive Evaluation

For this, we perform live video streaming while emulating network traces using Mahimahi's `linkshell`. The DASH player issues segment requests for specific video segments and simultaneously reports metrics such as network throughput, rebuffering duration, and downloaded bitrate to the IntelSwitch protocol decision engine. Based on these inputs, the decision engine uses the trained model to determine the next action.

5.6.1.3 Network Traces:

We now discuss the network traces used for training the IntelSwitch protocol decision engine and for testing it in both offline and online setups. We categorize our traces into real traces and controlled network traces. Real traces are obtained from publicly available datasets, whereas controlled traces are generated either by combining the real traces or by using `tc-netem` to introduce packet loss and delay.

Real-world Traces: The real-world traces include high bandwidth traces of 5G mid-band and 5G mmWave datasets from [72, 77] collected across walking and driving scenarios. The aggregate throughput for walking versus driving is 1.6 Gbps versus 3.2 Gbps for 5G mid-band, and 935 Mbps versus 1.1 Gbps for 5G mmWave. For low- and variable-bandwidth traces, we use a low-bandwidth walking trace of 4G [73, 78].

Controlled Traces: We create controlled network traces to evaluate how IntelSwitch enhances QoE under sudden network changes. The following controlled traces are used for this purpose: (1) High->Low (H->L): In this, it starts with around 700 Mbps bandwidth (5G mid-band equivalent) and drops to 40 Mbps and below (4G equivalent). (2) High->Low->High (H->L->H): This trace is an extension of the H->L trace, by restoring bandwidth to the 5G mid-band level after 4G, (3) High->Poor->High (H->P->H): This trace is a further extension of H->L->H, by adding 3% packet loss and 50 ms latency during the low-bandwidth phase, emulating degraded 4G walking conditions using tc netem [74]. (4) WiFi->High->Low->High (W->H->L->H): This trace depicts a scenario in which a device initially connected to WiFi, later due to mobility switches to 5G mmWave via a cellular network. Due to handover events, the device subsequently switches to 4G and later returns to 5G mid-band after some time, when again in a well-connected region.

Table 5.3: Training Network Traces Description

Training Network Traces	**Abbreviation**	**Max-BW (Mbps)**	**Min-BW (Mbps)**
Ethernet (1 Gbps)	Eth	957	928
4G Walking	4G-Walk	10	0.06
Poor & lossy network	Poor	40	2
5G-mid-band	MB-Drive	1227	19
	MB-Walk	217	0
5G-mmWave	MM-Drive	2045	0.12
	MM-Walk	1770	5.43

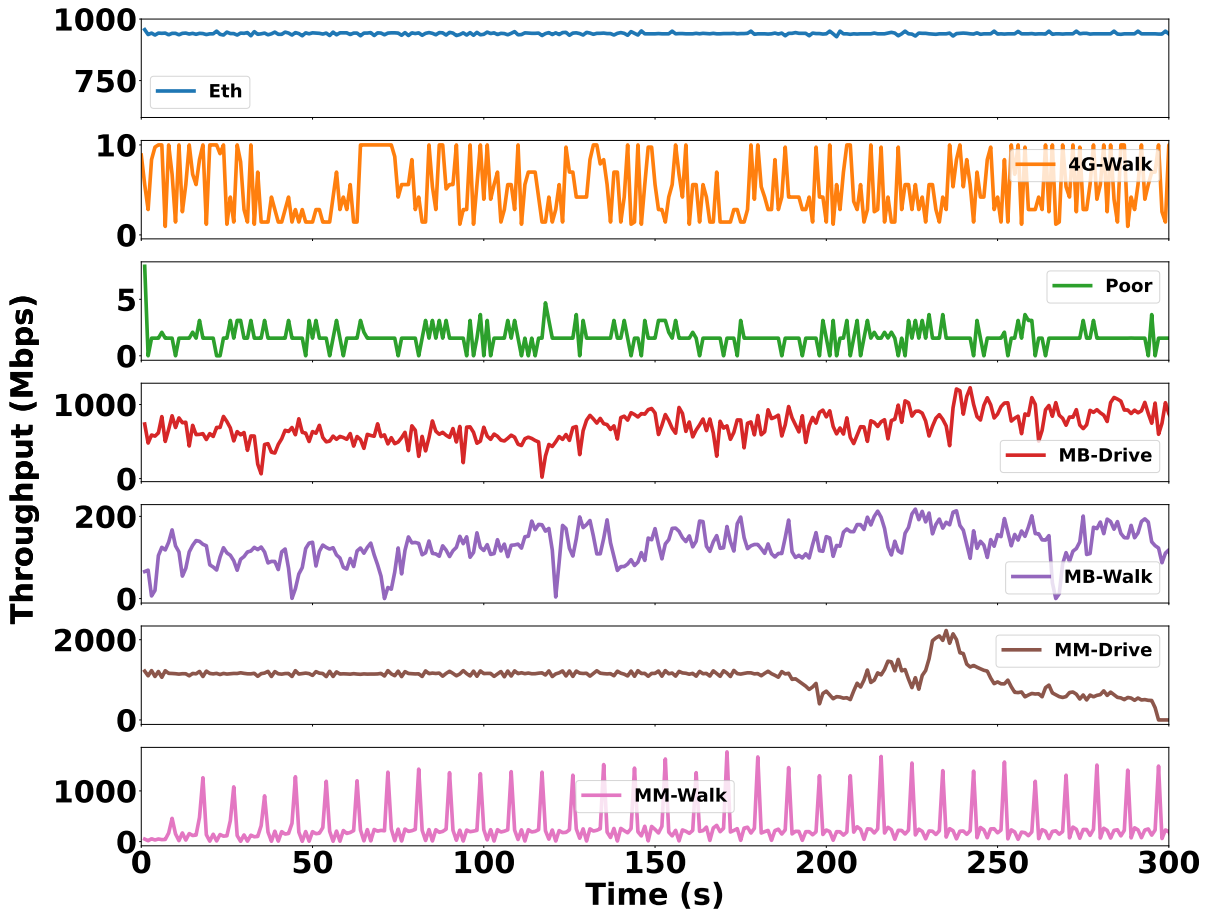


Figure 5.9: Training Network Traces throughput

Table 5.4: Testing Network Traces Description

Testing	Testing Network Traces	Abbreviation	Max-BW (Mbps)	Min-BW (Mbps)
Replay	High-Low	H->L	725	0
	High-low-High	H->L->H	883	17.5
	High-poor-High (3%loss + 50ms delay)	H->P->H	725	0
	WiFi-5G-4G-5G	W->H->L->H	1982	0
Real-time	5G-mid-band	MB-Drive	1267	470
		MB-Walk	1840	0
	5G-mmWave	MM-Drive	2981	0
		MM-Walk	2409	5
	Handover (4G->5G)	HO	1450	0

Training datasets: Fig. 5.9 presents throughput–time plots of our training datasets. We train IntelSwitch's protocol decision engine on real traces. Since the original 5G dataset from [72] is large in size, we partitioned it into separate files, each covering 320 seconds of data. One of the partitioned files was used for training, and the remaining

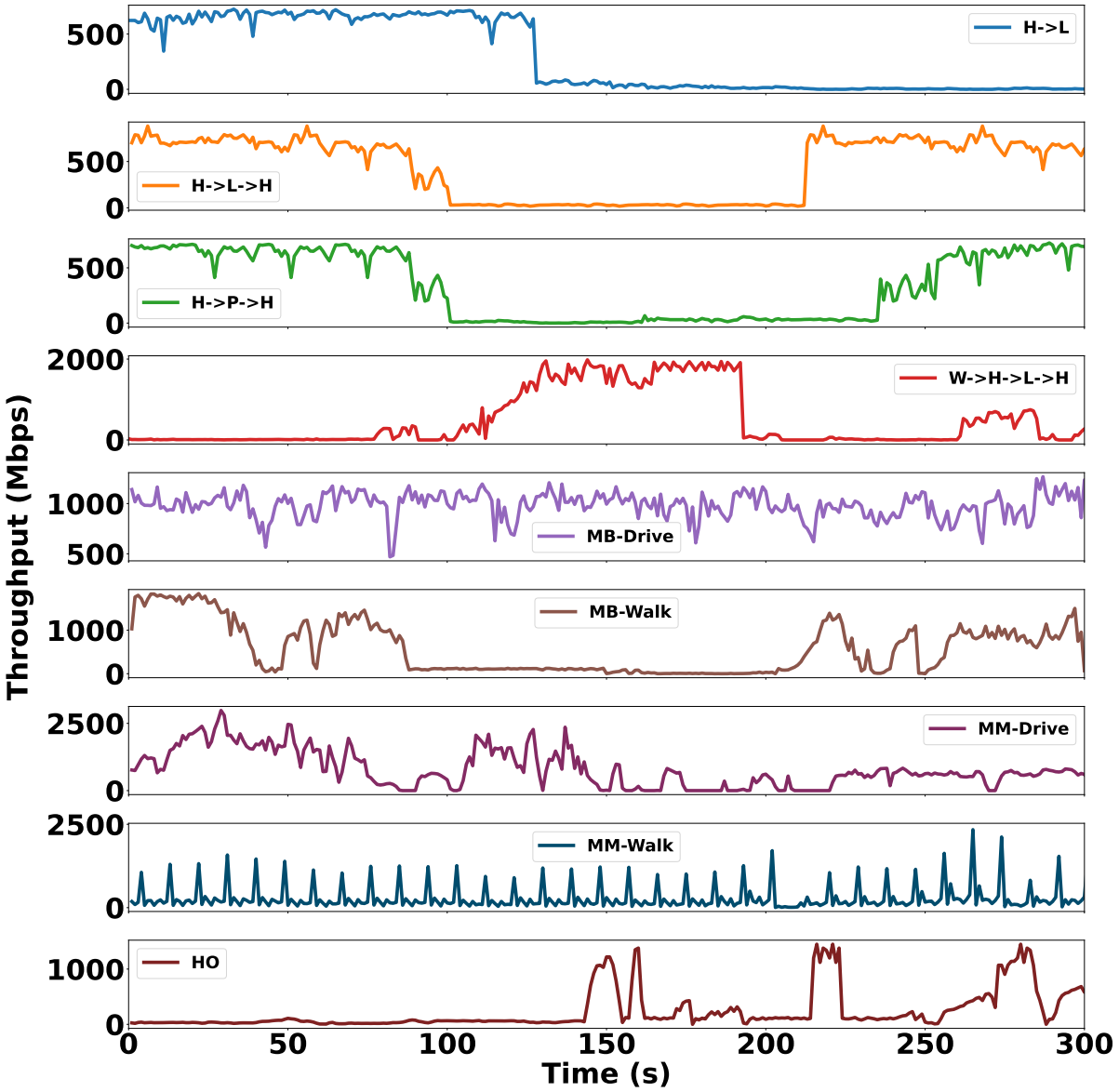


Figure 5.10: Testing Network Traces throughput

files were reserved for testing. We converted each of the real traces into the Mahimahi format. Table 5.3 provides the description of the training traces, which are: MM-Drive, MM-Walk, MB-Drive, MB-Walk, Ethernet, and Poor. The Ethernet trace represents a 1 Gbps link, whereas the Poor trace is emulated using `tc-netem` with a fixed bandwidth of 5 Mbps, 3% packet loss, and 50 ms latency.

Testing methodology: To evaluate IntelSwitch, we employ two complementary approaches: replay-based evaluation and real-time evaluation.

- **Replay-based evaluation:** In this approach, we use data collected from prior streaming sessions conducted over both TCP and QUIC under controlled network conditions. For example, to evaluate a scenario where the network transitions from high to low bandwidth (H $\rightarrow$ L), we first collect two separate traces: one with the video streamed over QUIC and another over TCP for the same network trace. During the replay phase, the decision engine selects the protocol (TCP or QUIC) for each video segment, and retrieves the corresponding segment information from the appropriate trace. This allows us to simulate the adaptive behavior of IntelSwitch in a reproducible and controlled environment while eliminating variability from live network fluctuations.

- **Real-time evaluation:** In this mode, the video is streamed live through a real network environment. The system continuously collects QoE and network metrics, which are fed into the decision engine. Based on the real-time decision, the next video segment request is issued using either TCP or QUIC. This method validates the system's ability to adapt dynamically and react to real-world network conditions in an end-to-end deployment.

Testing Datasets: We use 5G real traces (MM-Drive, MM-Walk, MB-Drive, and MB-Walk) not used for training, along with the controlled traces discussed earlier. Table 5.4 provides the description of the testing traces. Fig. 5.10 presents the throughput–time plots of them.

5.6.1.4 Baselines and ABR Algorithm:

We compare IntelSwitch with the following baselines:

1. **Static Protocols:** We use fixed transport protocols, namely TCP and QUIC, throughout the video session and compare their performance with IntelSwitch, which dynamically switches the transport protocol based on real-time network conditions.

2. **Heuristic Solution:** We use a heuristic solution of protocol switching. Specifically, if the application throughput is higher than $250Mbps$, we choose TCP as the transport protocol, else we choose QUIC.

3. **Supervised learning-based Solution:** We use WiseTrans [44], a machine-learning-based transport protocol selection framework, as a baseline. Originally, WiseTrans was designed for adaptive protocol selection for web page loading using a supervised learning model that selects between TCP and QUIC based on observed network characteristics. To enable a fair comparison in the context of adaptive video streaming, we reimplement and adapt WiseTrans for video streaming applications using the video streaming dataset collected for IntelSwitch. The original WiseTrans employs XGBoost, a tree-based classifier, trained offline using network-level features such as throughput, RTT, packet loss rate, and Time to First Byte (TTFB). In our implementation, we retain the same XGBoost-based learning framework but replace the original web-oriented features with video-streaming-specific segment-level features, including current bitrate, last bitrate, rebuffering duration, throughput, bitrate variation, dropped frames, and latency. Following the methodology of WiseTrans, the training samples are labeled according to the comparative performance of TCP and QUIC, with the better-performing protocol assigned as the target class. The details of the implementation can be found at the GitHub repo [3]

We use BOLA [79] as the underlying ABR algorithm. We have used CUBIC as the underlying congestion control algorithm for both TCP and QUIC at the server end.

