


# Managing Connections by QUIC-TCP Racing: A First Look of Streaming Media Performance over Popular HTTP/3 Browsers

Sapna Chaudhary , Naval Kumar Shukla, Prince Sachdeva, Sandip Chakraborty  *Senior Member, IEEE*,  
Mukulika Maity  *Member, IEEE*

**Abstract**—With the push towards HTTP/3, most modern browsers have started supporting it. HTTP/3 uses QUIC, which runs on top of UDP. However, a few Internet middleboxes tend to block or rate-limit UDP traffic; therefore, the browsers ensure compatibility by enabling *connection racing* via simultaneously initiating a TCP connection with the QUIC one. Each time the QUIC protocol suffers, connection racing is activated, and whichever protocol wins the race is further used for the application. In this paper, we study how browsers implement this connection racing mechanism and analyze its impact on applications that require a long-lived Internet connection, such as video streaming. We perform a large-scale measurement study across different browsers (Chrome/Chromium and Firefox), which helps to analyze why and how the repeated connection racing between protocols affects adaptive streaming QoE over 6013 YouTube sessions covering 5474 hours of streaming. Interestingly, we observe that YouTube QoE over an HTTP/3 supported browser suffers many times, and repeated connection racing is one of the major reasons that hinder the performance. We modified the Chromium browser source code to disable the connection racing altogether and observed that it improves the QoE for YouTube streaming over this modified browser. We then design and implement a solution that dynamically decides when to enable connection racing. We observe that it improves the QoE compared to the original browser. The analysis presented in this paper highlights the requirement of revisiting how browsers handle and switch between protocols through connection racing to ensure compatibility with middleboxes.

**Index Terms**—HTTP/3, QUIC, Connection Racing, QoE, Video Streaming, Browsers

## I. INTRODUCTION

In recent years, several Internet giants like Google, Facebook, Akamai, Cloudflare, Apple, etc., have migrated their services over HTTP/3. However, the browser adoption of HTTP/3 faced an exciting challenge. HTTP/3 works over QUIC, which works on top of UDP. Few legacy middleboxes still tend to block or rate-limits UDP connections, particularly for the apparent reason of network security [1], [2], [3]. Several HTTP/3-supported browsers, like Google Chrome, Mozilla Firefox, etc., take a strategy to enable backward compatibility for the middleboxes blocking or rate-limiting UDP by taking support of connection racing between the legacy TCP protocol

and a new QUIC protocol. If a protocol suffers or fails, the browser uses connection racing, and whichever connection wins the race is used to serve the requests. Notably, backward compatibility is designed for the scenario in which the browser suffers due to the presence of middleboxes.

In this paper, we observe that the repeated switching of the protocol at the browser level due to connection racing is not rare. It occurs quite frequently for specific scenarios, like under low network bandwidth at the low and middle-economy countries [4], [5], or when the bandwidth fluctuates, say, under mobility (Section III). Such repeated protocol switching by the browsers has multiple consequences;

- The underlying transport protocol becomes unstable. While this might not hurt the performance of services, like web browsing, it can affect long-lived Internet connections, like video streaming.
- Although QUIC provides zero or one-RTT connection establishment, this advantage might not be visible if the underlying UDP datagram gets lost. It is highly possible to have a connection failure due to poor network bandwidth. Although a QUIC client uses 0/1-RTT connection establishment with the remote host, the packets (Client Hello, Server Hello, etc.) are over the unreliable UDP datagrams and use one additional level of redirection (application  $\rightarrow$  transport  $\rightarrow$  network) compared to TCP connection establishment (transport  $\rightarrow$  network). Consequently, the datagrams may fail to get delivered over a lossy network, whereas the TCP segments reach successfully. Therefore, even in the absence of a middlebox blocking UDP datagrams, protocol switching happens from QUIC to TCP.

To analyze this impact methodically, we consider YouTube, one of the most popular online video streaming applications whose servers also support QUIC. We explore its performance over HTTP/3-supported browsers for three different parameters – the average playback bitrate, the temporal variation in the playback bitrate, and the average rebuffering or stall time. Notably, existing studies [6] have shown that these parameters significantly affect the video quality of experience (QoE).