5.6.1.5 Quality of Experience Metrics

We use a popular video streaming metric used by MPC [80] and Pensieve [53], which is defined as:

$$QoE = B_t - \gamma * R_t - \beta * BV_t \tag{5.2}$$

where B_i is the bitrate of the segment requested at time t, R_t is the rebuffering time experienced at time t that results from downloading $segment_i$ at bitrate B, which is penalised by the maximum encoded bitrate (700Mbps) and BV_i is the bitrate variation from $segment_{i-1}$ to $segment_i$ experienced at time t and penalised with $\beta = 1$. In addition to combined QoE, we also show individual QoE metrics. For evaluation, we report the mean values computed over 10 independent runs.

[3] `https://github.com/sapna2504/IntelSwitch-adaptive-protocol-switching/`

5.6.1.6 CPU Pressure

We use CPU pressure to monitor the overhead of our implementation. CPU pressure, as defined in the Linux PSI[81], is the proportion of time during which tasks are stalled while waiting for CPU access, reflecting contention for CPU resources. PSI distinguishes between partial stalls ("some"), when at least one task is stalled, and complete stalls ("full"), when all non-idle tasks are stalled simultaneously. Each of these is further tracked using moving averages over 10, 60, and 300 seconds ($avg10$, $avg60$, and $avg300$), which provide short, medium, and long-term views of CPU contention. We have used $avg60$ for comparison.

5.6.2 Evaluation Results

In this section, we present the evaluation results. First, we perform a benchmarking of IntelSwitch to investigate if any overheads are introduced by IntelSwitch. We then present the evaluation results of offline and online testing for IntelSwitch. Finally, we present microbenchmarking of IntelSwitch to understand its internal workings.

5.6.2.1 Benchmarking of IntelSwitch

We compare the performance obtained by the original DASH & browser (named *Vanilla-DB*) and with IntelSwitch, which incorporates all the modifications described in §. 5.5.1. The evaluation is conducted by streaming video content over a 1 Gbps Ethernet link using TCP as the transport protocol. We chose a high-bandwidth and stable Ethernet link to investigate any overheads introduced in IntelSwitch. We specifically chose TCP because it lacks the additional processing overhead that QUIC introduces. Hence, TCP serves as a suitable baseline for comparing the original and modified versions of Dash and Chromium, excluding protocol-level overhead. We run 10 iterations and plot the QoE distribution. Fig. 5.11 shows that IntelSwitch provides enhanced QoE compared to the Vanilla-DB. Note that in this case, TCP was running all throughout due to the Ethernet link. The improvement in QoE is attributed to the separation of the audio and video channels in IntelSwitch. As discussed in §. 5.5.1, to enable smooth protocol switching, we create separate connections for audio and video and let them run over complementary protocols. Such separation allows the browser to obtain a better QoE compared to the original one.

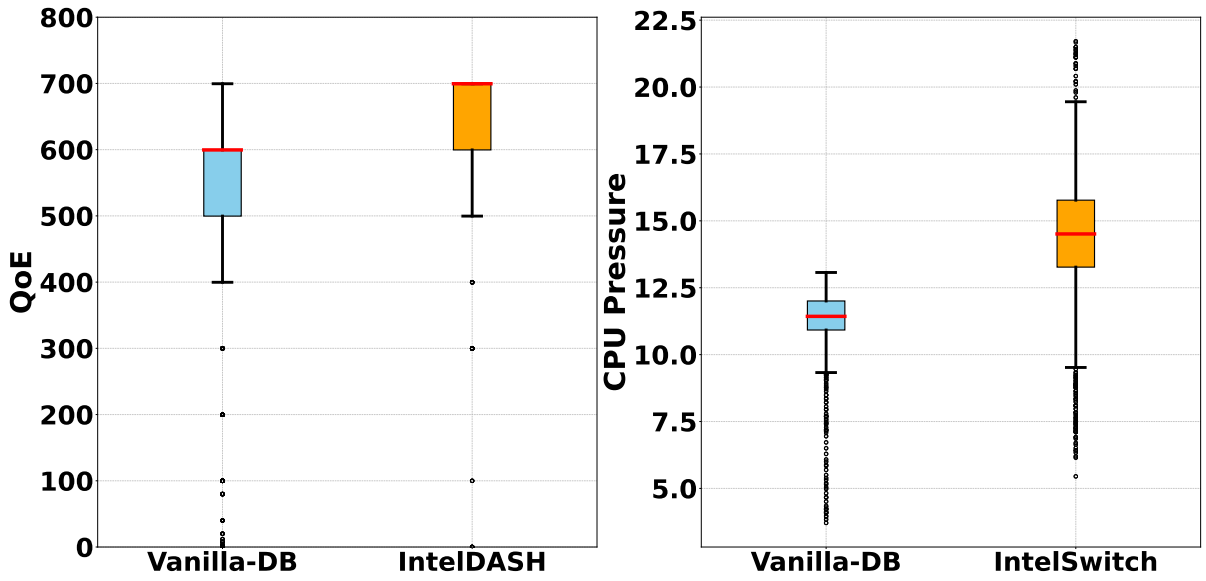


Figure 5.11: IntelSwitch implementation effect on QoE

Next, we compare the CPU pressure of the original configuration with IntelSwitch to investigate the overhead due to the additional computational burden of IntelSwitch. CPU pressure refers to the degree to which the CPU is heavily utilized and under stress. We observe that IntelSwitch streams consistently at a higher bitrate compared to the baseline. While this improves video quality, the elevated bitrate requires additional decoding and data handling, which in turn causes minor increases in the workload on the CPU. As a result, we observe higher CPU pressure increases to about 15 from 12. This demonstrates that our design principle and implementation do not introduce any significant overhead. Furthermore, the minor overhead that does occur is acceptable, as it enables the user to consistently experience higher bitrates.

5.6.2.2 Replay-Based Evaluation of IntelDASH

We first perform replay-based evaluation of IntelSwitch across the network traces discussed in Table 5.4. We use 10 iterations of TCP and QUIC video streaming data collected under the controlled traces, namely H→L, H→L→H, H→P→H, and W→H→L→H. Fig. 5.12 shows a comparison of QoE for IntelSwitch with using only TCP, only QUIC, Heuristic, and WiseTrans. For H→L, IntelSwitch outperforms only TCP, only QUIC, and Heuristic by 43.5%, 114.3%, and 1.7%, respectively and perform similar to WiseTrans. As shown in Fig. 5.10, for almost half the duration, the bandwidth is

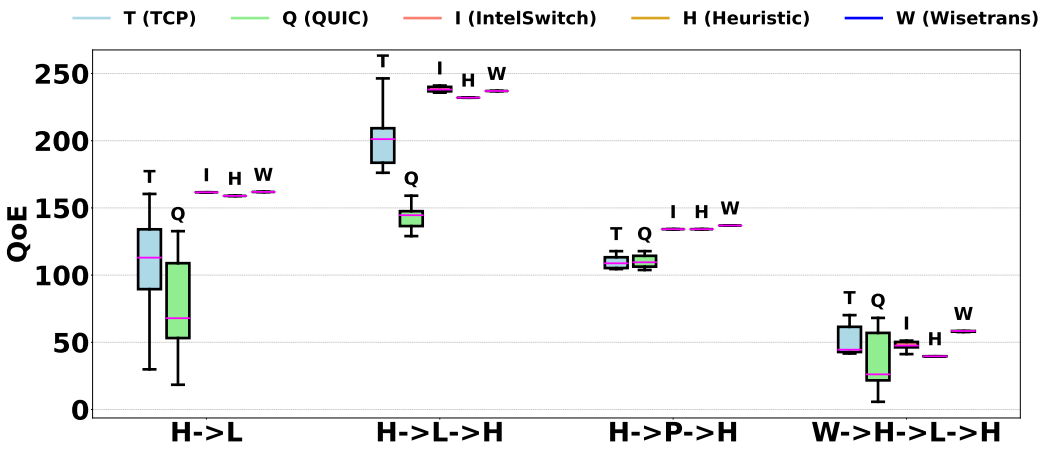


Figure 5.12: Replay-based Evaluation:QoE

high. Thus, TCP provides significantly superior performance than QUIC during high-bandwidth periods. IntelSwitch, WiseTrans, and Heuristic correctly choose TCP for this duration. However, as network bandwidth decreases, TCP's advantage diminishes, and QUIC begins to deliver better QoE. Consequently, IntelSwitch and Heuristic switches to QUIC. In contrast, WiseTrans switches more aggressively between TCP and QUIC, resulting in a higher number of protocol transitions and reduced protocol-selection stability shown in Fig 5.20. Nevertheless, the impact on streaming performance remains limited in this scenario since both TCP and QUIC provide comparable performance under this low-bandwidth network condition. Therefore, for this simple network with lesser variability, all three IntelSwitch, Heuristic, and WiseTrans perform similarly. Thus, adaptively selecting between the two protocols based on prevailing network conditions leads to a significantly higher overall QoE. Fig. 5.20 shows that in this simple case, IntelSwitch has a total of 2 switches: with one at the beginning (starting with QUIC and seeing the high bandwidth IntelSwitch's protocol decision engine decides to switch to TCP), then later at around 150 sec, when the bandwidth drops, IntelSwitch switches to QUIC. Similarly, Heuristic also switches to QUIC when throughput falls below the threshold, i.e., 250Mbps. This indicates that WiseTrans repeatedly alternates between TCP and QUIC instead of maintaining a stable protocol-selection policy, resulting in excessive switching with limited practical benefit. For H→L→H, IntelSwitch outperforms only TCP, only QUIC, and Heuristic by approximately 10.8%, 41.9%, and 2.6%, respectively. Note that IntelSwitch and WiseTrans perform similarly in this scenario as well. The relatively small improvement over TCP and Heuristic is expected, as this scenario (Fig. 5.10) predominantly features high bandwidth, where IntelSwitch performs close to TCP, and Heuristic also chooses to run over TCP. Between 80–100 seconds, the

bandwidth begins to decline and fluctuate. During this period, the throughput falls into a range where both TCP and QUIC provide comparable performance. Unlike Heuristic, which switches once to QUIC, IntelSwitch and WiseTrans alternates between the two protocols in this fluctuating region. This earlier detection of bandwidth changes and more frequent switching enable IntelSwitch and WiseTrans to sustain a higher bitrate for longer, which explains their slight but consistent improvement (about 2%) over Heuristic, while also outperforming TCP-only streaming. In the H→L transition, the change in bandwidth is abrupt, the bandwidth drop is abrupt, leaving little scope for gradual adaptation; hence, all approaches behave similarly in that specific segment. After 100 seconds, when the bandwidth falls to around 17 Mbps, both TCP and QUIC perform comparably, leading IntelSwitch and Heuristic to operate over QUIC. Notably, at several instances during QUIC streaming, the video sustains a higher bitrate compared to TCP-only streaming. Thus, IntelSwitch and Heuristic perform better than either TCP or QUIC streaming alone.

Furthermore, IntelSwitch and Heuristic switches back to TCP again when the network bandwidth increases (e.g., at around 200 sec). But later, towards the end of the video, network throughput goes down, though varying, but is always below 250Mbps, hence Heuristic switches and stays at QUIC. On the other hand, IntelSwitch makes switching as necessary. Thus, Fig. 5.20 shows that IntelSwitch makes about 10 switches (median is 10). In this scenario, IntelSwitch consistently outperforms QUIC, as it is mostly dominated by high bandwidth. The slight improvement of IntelSwitch over Heuristic is also because of a larger number of switches, that is, IntelSwitch switches to the appropriate protocol earlier than Heuristic, whereas Heuristic waits for the 250Mbps threshold to hit. A similar performance trend is observed for WiseTrans. However, unlike IntelSwitch, WiseTrans exhibits significantly more protocol switching, performing as many as 73 protocol switches during the same session. While this excessive switching does not noticeably affect QoE in this particular scenario because TCP and QUIC provide comparable performance for most of the trace, WiseTrans eventually benefits from selecting the same protocol as IntelSwitch towards the end of the session. Nevertheless, the substantially higher number of protocol transitions raises concerns regarding protocol-selection stability and highlights the need to understand whether such frequent switching could negatively impact performance in practical live-streaming deployments.

The H→P→H scenario is an extension of H→L→H, where, during low bandwidth, additional packet losses and delays occur. We found that IntelSwitch outperforms TCP

and QUIC by approximately 8.0% and 10.7%, respectively. Whereas all three IntelSwitch, Heuristic and WiseTrans perform similarly. At around 100 seconds, the network bandwidth decreases and begins to suffer from packet losses and delays up to 150 seconds. Under these conditions, QUIC achieves a higher bitrate than TCP due to its handling of the Head-of-Line (HoL) blocking issue. Consequently, when both IntelSwitch and Heuristic switch to QUIC, they benefit equally and thus exhibit comparable performance. Later, at 250 seconds, the bandwidth increases, and both IntelSwitch and Heuristic switch back to TCP, as Heuristic also detects throughput levels above 250 Mbps at that time. The reduced improvement observed for QUIC alone, compared to the previous case, is due to the added packet losses and delays, under which QUIC performs better than TCP. In this scenario, QUIC shows a relative advantage, whereas TCP experiences a decline in performance. In the case of WiseTrans, we observe fewer protocol switches than in the H→L→H scenario. However, the number of protocol transitions remains substantially higher than that of IntelSwitch and Heuristic, reaching up to 53 switches. Specifically, WiseTrans selects QUIC for a certain duration and subsequently exhibits repeated switching between TCP and QUIC. This behavior indicates a less stable protocol-selection policy compared to IntelSwitch and Heuristic, both of which make more selective and stable switching decisions.