We have streamed YouTube videos over Google Chrome/Chromium and Mozilla Firefox for 5474 streaming hours under three different network setups in the wild over the Internet, emulating the publicly available real dataset and different bandwidth patterns with controlled experiments. To

S. Chaudhary, N. K. Shukla, P. Sachdeva, and M. Maity are with the Indraprastha Institute of Information Technology (IIIT) Delhi, India 110020, Emails: {sapnac, naval19065, prince17080, mukulika}@iiitd.ac.in

S. Chakraborty is with the Department of Computer Science and Engineering, Indian Institute of Technology (IIT) Kharagpur, India 721302, Email: sandipc@cse.iitkgp.ac.in

the best of our knowledge, this is the first work that performs a large-scale study of media streaming performance over modern browsers that support HTTP/3 using a commercial end-point like YouTube. Interestingly, we observe that UDP rate-limiting middleboxes and congested networks show similar signatures, such as a high Round Trip Time (RTT) and a high rate of packet losses. As a result, the browsers' implementation of handling the transport protocols hinders getting the benefits of QUIC. Further, as the network quality fluctuates, the browser may initiate repeated switching between TCP and QUIC connections, thus affecting the performance of YouTube streaming.

Consequently, we argue that there is a need to revisit how browsers handle transport protocols underneath, irrespective of whether a middlebox blocks/rate-limits the UDP (hence, QUIC) or the network suffers from congestion. On the same lines, we present a dynamic solution designed by comparing the RTT experienced by both the TCP and the QUIC connections. The critical intuition in creating the solution is that even though a UDP rate-limiting middlebox and a poor network show a similar signature to browsers regarding RTT and stream errors, TCP remains unaffected by UDP rate-limiting middleboxes. Hence, the connection racing mechanism uses this as input to decide when to enable such racing. We implement this by modifying the Chromium source code and observe that it improves the QoE for more than 70% cases. In summary, our contributions to this paper are as follows.

**(1) Designing a novel testbed to study YouTube performance over HTTP/3 supported browsers:** Such testbed setup allows us to compare and evaluate the impact of browser configurations on adaptive bitrate streaming, where the performance also depends on the underlying network conditions.

**(2) Quantifying the impact of protocol stability on YouTube streaming performance:** 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, includes (i) emulation over different bandwidth patterns, (ii) in the wild collected data, and (iii) over real network traces. We observe that enabling HTTP/3 inside the browser does not help improve the QoE continually; instead, YouTube suffers over the HTTP/3 browser when network quality fluctuates or the bandwidth is low.

**(3) Evaluating application performance by proposing dynamic connection racing in the browser:** 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 in the Chromium browser to forcibly stop the connection racing if a protocol suffers due to a middlebox or 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. Next, we implement a dynamic solution for deciding when to enable connection racing. We made

this implementation and data open-sourced<sup>1</sup>. We evaluate our solution in poor networks and the face of UDP rate-limiting firewalls. We observe that it correctly enables connection racing in the case of a firewall and disables in the case of a poor network. Our solution improves the QoE compared to the original browser.

The rest of the paper is organized as follows. §II provides a broad overview of the existing literature. We then start by understanding how frequently switching between protocols is observed in reality by performing an in-the-wild study over four developing countries (§III). §IV explores how connection racing is implemented in Chromium and Mozilla Firefox browsers. §V presents our testbed setup. §VI demonstrates how connection racing affects streaming media QoE. §VII computes the correlation between connection racing and application QoE. §VIII deep dives into the root cause of poor performance of an application. Then, to establish the hypothesis, we modify the Chromium browser to disable connection racing altogether and analyze YouTube performance over this modified browser that transmits data purely over QUIC (§IX). Finally, we propose a dynamic solution for connection racing (§X). §XI concludes the paper by highlighting the limitations and future scopes in this direction.

## II. RELATED WORK

This section gives a broad overview of the prior work from two different perspectives as follows.

### A. Video Streaming over QUIC

In the seminal paper from Google on QUIC, Langley *et al.* [7] 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.* [8] have shown that QUIC congestion control needs to be tuned to support QoE for multimedia traffic. In the same line, a few other works [9], [10], [11], [12] in the literature have also adopted the QUIC protocol to support better streaming performance. Recently, Seufert *et al.* [13] 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 significant QoE improvement over 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 some of the QUIC streams were possibly discarded due to connection racing, as they had a significant amount of TCP packets. Shreedhar *et al.* [14] measured the QoE of YouTube videos using a headless Chrome browser. They found that QUIC experiences lesser stall than TCP/TLS when run over a network with 10% packet losses. Further, they observed that QUIC undergoes higher-quality switches than TCP in a lossy

<sup>1</sup><https://github.com/sapna2504/Chromium-Modification-V2>  
February 26, 2024)

(Access:

network. Kakhki *et al.* [15] conducted an in-depth analysis of QUIC and compared the performance of QUIC with the legacy protocol TCP. They observed that QUIC experiences fewer stalls, i.e., QUIC outperforms TCP only at higher video resolution. However, no significant improvement in QoE was observed at low-medium resolution compared to TCP. A few other works like [16], [17] proposed an improvement of QoE of video streaming by extending the QUIC protocol. Palmer *et al.* [17] suggested supporting unreliable streams in QUIC. They proposed an extended version of QUIC called *ClipStream* that offers reliability only on selected frames, i.e., I-Frames. They developed a prototype and observed that the modified QUIC provides better QoE than the original QUIC. Zheng *et al.* [16] 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 performed a large-scale study over short e-commerce videos and found *XLINK* helpful for video applications. *XLINK* can reduce the start-up delay and re-buffering rate of shorter videos. Such solutions are complementary to our study. Table I shows the comparison of prior work on video streaming over QUIC.