In the W→H→L→H scenario, we found that IntelSwitch outperforms only TCP and only QUIC and Heuristic by 6.17%, 80.70%, and 19.38%, respectively. This trace illustrates the impact of network variability on streaming performance. Initially, the available bandwidth fluctuates between 5 Mbps and 30 Mbps, a regime where TCP and QUIC achieve comparable performance. Around 80 seconds, the connection transitions from WiFi to a 5G mmWave cellular network, and bandwidth gradually increases up to 2 Gbps. During this transition, IntelSwitch does more switching, which makes the transition smoother, as discussed in the above scenario. Here, Heuristic switches protocol only once when it experiences a throughput of more than 250 Mbps. This is the time where IntelSwitch achieves better bitrate earlier than Heuristic. In this region, when network bandwidth is high, TCP provides the greatest benefit by sustaining the highest bitrate, whereas QUIC gets bottlenecked due to processing overhead. Thus, IntelSwitch chooses TCP at this point. When the network later switches from 5G to 4G, the available bandwidth drops sharply, causing both TCP and QUIC to suffer initially; however, QUIC recovers faster and delivers higher bitrates earlier than TCP. Hence, IntelSwitch switches to QUIC. Towards the end of the trace, the network reverts to 5G, with a peak bandwidth of 1 Gbps and noticeable variability. In this regime, TCP

once again outperforms QUIC, and IntelSwitch adapts by switching back to TCP. In contrast, Heuristic gets stuck on QUIC for longer, failing to take advantage of the available high but variable throughput. This delayed transition results in lower QoE for Heuristic, whereas the timely adaptation of IntelSwitch delivers much higher QoE compared to both QUIC-only and Heuristic baselines. In this bandwidth scenario, we observe that WiseTrans achieves approximately 18% higher mean QoE than IntelSwitch. Upon closer inspection, we find that this improvement is primarily driven by a higher average bitrate. However, this advantage must be interpreted in the context of WiseTrans's aggressive protocol-switching behavior. As shown in Fig. 5.20, WiseTrans performs up to 111 protocol switches during a single run, compared to only 10 switches for IntelSwitch. This frequent switching is a consequence of its reactive per-segment decision-making, which repeatedly alternates between TCP and QUIC in response to short-term network fluctuations. The higher bitrate achieved by WiseTrans in the W→H→L→H scenario is primarily due to its opportunistic selection of the TCP trace during the high-bandwidth phase, where the pre-collected TCP throughput reaches up to 600 Mbps. Consequently, WiseTrans is able to exploit these throughput spikes and achieve a higher average bitrate. For example, at segment 182, WiseTrans experiences a sudden bitrate drop from 600 Mbps to 400 Mbps due to aggressive bitrate overshooting. In contrast, IntelSwitch switches protocols only when a sustained performance advantage is observed, resulting in significantly fewer protocol transitions while maintaining comparable bitrate performance across most scenarios. This conservative strategy avoids abrupt bitrate fluctuations and promotes greater streaming stability. These results highlight a fundamental trade-off between the two approaches. WiseTrans greedily optimizes for short-term throughput gains, leading to higher bitrate but also significantly more protocol switching. In contrast, IntelSwitch prioritizes stable long-term performance with minimal switching overhead through selective and adaptive protocol decisions. While the replay-based evaluation favors WiseTrans in this particular scenario, it is important to note that replay-based experiments rely on previously collected protocol traces and select QoE values from the available measurements. In a live streaming environment, buffer occupancy, bitrate adaptation, protocol-switching overhead, and transport behavior interact continuously. Therefore, aggressive protocol switching that appears beneficial in replay-based evaluation may not necessarily translate into superior performance in real-world deployments. We believe that the more stable and selective protocol-switching behavior of IntelSwitch is likely to provide greater benefits during live streaming.

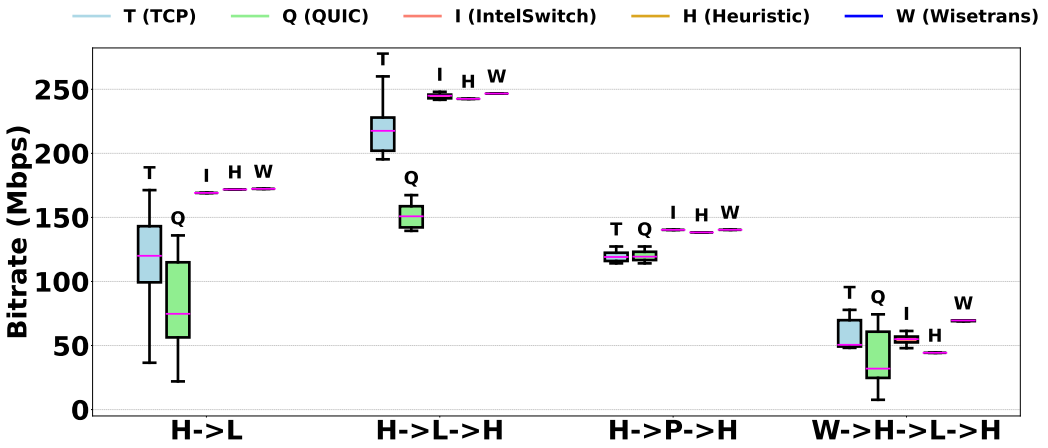


FIGURE 5.13: Replay-based Evaluation: Bitrate (Mbps)

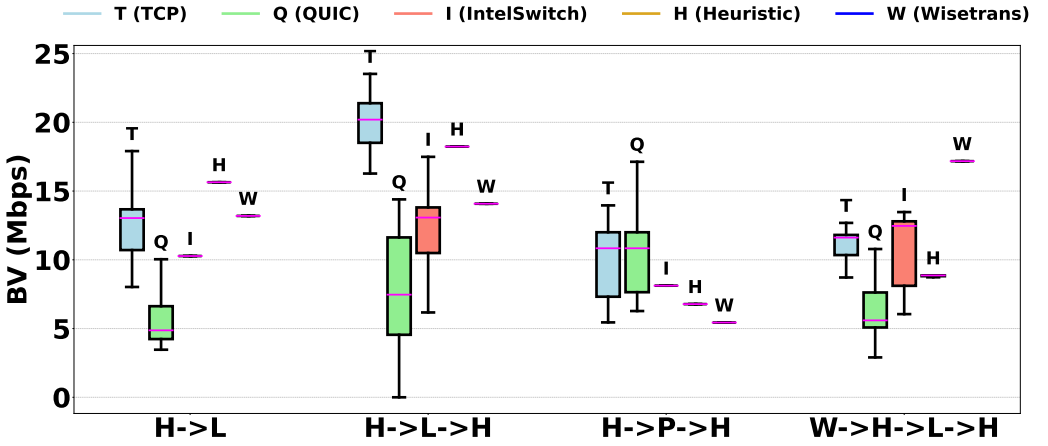


FIGURE 5.14: Replay-based Evaluation: Bitrate Variation (Mbps)

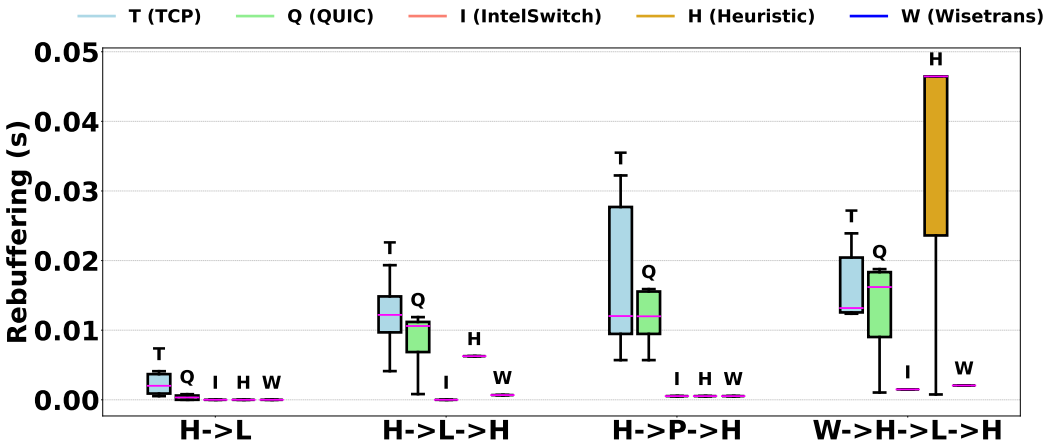


FIGURE 5.15: Replay-based Evaluation: Rebuffering (seconds)

Apart from overall QoE, we also analyze individual QoE parameters, as shown in Fig.5.13, Fig.5.14, and Fig.5.15. The figures show that IntelSwitch achieves a clear advantage in terms of bitrate, streaming video at a higher average bitrate when protocols are used interchangeably. Additionally, we observe that the number of bitrate variation instances and rebuffering instances is lower compared to using only QUIC, only TCP, or Heuristic. While these simple traces allowed us to understand the IntelSwitch's protocol decision choices. We next perform a real-time adaptive evaluation of IntelSwitch to assess its performance in real-time.

5.6.2.3 Real-Time Adaptive Evaluation of IntelSwitch

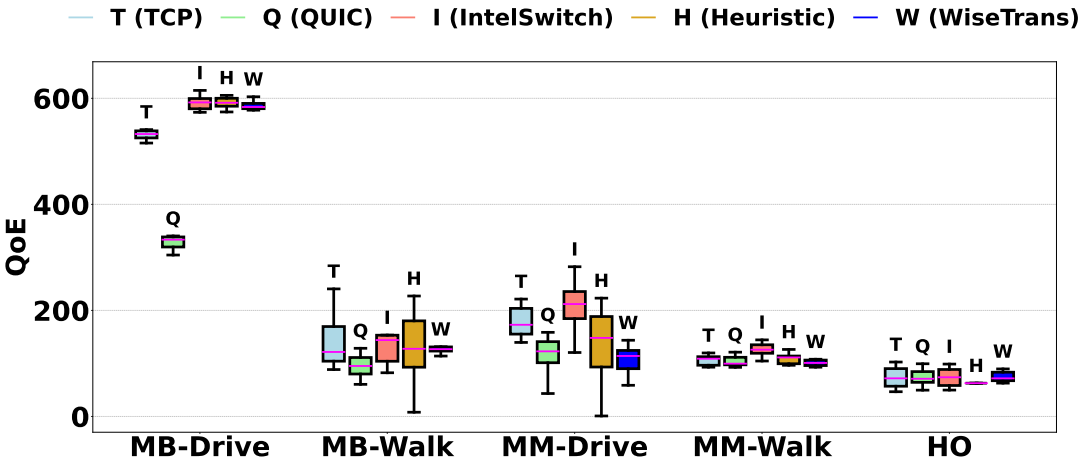


Figure 5.16: Real-time adaptive evaluation: QoE

We now perform a real-time evaluation IntelSwitch on five different real network traces listed in Table 5.4, namely, MB-Drive, MB-Walk, MM-Drive, MM-Walk, and HO. We run 10 iterations for each network type and plot the results. Fig. 5.16 shows a comparison of QoE for IntelSwitch with using only TCP, only QUIC, Heuristic, and WiseTrans. For MB-Drive, IntelSwitch provides significant improvement of 65% compared to using only QUIC and provides similar QoE compared to only TCP, Heuristic, and WiseTrans. Fig. 5.10 shows MB-Drive experiences a consistent throughput of around 1 Gbps with some deviations. As a result, as shown in Fig. 5.20, both IntelSwitch and Heuristic consistently choose TCP throughout (with no switching), resulting in a significant enhancement compared to QUIC only. In case of WiseTrans, we observe 2 switches at the beginning of every run where WiseTrans chooses QUIC for random 2 segments. This does not significantly impact performance, and we observe

that WiseTrans performs similarly to IntelSwitch and Heuristic. IntelSwitch achieves approximately 4% better QoE compared to TCP. This slight improvement is attributed to the decisions made by the ABR algorithm. In the case of MB-Walk, IntelSwitch achieves a QoE improvement of 18.45% over TCP, 51.31% over QUIC, 13% over Heuristic, and 12.5% over WiseTrans. The trace exhibits a lot of variability; the first instance of bandwidth variation occurs from 30 to 60 seconds, and we found that IntelSwitch does relatively more switching compared to Heuristic, which in turn contributes to a smoother transition from TCP to QUIC at this point. Within 10 seconds after the bandwidth drop, the network bandwidth began to increase again, and all three IntelSwitch, Heuristic, and WiseTrans switched to TCP immediately. Later, another drop in bandwidth is experienced at around 75 seconds, where the application throughput experienced by Heuristic was higher than 250 Mbps; therefore, it does not switch, but IntelSwitch detects the bandwidth change and switches to QUIC, then returns to TCP. Around $t = 100$ seconds, the network bandwidth becomes extremely low, occasionally reaching zero; In such scenarios, we found that QUIC recovers more quickly than TCP, thereby providing higher QoE compared to TCP. Since at low bandwidth IntelSwitch and Heuristic both switch to QUIC, therefore both provide equal benefit in terms of QoE. Between $t = 200$ and $t = 250$ seconds, the variability is substantial, leading both protocols to perform similarly. Overall, both TCP and QUIC offer similar performance, with each providing better performance in different areas. Note that in such scenarios, IntelSwitch adapts well and switches more frequently than Heuristic and thus provides a stable QoE compared to Heuristic. Moreover, we observe Heuristic experiences more rebuffering and choose a lower bitrate. The reason for the higher rebuffering is its inability to switch to the appropriate protocol at the right time, caused by slower updates in the estimated throughput values. It is important to note that WiseTrans performs more frequent protocol switching (76) and often alternates between TCP and QUIC in a reactive manner. In contrast, IntelSwitch maintains the selected protocol unless a sustained performance advantage is observed from switching. This allows the ABR algorithm sufficient time to adapt to the characteristics of the underlying transport protocol, resulting in a more stable protocol-selection strategy and avoiding unnecessary protocol switching. Consequently, IntelSwitch is able to achieve higher bitrates and improved QoE under varying network conditions, thereby outperforming WiseTrans.

In the case of MM-Drive, IntelSwitch achieves approximately 72.62% better QoE than QUIC, 22.55% better than TCP, 43.07% better than Heuristic and 46% better than WiseTrans. As Fig. 5.10 shows, MM-Drive has significant variations due to the use of

mmWave, which is very vulnerable to blockages. At the beginning of this trace, the bandwidth is high, reaching up to 2.5 Gbps, during which all three IntelSwitch,Heuristic, and WiseTrans streams are transmitted over TCP. At time t=140 to 145 sec, we observe that Heuristic switches to QUIC as soon as the bandwidth drops below 250 Mbps. In contrast, IntelSwitch does not fully commit to QUIC; instead, it briefly switches to QUIC and then returns to TCP, even when the available throughput is around 155 Mbps. During this transition, TCP sustains a higher bitrate than QUIC. After 150 seconds, the bandwidth remains consistently low, leading both IntelSwitch and Heuristic to continue streaming over QUIC. Overall, in this scenario, IntelSwitch makes about 11 protocol switches compared to 7 in Heuristic. This higher switching frequency in IntelSwitch contributes to smoother transitions and, ultimately, a better QoE compared to Heuristic. We observe that WiseTrans exhibits behavior similar to Heuristic when the available bandwidth decreases. In particular, WiseTrans continues to use QUIC for extended periods under low-bandwidth conditions, whereas IntelSwitch quickly switches to complete TCP when it detects a sustained performance advantage. Furthermore, WiseTrans frequently alternates between TCP and QUIC, resulting in a total of 37 protocol switches during the session. Despite these additional protocol transitions, WiseTrans does not achieve a higher bitrate. The frequent switching prevents the ABR algorithm from effectively adapting to the characteristics of the underlying transport protocol, thereby limiting bitrate improvements. In contrast, IntelSwitch maintains a more stable protocol-selection policy, allowing the ABR algorithm sufficient time to adapt and utilize the selected transport more effectively. We also observe that the excessive protocol switching in WiseTrans negatively impacts buffer growth. When the bandwidth drops significantly between t = 180 and 200 seconds, the frequent switching fails to maintain adequate buffer occupancy, resulting in slightly higher rebuffering compared to IntelSwitch. On the other hand, the more stable and selective protocol decisions made by IntelSwitch enable it to sustain higher buffer levels, avoid unnecessary rebuffering, and deliver better overall streaming performance.

MM-Walk shows an interesting pattern of periodical spikes. IntelSwitch demonstrates improved QoE compared to both static transport protocols and compared to Heuristic and WiseTrans. Specifically, it outperforms only TCP by 19.52% and only QUIC by 26.59% and by 13.1% better than Heuristic and 20% better than WiseTrans. In this scenario, the application achieves an average throughput of less than 250 Mbps, with occasional spikes exceeding this threshold. Under these conditions, Heuristic streams over QUIC, while IntelSwitch prefers TCP. When the bandwidth drops, for example, at

t=230 seconds, IntelSwitch switches to QUIC and achieves better QoE, while already sustaining a higher bitrate than Heuristic at that time. For TCP-only streaming, due to the static protocol, the ABR stabilizes at 100 Mbps. In contrast, protocol switching in IntelSwitch, Heuristic and WiseTrans enables the ABR to stream at a higher bitrate of around 120 Mbps. This results in slightly more rebuffering, but the median rebuffering time remains comparable to the TCP-only case, while delivering a higher overall QoE. Notably, in static protocol scenarios with constant throughput, BOLA stabilizes buffer occupancy and streams at the highest sustainable bitrate below the throughput, resulting in smooth playback with minimal rebuffering and very few switches. With protocol switching, however, the buffer evolves more dynamically as the system adapts to transitions between TCP and QUIC. These dynamic buffer variations reflect the protocol-aware adaptability of IntelSwitch, Heuristic and WiseTrans, enabling sustaining QoE even under fluctuating network conditions. These improvements highlight the effectiveness of adaptive switching in response to varying network conditions. Even though the network bandwidth fluctuates with a periodic pattern, the spikes are momentary. As a result, applications do not perceive this high bandwidth. IntelSwitch also does not 'overreact' to momentary spikes and makes only a minimal number of switches (less than 4). In contrast, the heuristic approach relies on fixed thresholds for protocol switching decisions, and in highly variable networks, it often misses opportunities to improve QoE. IntelSwitch, by capturing these short-lived events, is able to make more effective switching decisions, thereby enhancing the user experience. On the other hand, whenever WiseTrans observes a reduction in throughput, particularly in regions where TCP and QUIC exhibit similar performance, it tends to switch to QUIC and remain there for extended periods. In contrast, IntelSwitch continuously evaluates the long-term impact of its protocol decisions and switches from QUIC to TCP whenever a sustained improvement in application throughput is observed. As a result, IntelSwitch achieves slightly higher throughput and consequently higher bitrates than WiseTrans. These observations indicate that IntelSwitch makes more stable and selective protocol-switching decisions, avoiding unnecessary protocol switching while adapting effectively to changing network conditions. In comparison, WiseTrans relies on more reactive protocol-selection behavior, which may not always translate into improved application performance. Overall, the results suggest that the RL-based decision framework in IntelSwitch is able to learn protocol-selection policies that better capture long-term streaming performance than the supervised-learning-based approach used in WiseTrans.

For the HO scenario, IntelSwitch outperforms only TCP, only QUIC, and Heuristic by 2.8%, 3.9%, and 5.7%, respectively. IntelSwitch and WiseTrans perform similarly in this scenario. This handover scenario from 4G to 5G predominantly experiences a low bandwidth with occasional spikes (Fig. 5.10). At this bandwidth, both TCP and QUIC exhibit comparable performance. The IntelSwitch decision engine correctly selects QUIC, as it typically outperforms TCP under poor network conditions whereas WiseTrans engine alternatively select TCP and QUIC. During high bandwidth, IntelSwitch, Heuristic and WiseTrans correctly choose TCP. However, since periods of high bandwidth are short-lived, the overall performance of IntelSwitch remains comparable to running solely over TCP or solely over QUIC. The advantage of IntelSwitch becomes evident around t = 250 seconds, when the bandwidth drops sharply between high-bandwidth phases shown in Fig. 5.10. In this case, IntelSwitch switches to QUIC to sustain throughput and later returns to TCP as conditions improve. WiseTrans in this case switch to QUIC but also alternatively chose TCP. As shown in Fig. 5.20, IntelSwitch exhibits a median of four protocol switches, compared to two for Heuristic and 33 for WiseTrans. These limited switches of IntelSwitch compared to WiseTrans, triggered by bandwidth spikes, allow faster and stable adaptation to rapid network fluctuations and thereby contribute to improved QoE. Apart from overall QoE, we also analyze individual QoE parameters, as shown in Fig. 5.17, Fig. 5.18, and Fig. 5.19. These results demonstrate that IntelSwitch either consistently outperforms the Heuristic and WiseTrans approach or, in the worst case, achieves comparable performance across all individual QoE metrics. Moreover, IntelSwitch attains a higher or similar bitrate, lower bitrate variation, and nearly identical stall duration compared to both TCP and QUIC across all network scenarios. Next, to gain deeper insights into the behavior and efficiency of IntelSwitch, we conduct a set of microbenchmarking experiments, as discussed in the following section.

5.6.2.4 Microbenchmarking of IntelSwitch

We now perform microbenchmarking of IntelSwitch to understand the internal working of IntelSwitch. We first investigate the convergence of our protocol decision engine training. We then investigate when and how it decides to switch protocol. For the same, we take one of the live testing samples, i.e., we take HO (Hand Over) trace.

Convergence of IntelSwitch Protocol Decision Engine: Fig. 5.21 plots the entropy and reward with the number of epochs. One epoch consists of 100 time-steps. Initially, the

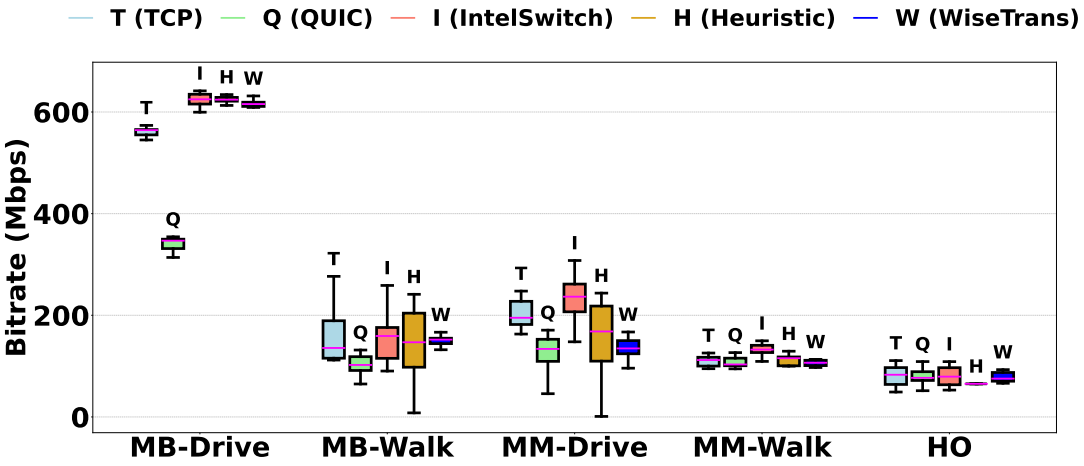


FIGURE 5.17: Real-time adaptive evaluation:Bitrate (Mbps)

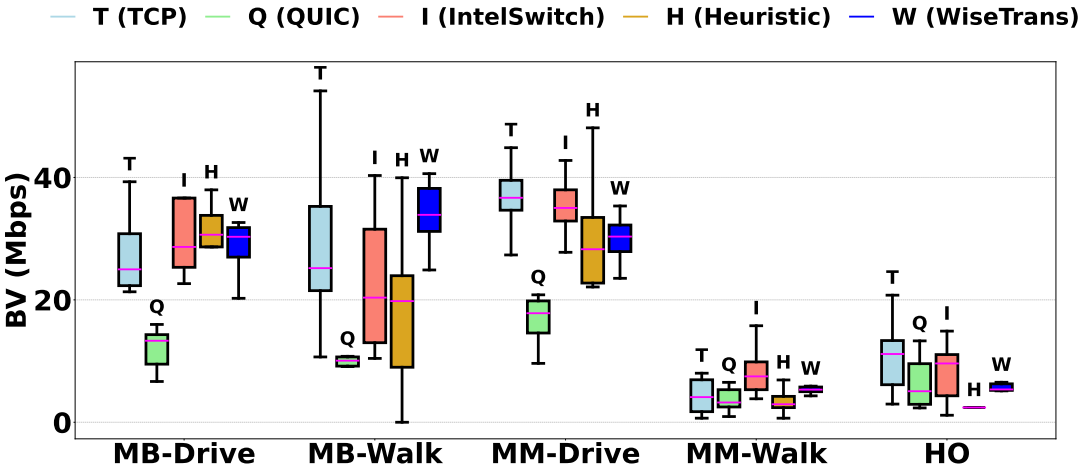


FIGURE 5.18: Real-time adaptive evaluation:Bitrate Variation (Mbps)

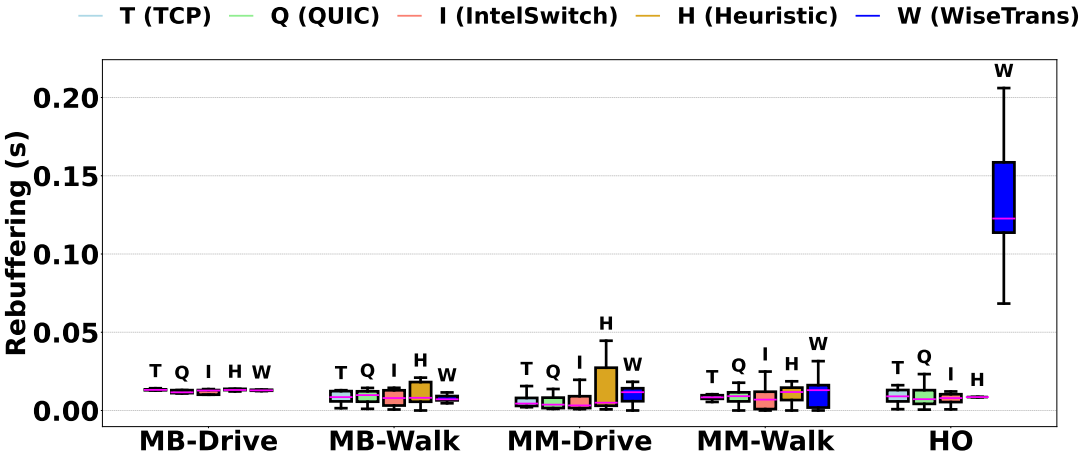


FIGURE 5.19: Real-time adaptive evaluation: Rebuffering (seconds)