### B. Middleboxes and its impact on application QoE

Langley *et al.* [7] 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 subsequently blocking it. The authors realized the pain points of sustainability of a protocol across various middleboxes through “*deployment impossibility cycle*”. Hao Li *et al.* [3] highlight that updating the middleboxes for supporting emerging transport protocols is still tricky. 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. Although the prior work has looked at the impact of middleboxes on applications, none of the papers considered whether applications can get impacted unnecessarily even without a middlebox.

## III. MOTIVATION: FREQUENCY OF CONNECTION SWITCHING IN THE WILD DURING QUIC-TCP RACING

We start by performing a pilot measurement “*in-the-wild*” in low and middle-income countries to analyze how frequent connection 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 countries – India, Kenya, Pakistan, and Saudi Arabia. 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 min using a packet capture application named *PcapDroid*.

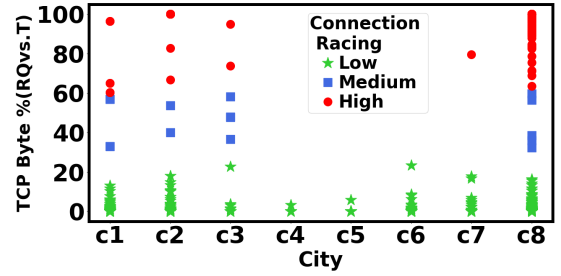


Figure 1: Percentage of TCP bytes ( $RQvs.T_v$ ) due to connection Racing over different cities where C1: Delhi, C2: Bangalore, C3: Kanpur, C4: Howrah and C5: Aligarh are of India, C6: Nairobi of Kenya, C7: Riyadh of Saudi Arabia and C8: Lahore of Pakistan. Percentage of TCP bytes ( $RQvs.T_v$ ): Low (0 – 30%), medium (30 – 60%), and high (60 – 100%)

To ensure packets of no other mobile applications are captured except YouTube, we use YouTube as the target app option of the *PcapDroid*. We collected a total of 56 such packet traces to analyze whether the presence of TCP bytes is common across various locations when QUIC is already enabled inside the YouTube app.

We quantify connection racing ( $RQvs.T_v$  i.e.,  $\underline{R}acing$  between  $\underline{Q}UIC$  and  $\underline{T}CP$ ) for a  $\underline{Q}_{ENB}$  streaming session ( $v$ ) as a ratio between bytes transferred only over TCP ( $T_v$ ) to the total bytes transferred over both QUIC ( $Q_v$ ) and TCP ( $T_v$ ). Notably, we consider the bytes only for packets that contain application data and ignore control packets.  $RQvs.T_v = \frac{T_v}{(T_v + Q_v)}$ .

Fig. 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 standard in scenarios where the network bandwidth is poor, or there is a variation in the bandwidth as is typically experienced during mobility. The volunteers reported poor application performance in terms of poor video quality and rebuffering while watching YouTube videos. Besides poor network impacting ABR (Adaptive Bitrate) decisions, we wondered whether unnecessary connection racing affects users’ QoE (Quality of Experience). We next explore this in further detail by understanding the implementation of connection racing in Chromium and Mozilla Firefox.

## IV. EXPLORING BROWSER’S IMPLEMENTATION

In this section, we discuss the implementation of connection racing in two popular browsers.

**Chromium Implementation [18]:** Fig. 2 shows the connection racing inside the Chromium browser. The browser first checks in the cache whether the YouTube server supports QUIC as an alternate service. If not, then the browser creates a job, namely *main job*, for initiating a TCP connection to enquire about QUIC support by the server. Such support

Table I: Existing work on video streaming over QUIC either use shorter videos or does not vary the bandwidth. Many of them did not explore the browser connection racing implementation and further root cause analysis.