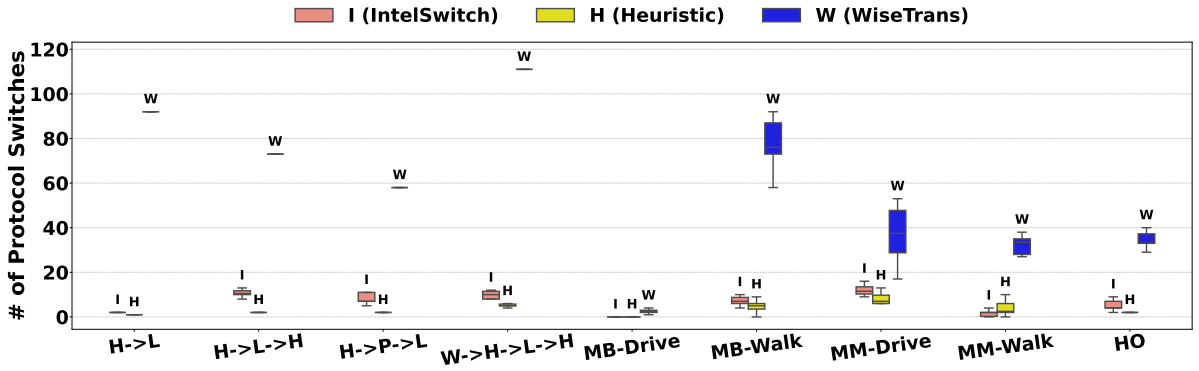


FIGURE 5.20: IntelSwitch number of protocol switch count under real-traces and in controlled traces.

reward fluctuates significantly as the agent explores different actions. Over time, the fluctuations decrease, and the reward gradually stabilizes after 3000 epochs, indicating that the agent has learned an effective policy for protocol selection. Total duration for training is 20 hours. This convergence behavior demonstrates that IntelSwitch successfully adapts its decision-making to maximize QoE in diverse network conditions.

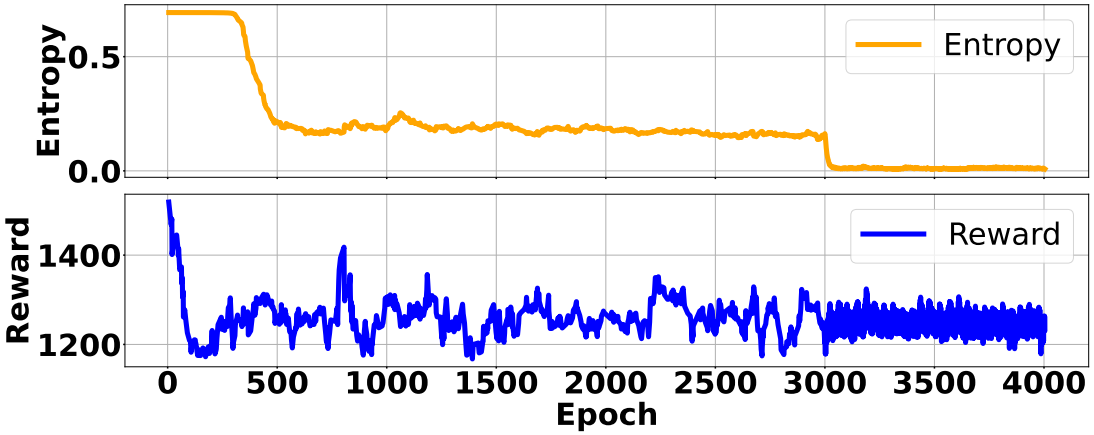


FIGURE 5.21: Convergence time of IntelSwitch protocol decision engine to provide optimal protocol for video streaming

Working of IntelSwitch:

Replay-Based Adaptive Streaming: To understand how IntelSwitch works, we start with understanding Replay-based evaluation of IntelSwitch, where we use controlled traces and run IntelSwitch offline. Fig. 5.22 illustrates the H− >L bandwidth-controlled replay-based experiment. In this bandwidth trace, initially for 150 seconds, the network bandwidth is high up to $500Mbps$, and then from 150 seconds to 300 seconds, the bandwidth remains low and even drops to 0. At the beginning of the video session,

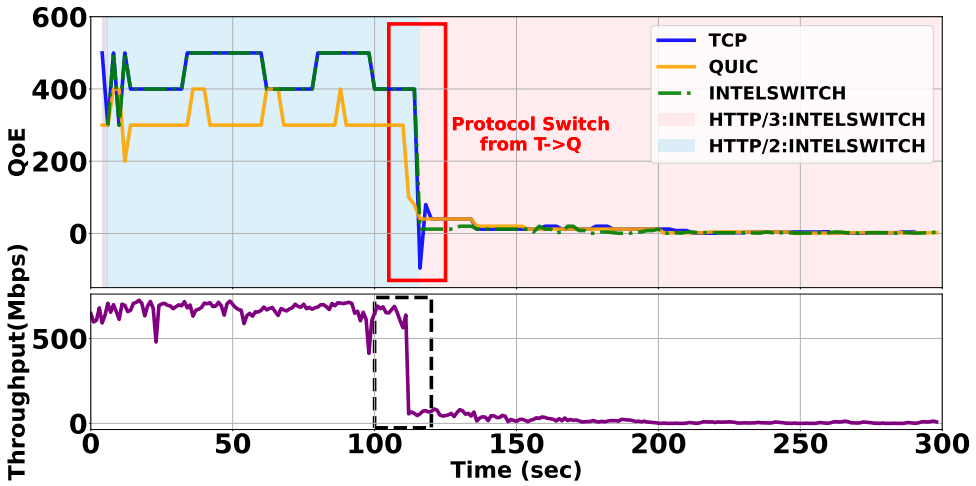


FIGURE 5.22: IntelSwitch switching example in H-L bandwidth trace

the DASH player experiences high application throughput due to the availability of abundant network bandwidth. Under these conditions, we observe that QUIC streams at a lower bitrate than TCP. Consequently, IntelSwitch correctly selects TCP as the preferred transport protocol during the high-bandwidth period, as TCP achieves higher video quality without negatively affecting playback performance. At approximately (t = 130) seconds, the available network bandwidth drops significantly. During this low-bandwidth period, the bandwidth occasionally falls to nearly zero, which may result in increased packet delays, losses, or retransmission events. Under such challenging network conditions, QUIC exhibits better performance than TCP. Accordingly, IntelSwitch adapts by selecting QUIC as the transport protocol. The results show that TCP is more adversely affected by the sudden bandwidth reduction than QUIC. One contributing factor is that TCP had previously been streaming at a higher bitrate, making the abrupt bandwidth drop more disruptive to both segment delivery and the bitrate adaptation process. As a result, the impact of the bandwidth degradation is more pronounced for TCP, leading to greater performance degradation during the low-bandwidth interval. It is important to note that in the TCP-only and QUIC-only configurations, both audio and video streams were delivered over the same transport protocol. Therefore, the observed performance reflects not only the characteristics of the transport protocol itself but also the effects of carrying both media streams over a common transport path.

Real-Time Adaptive Streaming: Fig. 5.23 illustrates how IntelSwitch dynamically switches between protocols based on network conditions in a real time video streaming. It also highlights a specific instance where the model selects QUIC when the network throughput drops and switches back to TCP as the throughput improves. From t=0

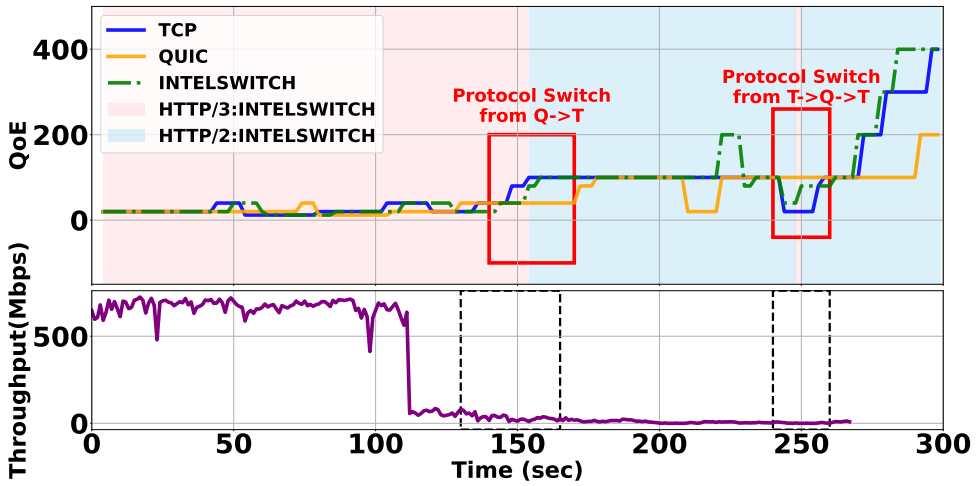


FIGURE 5.23: IntelSwitch switching example in HO scenario

to t=130 seconds, the network throughput was low but not lossy; hence, QUIC and TCP exhibited similar performance, and IntelSwitch chose to operate over QUIC. At t=130 seconds, a handover occurred, increasing the available bandwidth to approximately 900Mbps. IntelSwitch detected this network change and, after one segment request, switched to TCP. Thus, the protocol transition occurred from QUIC to TCP. The QoE of IntelSwitch was found to be comparable to that of TCP, indicating no negative impact from switching; rather, the change contributed to improved user QoE. Following the network change, under high-bandwidth conditions, the ABR algorithm using TCP as the underlying protocol quickly ramped up to a higher bitrate, whereas QUIC maintained streaming at 100Mbps and maximum up to 200Mbps towards the end. Similarly, since IntelSwitch switched to TCP, it was also able to achieve a higher video bitrate, reaching up to 400Mbps. Additionally, at t=245 seconds, the network throughput dropped to 0 for a few seconds. IntelSwitch successfully detected this change and switched to QUIC. Immediately after the network bandwidth increased, IntelSwitch switched back to TCP. Hence, IntelSwitch provides overall better QoE to users compared to using only QUIC for the entire video streaming session.

Ablation Study: IntelSwitch design consists of two main components: (1) assigning different transport protocols to different media segments (audio and video), and (2) dynamically switching transport protocols based on the decisions made by the Protocol Decision Engine. These components work closely together to enable adaptive protocol selection during streaming. The rationale for using different transport protocols for audio and video is primarily to reduce the overhead of transport protocol switching during video streaming sessions. By transmitting audio and video over different transport protocols,

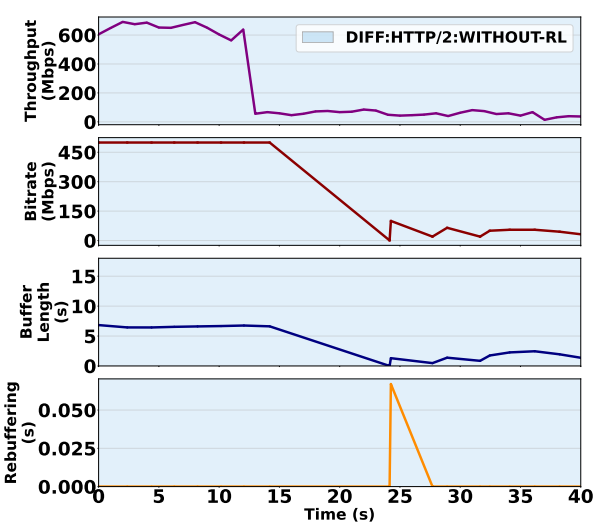


Figure 5.24: DIFF-WITHOUT-RL: Video over TCP, and audio over QUIC, with no protocol switching under an H→L bandwidth pattern.

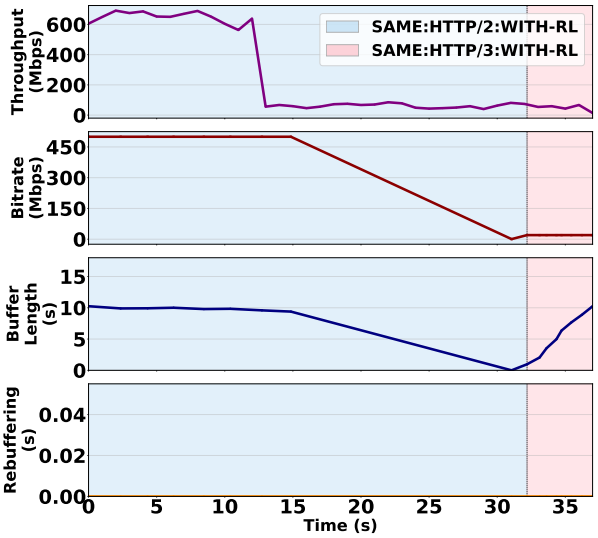


Figure 5.25: SAME-WITH-RL: Audio and video use the same transport protocol with RL-based protocol switching under an H→L bandwidth pattern.

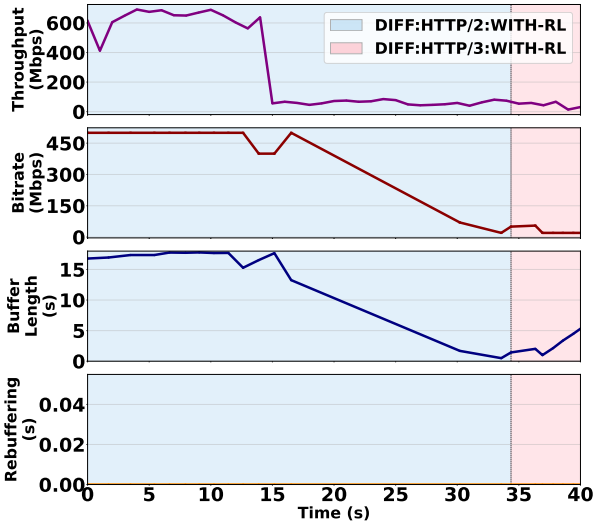


Figure 5.26: DIFF-WITH-RL: Audio and video use different transport protocols with RL-based protocol switching under an H→L bandwidth pattern.

we keep both TCP and QUIC connections active simultaneously. As a result, when the protocol decision engine decides to switch the transport protocol for video streaming, the corresponding connection is already established, thereby reducing switching latency and minimizing additional QoE degradation. To justify our IntelSwitch design choice, we

conducted an ablation study to evaluate the contribution of each component and understand its impact on overall performance. We retain only one component of IntelSwitch and remove the 2nd and then evaluate the performance at the time of protocol switching. We did the following: (1) Different protocols for audio and video without the Protocol Decision Engine (static protocol assignment throughout the video session), (2) The same protocol for both audio and video with the Protocol Decision Engine (3) Different protocols for audio and video with the Protocol Decision Engine (IntelSwitch). Note that when audio and video segments are sent over different protocols we call it DIFF and when they are sent over same protocol we call it SAME.

Results: Figure 5.24 shows the behavior when audio and video are transmitted over different transport protocols, with video over TCP and audio over QUIC. Following the bandwidth drop, the next segment request is issued approximately 10 seconds later. However, this results in a bitrate reduction to 500 Kbps and introduces a rebuffering event of 0.067 seconds. This observation suggests that merely separating audio and video traffic across different transport protocols is insufficient to fully mitigate the impact of sudden bandwidth degradation. Next, we remove the first design component of using separate protocols for audio and video and instead transmit both media streams over the same transport protocol. Protocol selection is dynamically controlled by the RL-based protocol decision engine. In Figure 5.25, the next segment request arrives approximately 16 seconds after the bandwidth drop. Although the bitrate initially decreases to 500 Kbps, it subsequently recovers to 20 Mbps without causing any rebuffering. This behavior can be attributed to the RL-based protocol decision engine, which switches the transport protocol to QUIC after detecting the degraded network conditions. Since QUIC performs better than TCP in low-bandwidth environments, the player is able to maintain uninterrupted playback despite the temporary bitrate reduction. Finally, we combine both design components: the use of different transport protocols for audio and video, and RL-driven protocol switching. Figure 5.26 after the bandwidth drop, the next segment request for a 70 Mbps bitrate is issued after approximately 14 seconds, followed by a request for a 20 Mbps representation after an additional 5 seconds. Unlike the previous configurations, the protocol switch enables the player to adapt more effectively to the changing network conditions, resulting in a faster bitrate recovery while eliminating rebuffering events and avoiding abrupt reductions in video bitrate. Compared to the previous two configurations, this design achieves a better balance between responsiveness and playback stability. We also dig deeper into network logs to see the impact of protocol switching with and without the first component of IntelSwitch shown in figure 5.27 and 5.28. We found that establishing a new QUIC connection incurs lower overhead than establishing

a new TCP+TLS connection because QUIC combines the transport and cryptographic handshakes together. However, the smoothest switching behavior is achieved when an alternate transport connection is already available before switching occurs. Specifically, in the DIFF setup, where an alternate connection is already active, the protocol-switching delay is reduced by approximately 866 ms because the existing connection can be reused immediately, eliminating additional connection-establishment overhead. In contrast, in the SAME setup, establishing a new QUIC connection during switching introduces additional setup delay, increasing the switching time to approximately 1013 ms. Out of this 1013 ms, the QUIC connection establishment first request goes after 640 ms, and then QUIC took 297 ms to establish a new QUIC connection, followed by another 76 ms to send the first QUIC 1500-byte data.

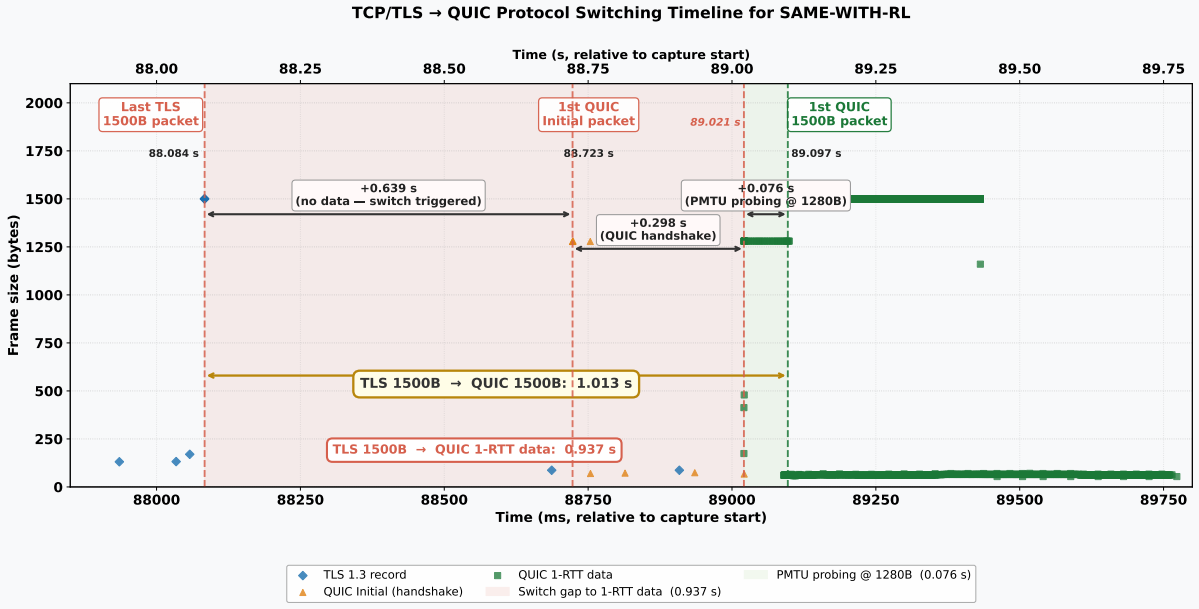


Figure 5.27: Protocol Switch Timeline of SAME-WITH-RL

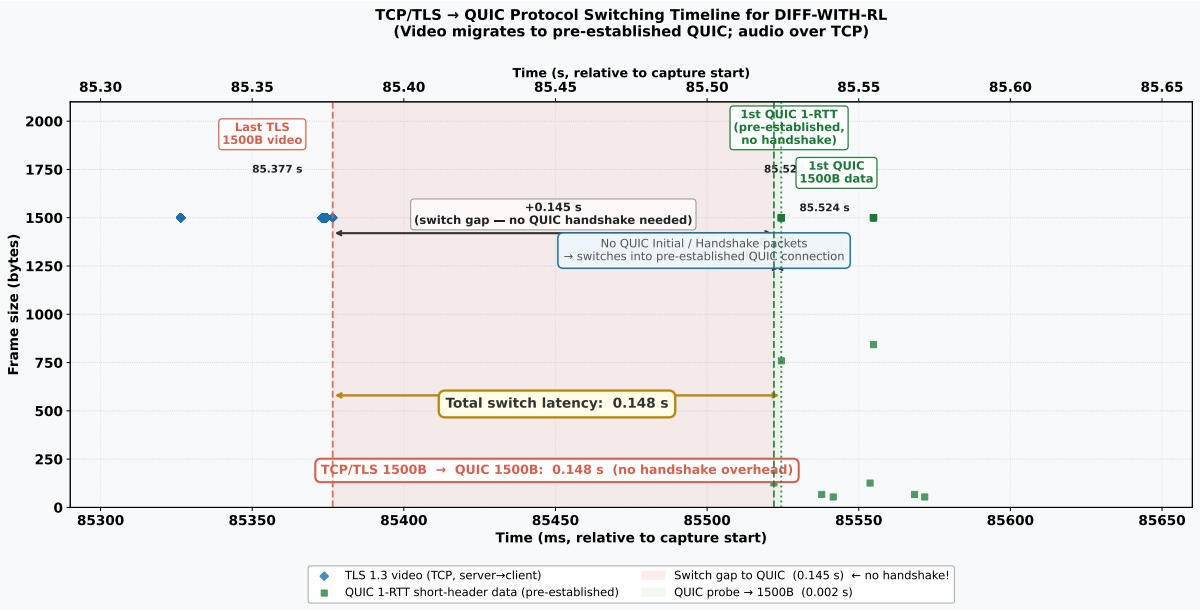


FIGURE 5.28: Protocol Switch Timeline of DIFF-WITH-RL

5.6.2.5 IntelSwitch with Different ABR Algorithms

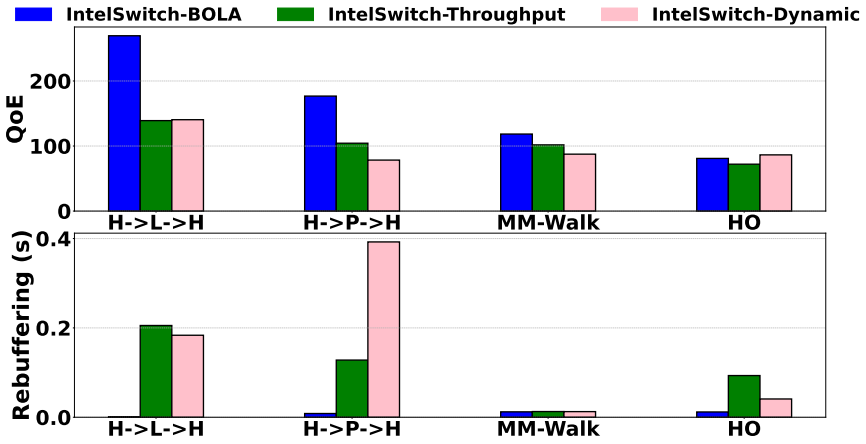


FIGURE 5.29: IntelSwitch comparison across different ABR

We compared the performance of IntelSwitch using the three built-in ABR algorithms in `dash.js`: *ThroughputRule*, *BolaRule*, and *Dynamic* under real and replay-based network traces. Since each ABR responds differently to bandwidth changes, this comparison

reveals which strategy best complements IntelSwitch. As shown in Fig. 5.29, *ThroughputRule* and *Dynamic* struggle to maintain stability when bandwidth drops sharply, leading to frequent rebuffering. In contrast, *BolaRule* uses buffer occupancy for decisions, enabling smoother adaptation, better QoE and lower rebuffering.

IntelSwitch with RL-based ABR algorithm Pensieve: Since Pensieve is primarily trained in offline and trace-driven simulated environments, it does not transfer well to real-world DASH players. We initially attempted to utilize the available pre-trained Pensieve model [78, 82] in our streaming setup. However, to make it compatible, we modified the DASH client to provide the specific input states expected by the model, such as rebuffering values, buffer levels, and chunk sizes. Despite these modifications, the model consistently produced incorrect bitrate decisions. To further investigate, we also retrained Pensieve using our own bitrate ladder under 5G, 4G, and WiFi settings, but the retrained model exhibited similar behavior and failed to yield stable or accurate adaptation decisions. Even after these adjustments, when evaluated without IntelSwitch itself, Pensieve still failed to operate correctly. This failure is not surprising in light of observations from the Pensieve5G work [83], which argue that while Pensieve has been shown to work reasonably well under 3G and 4G environments, it fails to generalize in 5G settings. RL-based methods struggle in 5G because of mis-scaled state normalization (e.g., throughput in Mbps being orders of magnitude larger in 5G than in older settings) and mismatch in how download time is measured (simulator computes download time $= chunk_{end} - request_{start}$, while real network behavior introduces additional RTT gaps). This limitation arises because Pensieve's training environment assumes idealized buffer dynamics, fixed segment durations, and synthetic throughput traces, which do not capture the complexities of real network stacks and player-level variability, particularly at higher bandwidths, such as in Ethernet with a 1Gbps link, 5G mmWave, and 5G mid-band. Moreover, replay-based testing of Pensieve would require collecting data under all bitrate levels and for both protocols, which is impractical; hence, we rely on live testing only. Even after our efforts to align inputs and retrain, Pensieve could not operate correctly in our 5G environment. As a result, Pensieve fails to generalize to real deployments, and we were unable to meaningfully test it with our IntelSwitch setup as live testing.

In summary, our study demonstrates that the assumption that a single transport protocol can consistently outperform others across all network environments does not hold in practice. To overcome this limitation, we designed and implemented IntelSwitch, a reinforcement learning–based adaptive transport protocol switching framework that dynamically selects between TCP and QUIC based on real-time network conditions. Through

extensive experimentation across nine diverse network scenarios, we demonstrated that IntelSwitch substantially improves QoE, achieving up to 47% higher QoE compared to TCP-only streaming and up to 114% higher compared to QUIC-only streaming. These gains were made possible by coordinated modifications to both the DASH player and the Chromium browser, enabling seamless protocol switching during video playback. While these results highlight the strong potential of adaptive transport protocol selection, they also reveal several challenges that arise when moving toward real-world deployment, including compatibility with heterogeneous congestion control algorithms, variability in ABR strategies, and device-level constraints.

5.7 Summary of the Chapter

In this chapter of the thesis, we found that the assumption that a single transport protocol can consistently outperform others across all scenarios is not supported. To address this limitation, we propose **IntelSwitch**, a reinforcement learning-based framework for adaptive transport protocol selection. Through an extensive evaluation involving 9 different types of networks, we demonstrate that **IntelSwitch** significantly enhances QoE, where available bandwidth fluctuates between high and low levels. To enable this capability, we implemented modifications in both the DASH player and the Chromium browser. In our experiments, **IntelSwitch** achieves up to 47% higher QoE compared to streaming solely over TCP, and up to 114% higher QoE compared to streaming solely over QUIC. These results suggest that widespread adoption of **IntelSwitch** could substantially improve QoE at scale.

Having established the motivations, design, and benefits of IntelSwitch, we now turn to the final chapter, where we discuss the broader insights gained throughout this thesis and outline promising directions for future research in transport protocol-aware video streaming.

Chapter 6

Conclusion and Future Work

'If I have seen further, it is by standing on the shoulders of giants."
— Isaac Newton

Over the years, numerous new Internet protocols have been proposed to overcome the limitations of older protocol designs. As a result, a significant body of research has focused on understanding how these new protocols perform in comparison to their predecessors across diverse network environments. QUIC is one such modern transport protocol, introduced to address several well-known shortcomings of the legacy transport protocol TCP, particularly in terms of connection establishment latency and resilience to loss. Unlike TCP, QUIC is implemented in user space and is tightly integrated into modern web browsers, which has enabled rapid deployment but also introduced new design and implementation challenges.

Browsers implementation of QUIC incorporates fallback mechanisms to ensure robustness. When the new protocol fails for any reason, the browser falls back to a well-tested legacy protocol such as TCP to preserve connectivity and application continuity. The presence of such fallback mechanisms does not imply that the new protocol is fundamentally flawed; rather, it reflects the reality that no protocol performs optimally under all possible conditions. Different protocols excel in different environments, and fallback mechanisms are a necessary design choice to handle scenarios where the newer protocol encounters limitations.

This thesis demonstrates that performance limitations often arise not from the transport protocol itself, but from external factors that influence how the protocol operates

in practice. Through large-scale measurement using our H3B tool spanning more than 5474 hours of YouTube streaming sessions, we revealed that QUIC-enabled browsers frequently experience performance degradation due to repeated protocol switching triggered by connection racing implementation within the browser. Importantly, we demonstrated that browsers do not distinguish between failures caused by middleboxes and those caused by network congestion, leading to unnecessary transitions between QUIC and TCP and resulting in lower bitrates, higher stalls, and unstable QoE. To address these challenges, we first analyzed the implementation of connection racing within the browser. We further demonstrated how unnecessary protocol switching due to connection racing has a significant impact on long-lived video sessions. We then proposed a browser-level modification to intelligently control connection racing, preventing harmful switching when QUIC's poor performance is not due to middlebox interference. Alongside this, we designed a server-side eBPF-based congestion-state transfer mechanism that enables QUIC to TCP congestion window transfer during protocol switching. Our evaluations showed that this significantly reduces the performance penalty of protocol switching, allowing the TCP connection to resume transmission at a much higher rate and improving buffer stability.

During the course of this study, we observed both from our own findings and from prior work that no single transport protocol consistently delivers the best performance across all network environments. This realization highlights the need for an adaptive transport protocol switching mechanism specifically tailored for video streaming, a gap that has largely remained unaddressed. While most existing works on adaptive protocol selection focus on web services, they overlook the unique challenges involved in designing such mechanisms for video streaming, a domain that dominates today's Internet traffic. These challenges, ranging from handling continuous media delivery to maintaining stable QoE under dynamic conditions, have contributed to the lack of an adaptive transport protocol switching framework specifically designed for video streaming. To bridge this gap, we introduced IntelSwitch, a reinforcement learning–based adaptive transport protocol selection framework that enables intelligent, real-time switching between QUIC and TCP. IntelSwitch is designed through coordinated modifications to both the DASH player and the Chromium browser, allowing it to make protocol decisions that maximize QoE under dynamically changing network conditions.

In conclusion, this thesis demonstrates that achieving consistently high video streaming performance requires the intelligent and adaptive use of transport protocols, rather than relying on any single protocol by default. As both Internet infrastructure and application

demands continue to evolve, such adaptive systems will be essential for delivering stable, high-quality user experiences at scale.

6.1 Future Work

Several promising directions exist that can extend and enhance the work presented in this thesis. In this thesis, our experiments were conducted in a controlled environment where many system-level factors were held constant across runs. As observed with the IntelSwitch framework real-world deployment, however, introduces additional complexity, including variability across congestion control algorithms, ABR schemes, hardware capabilities, device-level resource constraints, and background system load. Ensuring interoperability and stability in such diverse environments remains an important challenge.

In this thesis, we utilized CUBIC as the congestion control algorithm for both TCP and QUIC, focusing primarily on user-level processing overhead. A natural extension of this work is to investigate how different congestion control algorithms influence the relative performance of TCP and QUIC, and how protocol switching behaves when each protocol operates with a different congestion control mechanism. Such an exploration would provide deeper insight into whether adaptive switching remains effective or becomes even more necessary under heterogeneous congestion control strategies. Therefore, the future work directions are summarized as follows:

- Our analysis indicates that connection racing is a prime reason behind the poor performance of YouTube streaming over a QUIC-enabled browser. However, we also observe cases where QUIC-enabled sessions perform poorly even when no protocol switching occurs. Furthermore, even after manually disabling connection racing in the modified browser, the original browser still outperformed the modified version in roughly 40% of the cases. These findings indicate that additional factors contributing to QUIC performance remain unexplored and require further investigation as future work.

- Video streaming performance is also strongly shaped by the Adaptive Bitrate (ABR) algorithm used by the player. Our current framework relies on the default ABR modules available in dash.js, but a wide range of heuristic and ML-based ABR algorithms are deployed in practice. Understanding how different ABR algorithms

interact with an adaptive transport protocol switching framework is an important direction for future research. This is particularly relevant because our approach uses reinforcement learning for protocol selection, and integrating it with modern ML-based ABRs may require a richer training environment capable of providing realistic and diverse feedback to the protocol decision engine.

- Another promising direction is extending the design and evaluation of IntelSwitch to mobile devices. While this thesis focuses on desktop environments, the majority of video streaming consumption occurs on smartphones and tablets. Mobile platforms introduce unique constraints such as limited CPU resources, concurrent background applications, and aggressive power-management policies, which can magnify the user-space processing overhead of QUIC. Consequently, adaptive protocol switching may behave very differently on mobile devices, requiring a specialized design tailored to mobile constraints. Moreover, transport protocol choice on mobile devices is influenced not only by throughput and QoE but also by energy consumption and data usage. TCP often sustains higher bitrates, but these higher rates can accelerate battery depletion on constrained devices. QUIC, on the other hand, may operate at lower bitrates due to user-level processing overhead, which can reduce energy usage and slow down data consumption, beneficial for users with battery or data limits. In cellular networks where data caps are common, QUIC may be the preferred choice when conserving data is important, whereas TCP becomes advantageous when maximizing video quality is the priority. These observations highlight the need for context-aware, adaptive protocol switching on mobile devices, where the selection between TCP and QUIC depends on bitrate requirements, battery constraints, and data usage considerations.

- Beyond adaptive switching, another key direction for future work is the development of a more generalized congestion-state transfer mechanism. The preliminary results in this thesis demonstrate that transferring the congestion window across protocols can significantly mitigate the performance penalty of QUIC-to-TCP fallback. However, the current implementation assumes that both protocols use the same congestion control algorithm. In real deployments, TCP and QUIC often run different congestion control algorithms, and a direct transfer of congestion window values may not be valid or effective. Additionally, different ABR algorithms request segments of different bitrates, thereby altering throughput demand and influencing the underlying congestion evolution. These factors interact in complex ways during the

streaming process. Therefore, future work should explore a generalized, protocol-agnostic congestion-state transfer strategy that can operate effectively both when the browser performs fallback and when the IntelSwitch framework initiates protocol switching. Such a solution must account for heterogeneous congestion control algorithms, ABR-induced variability, and the intricate interplay between transport behavior and application-level decision making.

Taken together, these directions highlight the need for a comprehensive, real-world training and evaluation setup that captures the variability of modern Internet conditions, congestion control behaviors, ABR strategies, and device-level resource constraints. Advancing adaptive protocol switching in such environments remains both a challenging and impactful avenue for future exploration.

References

[1] Sandvine Incorporated. Global internet phenomena report: 2023. Technical report, Sandvine, 2023. URL https://www.sandvine.com/hubfs/Sandvine_Redesign_2019/Downloads/2023/reports/Sandvine%20GIPR%202023.pdf. Accessed: 2025-07-20.

[2] Applogic Networks. Sandvine's 2024 global internet phenomena report: Global internet usage continues to grow. https://shorturl.at/ncl4C, 2024. Accessed: 2025-07-20.

[3] Geoff Huston. A quic progress report. https://blog.apnic.net/2025/06/17/a-quic-progress-report/, Jun 17 2025. Accessed: 2025-XX-XX.

[4] J. Iyengar, Ed. Fastly, M. Thomson, and Ed. Mozilla. QUIC: A UDP-Based Multiplexed and Secure Transport. https://datatracker.ietf.org/doc/html/rfc9000, 2021. [Online; accessed 10-05-2022].

[5] B. Trammell M. Kuehlewind. Applicability of the QUIC Transport Protocol. https://www.ietf.org/archive/id/draft-ietf-quic-applicability-09.html, 2021. [accessed July 2026].

[6] Massimiliano Stucchi. Mongolia Joins Growing Number of Countries Reducing Openness and Resilience of the Internet. https://rb.gy/zyolpk, 2020. [Online; accessed July 2026].

[7] Hao Li, Changhao Wu, Guangda Sun, Peng Zhang, Danfeng Shan, Tian Pan, and Chengchen Hu. Programming network stack for middleboxes with rubik. In *NSDI*, 2021.

[8] Sreeni Tellakula. Comparing HTTP/3 vs. HTTP/2 Performance. https://blog.cloudflare.com/http-3-vs-http-2/, 2020. [Online; accessed 19-October-2021].

[9] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the conference of the ACM special interest group on data communication*, 2017.

[10] Mathis Engelbart and Jörg Ott. Congestion control for real-time media over quic. In *Proceedings of the 2021 Workshop on Evolution, Performance and Interoperability of QUIC*.

[11] Michael Seufert, Raimund Schatz, Nikolas Wehner, and Pedro Casas. Quicker or not?-an empirical analysis of quic vs tcp for video streaming qoe provisioning. In *2019 22nd Conference on ICIN*.

[12] Tanya Shreedhar, Rohit Panda, Sergey Podanev, and Vaibhav Bajpai. Evaluating quic performance over web, cloud storage and video workloads. *IEEE Transactions on Network and Service Management*, 2021.

[13] Arash Molavi Kakhki, Samuel Jero, David Choffnes, Cristina Nita-Rotaru, and Alan Mislove. Taking a long look at quic: an approach for rigorous evaluation of rapidly evolving transport protocols. In *Proceedings of the 2017 Internet Measurement Conference*, pages 290–303, 2017.

[14] Naveenraj Muthuraj, Nooshin Eghbal, and Paul Lu. Replication: "taking a long look at quic". In *Proceedings of the 2024 ACM on Internet Measurement Conference*, 2024.

[15] Lin Qi, Zhihong Qiao, Aowei Zhang, Hui Qi, Weiwu Ren, Xiaoqiang Di, and Rui Wang. Performance analysis of quic-udp protocol under high load. In *International Conference on Mobile Wireless Middleware, Operating Systems, and Applications*, pages 69–77. Springer, 2020.

[16] Xumiao Zhang, Shuowei Jin, Yi He, Ahmad Hassan, Z Morley Mao, Feng Qian, and Zhi-Li Zhang. Quic is not quick enough over fast internet. In *Proceedings of the ACM Web Conference 2024*, pages 2713–2722, 2024.

[17] Benedikt Jaeger, Johannes Zirngibl, Marcel Kempf, Kevin Ploch, and Georg Carle. Quic on the highway: Evaluating performance on high-rate links. In *2023 IFIP Networking Conference (IFIP Networking)*, pages 1–9. IEEE, 2023.

[18] Zhilong Zheng, Yunfei Ma, Yanmei Liu, Furong Yang, Zhenyu Li, Yuanbo Zhang, Jiuhai Zhang, Wei Shi, Wentao Chen, Ding Li, et al. Xlink: Qoe-driven multi-path quic transport in large-scale video services. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021.

[19] Mirko Palmer, Thorben Krüger, Balakrishnan Chandrasekaran, and Anja Feldmann. The quic fix for optimal video streaming. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, pages 43–49, 2018.

[20] Theodoros Karagkioules, Dimitrios Tsilimantos, Stefan Valentin, Florian Wamser, Bernd Zeidler, Michael Seufert, Frank Loh, and Phuoc Tran-Gia. A public dataset for youtube's mobile streaming client. In *2018 Network Traffic Measurement and Analysis Conference (TMA)*, pages 1–6. IEEE, 2018.

[21] Florian Wamser, Michael Seufert, Pedro Casas, Ralf Irmer, Phuoc Tran-Gia, and Raimund Schatz. Yomoapp: A tool for analyzing qoe of youtube http adaptive streaming in mobile networks. In *2015 European Conference on Networks and Communications (EuCNC)*. IEEE, 2015.

[22] Pengqiang Bi, Yifei Zou, Mengbai Xiao, Dongxiao Yu, Yijun Li, Zhixiong Liu, and Qun Xie. LiteQUIC: Improving QoE of video streams by reducing CPU overhead of QUIC. In *Proceedings of the 32nd ACM International Conference on Multimedia*, pages 7918–7927, 2024.

[23] Craig Gutterman, Katherine Guo, Sarthak Arora, Xiaoyang Wang, Les Wu, Ethan Katz-Bassett, and Gil Zussman. Requet: Real-time qoe detection for encrypted youtube traffic. In *Proceedings of the 10th ACM Multimedia Systems Conference*, pages 48–59, 2019.

[24] Sarah Wassermann, Pedro Casas, Michael Seufert, and Florian Wamser. On the analysis of youtube qoe in cellular networks through in-smartphone measurements. In *2019 12th IFIP Wireless and Mobile Networking Conference (WMNC)*. IEEE, 2019.

[25] Chromium. Chromium Code Search. `https://source.chromium.org/chromium/chromium/src`. [Online; accessed July 2026].

[26] Ryan Hamilton. Re: [QUIC] graceful fallback to tcp. `https://mailarchive.ietf.org/arch/msg/quic/ph1IAVBa5pW1AgDvr8fbD741Auw/`, 2016. [Online; accessed July 2026].

[27] Chromium Code Search. QUIC error codes. `https://source.chromium.org/chromium/chromium/src/+/HEAD:net/third_party/quiche/src/quiche/quic/core/quic_error_codes.h`, 2022. [Online; accessed July 2026].

[28] Firefox. Mozilla-Central. `https://searchfox.org/mozilla-central/source/netwerk`. [Online; accessed July 2026].

[29] Chao Chen, Lark Kwon Choi, Gustavo De Veciana, Constantine Caramanis, Robert W Heath, and Alan C Bovik. Modeling the time—varying subjective quality of http video streams with rate adaptations. *IEEE Transactions on Image Processing*, 23(5):2206–2221, 2014.

[30] Zhengfang Duanmu, Kai Zeng, Kede Ma, Abdul Rehman, and Zhou Wang. A quality-of-experience index for streaming video. *IEEE Journal of Selected Topics in Signal Processing*, 11(1):154–166, 2016.

[31] Christos George Bampis, Zhi Li, Anush Krishna Moorthy, Ioannis Katsavounidis, Anne Aaron, and Alan Conrad Bovik. Study of temporal effects on subjective video quality of experience. *IEEE Transactions on Image Processing*, 26(11):5217–5231, 2017.

[32] Christos G Bampis, Zhi Li, Ioannis Katsavounidis, Te-Yuan Huang, Chaitanya Ekanadham, and Alan C Bovik. Towards perceptually optimized adaptive video streaming-a realistic quality of experience database. *IEEE Transactions on Image Processing*, 30:5182–5197, 2021.

[33] Sevket Arisu and Ali C Begen. Quickly starting media streams using quic. In *Proceedings of the 23rd Packet Video Workshop*, 2018.

[34] Colin Perkins and Jörg Ott. Real-time audio-visual media transport over QUIC. In *In Proceedings of EPIQ'20*.

[35] Joris Herbots, Maarten Wijnants, Wim Lamotte, and Peter Quax. Cross-layer metrics sharing for quicker video streaming. In *Proceedings of CoNEXT'20*.

[36] Daniele Lorenzi, Minh Nguyen, Farzad Tashtarian, Simone Milani, Hermann Hellwagner, and Christian Timmerer. 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*.

[37] Quentin De Coninck, François Michel, Maxime Piraux, Florentin Rochet, Thomas Given-Wilson, Axel Legay, Olivier Pereira, and Olivier Bonaventure. Pluginizing QUIC. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 59–74. 2019.

[38] Sarah Cook, Bertrand Mathieu, Patrick Truong, and Isabelle Hamchaoui. Quic: Better for what and for whom? In *2017 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, IEEE, 2017.

[39] Korian Edeline and Benoit Donnet. A first look at the prevalence and persistence of middleboxes in the wild. In *2017 29th International Teletraffic Congress (ITC 29)*, volume 1, pages 161–168. IEEE, 2017.

[40] Konrad Yuri Gbur and Florian Tschorsch. A QUIC (K) way through your firewall? *arXiv preprint arXiv:2107.05939*, 2021.

[41] Yihui Yan and Zhice Yang. When quic's connection migration meets middleboxes a case study on mobile wi-fi hotspot. In *2021 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2021.

[42] Andrew. Yourtchenko Dan. Wing. Happy Eyeballs: Success with Dual-Stack Hosts. `https://datatracker.ietf.org/doc/html/rfc6555`, 2012. [accessed July 2026].

[43] Marcel Kempf, Benedikt Jaeger, Johannes Zirngibl, Kevin Ploch, and Georg Carle. Quic on the fast lane: Extending performance evaluations on high-rate links. *Computer Communications*, 223:90–100, 2024.

[44] Jia Zhang, Enhuan Dong, Zili Meng, Yuan Yang, Mingwei Xu, Sijie Yang, Miao Zhang, and Yang Yue. WiseTrans: Adaptive transport protocol selection for mobile web service. In *Proceedings of the Web Conference 2021*, 2021.

[45] Jia Zhang, Shaorui Ren, Enhuan Dong, Zili Meng, Yuan Yang, Mingwei Xu, Sijie Yang, Miao Zhang, and Yang Yue. Reducing mobile web latency through adaptively selecting transport protocol. *IEEE/ACM Transactions on Networking*, 2023.

[46] Mengying Zhou, Zheng Li, Shihan Lin, Xin Wang, and Yang Chen. FlexHTTP: an intelligent and scalable HTTP version selection system. In *Proceedings of the 2nd European Workshop on Machine Learning and Systems*, 2022.

[47] Anirudh Ganji and Muhammad Shahzad. Characterizing the performance of QUIC on android and wear OS devices. In *2021 International Conference on Computer Communications and Networks (ICCCN)*. IEEE, 2021.

[48] Jana Iyengar and Martin Thomson. RFC 9000: QUIC: A UDP-based multiplexed and secure transport. *Omtermet Emgomeeromg Task Force*, 2021.

[49] M Duke. RFC 9369 QUIC version 2. 2023.

[50] Ian Swett. QUIC Deployment Experience @ Google. `https://www.ietf.org/proceedings/96/slides/slides-96-quic-3.pdf`, 2017. [Online; accessed 19-October-2021].

[51] Ian Swett David Schinazi, Fan Yang. Chrome is deploying HTTP/3 and IETF QUIC. `https://blog.chromium.org/2020/10/chrome-is-deploying-http3-and-ietf-quic.html`, 2020. [Online; accessed 19-October-2021].

[52] Alex Yu. Benchmarking QUIC. `https://medium.com/@the.real.yushuf/benchmarking-quic-1fd043e944c7`, 2020. [Online; accessed 19-October-2020].

[53] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the conference of the ACM special interest group on data communication*, 2017.

[54] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. PCC vivace: Online-learning congestion control. In *15th {USENIX} Symposium on Networked Systems Design and Implementation*, 2018.

[55] Javad Nejati and Aruna Balasubramanian. An in-depth study of mobile browser performance. In *Proceedings of the 25th International Conference on World Wide Web*, 2016.

[56] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. Toward formally verifying congestion control behavior. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*.

[57] Frank Cangialosi, Akshay Narayan, Prateesh Goyal, Radhika Mittal, Mohammad Alizadeh, and Hari Balakrishnan. Site-to-site internet traffic control. In *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021.

[58] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *2015 USENIX Annual Technical Conference*.

[59] Craig Gutterman, Katherine Guo, Sarthak Arora, Xiaoyang Wang, Les Wu, Ethan Katz-Bassett, and Gil Zussman. Requet Dataset. `https://github.com/Wimnet/RequetDataSet`, 2019. [accessed July 2026].

[60] YouTube. YouTube Embedded Players and Player Parameters. `https://developers.google.com/youtube/player_parameters`, 2021. [Online; accessed July 2026].

[61] Sheldon M. Ross. *Introduction to Probability and Statistics for Engineers and Scientists*. Academic Press, Burlington, MA, 4 edition, 2010.

[62] Bernard L Welch. The generalization of 'student's'problem when several different population varlances are involved. *Biometrika*, 34(1-2):28–35, 1947.

[63] Douglas C Montgomery. *Design and analysis of experiments*. John wiley & sons, 2017.

[64] Google Chrome. web-networks-Network-Quality-Estimation-in-Chrome. urlhttps://www.w3.org/2020/02/05-web-networks-Network-Quality-Estimation-in-Chrome.pdf, 2020. [Online; accessed July 2026].

[65] Chrome. Network Information API Sample. urlhttps://googlechrome.github.io/samples/network information/index.html, 2020. [Online; accessed July 2026].

[66] Shawn Ostermann. tcptrace. `http://tcptrace.org/`, 2003. [accessed July 2026].

[67] Big buck bunny test videos. `https://test-videos.co.uk/bigbuckbunny/mp4-h264`. Accessed: 2025-02-12.

[68] Gaetano Carlucci, Luca De Cicco, and Saverio Mascolo. Http over udp: an experimental investigation of quic. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 609–614. ACM, 2015.

[69] Géza Szabó, Sándor Rácz, Daniel Bezzera, Igor Nogueira, and Djamel Sadok. Media qoe enhancement with quic. In *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 219–220. IEEE, 2016.

[70] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In *Usenix Annual Technical Conference*, pages 417–429, 2015.

[71] FFmpeg. FFmpeg. `https://ffmpeg.org/`. Accessed: July 2026.

[72] Rostand A K. Fezeu, Claudio Fiandrino, Eman Ramadan, Jason Carpenter, Lilian Coelho De Freitas, Faaiq Bilal, Wei Ye, Joerg Widmer, Feng Qian, and Zhi-Li Zhang. Unveiling the 5g mid-band landscape: From network deployment to performance and application qoe. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 358–372, 2024.

[73] Shubham Chaudhary, Navneet Mishra, Keshav Gambhir, Tanmay Rajore, Arani Bhattacharya, and Mukulika Maity. Compact: Content-aware multipath live video streaming for online classes using video tiles. In *Proceedings of the 16th ACM Multimedia Systems Conference*, pages 201–213, 2025.

[74] man7.org Linux man-pages. tc-netem(8) - linux traffic control network emulator. `https://man7.org/linux/man-pages/man8/tc-netem.8.html`, 2024. Accessed: 2025-07-09.

[75] Zifan Wu, Chao Yu, Deheng Ye, Junge Zhang, Hankz Hankui Zhuo, et al. Coordinated proximal policy optimization. *Advances in Neural Information Processing Systems*, 34:26437–26448, 2021.

[76] LiteSpeed Technologies Inc. Openlitespeed web server. `https://openlitespeed.org/`, 2013. Accessed: 2025-08-03.

[77] Arvind Narayanan, Eman Ramadan, Rishabh Mehta, Xinyue Hu, Qingxu Liu, Rostand AK Fezeu, Udhaya Kumar Dayalan, Saurabh Verma, Peiqi Ji, Tao Li, et al. Lumos5g: Mapping and predicting commercial mmwave 5g throughput. In *Proceedings of the ACM internet measurement conference*, pages 176–193, 2020.

[78] Arvind Narayanan, Xumiao Zhang, Ruiyang Zhu, Ahmad Hassan, Shuowei Jin, Xiao Zhu, Xiaoxuan Zhang, Denis Rybkin, Zhengxuan Yang, Zhuoqing Morley Mao, et al. A variegated look at 5g in the wild: performance, power, and qoe implications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 610–625, 2021.

[79] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. BOLA: Near-optimal bitrate adaptation for online videos. *IEEE/ACM Transactions on Networking*, 28(4): 1698–1711, 2020.

[80] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 325–338, 2015.

[81] Johannes Weiner. PSI – Pressure Stall Information. `https://docs.kernel.org/accounting/psi.html`, April 2018. Linux Kernel Documentation.

[82] Arvind Narayanan, Xumiao Zhang, Ruiyang Zhu, Ahmad Hassan, Shuowei Jin, Xiao Zhu, Xiaoxuan Zhang, Denis Rybkin, Zhengxuan Yang, Z. Morley Mao, Feng Qian, and Zhi Li Zhang. A variegated look at 5g in the wild: Performance, power, and QoE implications. In *Proceedings of the 2021 ACM SIGCOMM Conference (SIGCOMM '21)*, pages 610–625, 2021. Artifact repository: `https://github.com/SIGCOMM21-5G/artifact`.

[83] Tianchi Huang (godka). pensieve-5g: Pensieve adaptation for 5g networks. `https://github.com/godka/pensieve-5G`, 2021. GitHub repository; accessed 2025-09-26.

List of Publications

1. **Chaudhary, Sapna**, Mukulika Maity, Sandip Chakraborty, and Naval Kumar Shukla. "A Dataset for Analyzing Streaming Media Performance over HTTP/3 Browsers." accepted in Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track (NeurIPS). 2023. (Published)

2. **Chaudhary, Sapna**, Naval Kumar Shukla, Prince Sachdeva, Sandip Chakraborty, and Mukulika Maity. "Managing Connections by QUIC-TCP Racing: A First Look of Streaming Media Performance over Popular HTTP/3 Browsers", IEEE Transactions on Network and Service Management (TNSM), 2023. (Published)

3. **Sapna Chaudhary**, Mukulika Maity, and Sandip Chakraborty "Intel-Switch: Optimized Adaptive Protocol Selection Mechanism for Video Streaming Application" In 2025 17th International Conference on COMmunication Systems and NETworks (COMSNETS), pp. 799-801. IEEE, 2025.

4. **Sapna Chaudhary**, Kartikeya Sehgal, Sandip Chakraborty, and Mukulika Maity. "Learning to Switch: Adaptive Transport Protocols for High-Performance Video Streaming" IEEE Transactions on Networking (ToN) (Under Review).

Courses

1. Machine Learning
2. Wireless Networks
3. Networks and Systems Security II
4. Advances Topics in Mobile Computing