	Root Cause Analysis	Network Bandwidth	Network RTT	Browsers	Explored QUIC Implementation	Video Duration	Application
Langley <i>et al.</i> [7]	✓	-	✓	Chrome/Chromium	✓	-	Video Streaming and Web Browsing
Seufert <i>et al.</i> [13]	×	1 Mbps	×	Chrome	×	180 sec	YouTube Video Streaming
Shreedhar <i>et al.</i> [14]	✓	20 - 100 Mbps	✓	Chrome	×	180 sec	Video Streaming and Web Browsing
Kakhki <i>et al.</i> [15]	✓	5 - 100Mbps	✓	Chrome	✓	60 sec	Web Browsing and limited Video Streaming
Palmer <i>et al.</i> [17]	×	20 Mbps	×	Chrome	×	296 sec	Video Streaming
Zheng <i>et al.</i> [16]	✓	20- 30 Mbps	✓	-	-	120 sec	Video Streaming
Our Work	✓	64Kbps - 1Mbps	✓	Chrome/Chromium and Firefox	✓	2700 sec - 4500 sec	Video Streaming

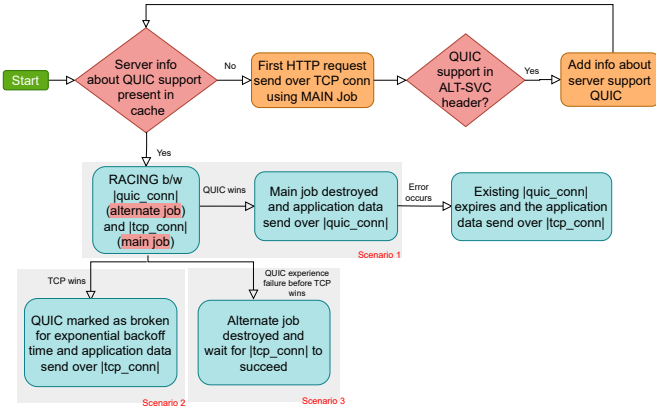


Figure 2: Connection Racing Implementation in Chromium

about alternative services is advertised by `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 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) [7], [19] 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, suppose the browser experiences a stream error [20] such as `QUIC_STREAM_CONNECT_ERROR` or `QUIC_STREAM_REQUEST_INCOMPLETE` while associating the data stream with the `alternate` job. In that case, it waits for the TCP connection to succeed and then binds the application data with the `main` job. Further, if, for any reason, the browser (idle application, poor bandwidth, etc.) does not communicate with the server for an idle timeout duration (default 29 seconds),

then the existing QUIC connection expires. The next time the browser wants to communicate with the server, it must create a new QUIC connection and repeat all the above steps.

**Firefox Implementation [21]:** In Firefox, a browser gets to know about QUIC support either via `alt-svc` header or `HTTPSSVC` DNS record. If the YouTube server supports QUIC, 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 creates or reuses an existing backup connection (TCP). Once the backup connection is ready, the browser immediately shifts the transaction to that 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, the browser adds the server domain to a HTTP/3 excluded list that it maintains. Hence, this server will not establish the QUIC connection unless the browser window is closed and the list gets reset.

## V. MEASUREMENT METHODOLOGY

To answer whether unnecessary connection racing impacts the QoE of users, we follow the following measurement methodology. Instead of collecting the data directly over an Android YouTube app, we resorted to a browser-based emulation platform for the following reasons. (i) The ABR streaming parameters can be accessed from the HTTP headers of the YouTube video requests [22]. However, the YouTube Android app does not provide a logging extension and comes up with its security certificates. Therefore, tools like MITM proxy do not work to intercept YouTube traffic in between. (ii) Android browsers, like mobile Chrome, do not provide a logging extension to log HTTP headers. (iii) Most MITM proxy applications do not support QUIC extension yet [23].

### A. Experimental Setup

Fig. 3 shows the overall testbed setup. The YouTube application running over a desktop browser enables us to log the streaming events from the browser's logging APIs. Hence, the broad idea is to stream YouTube videos over a desktop browser. While streaming the videos, we emulate the network behavior through a benchmark traffic shaper *mahimahi*. We emulate the following two network behaviors: (1) *controlled*:



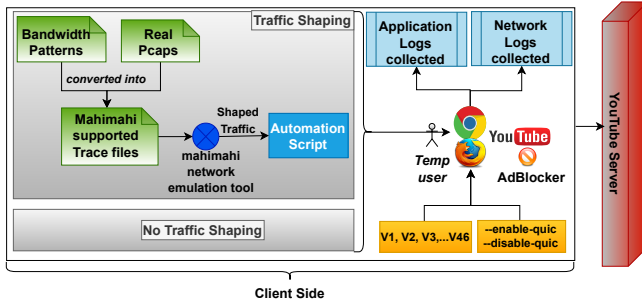


Figure 3: Testbed setup for YouTube performance analysis over Google Chrome and Mozilla Firefox

emulate various bandwidth patterns, (2) *semi-controlled*: emulate various real traces collected under mobility in WiFi and cellular networks. Note that *mahimahi* network shaper emulates a network behavior using a trace file. We next discuss different components of this setup. We conduct the experiments using 46 videos selected from YouTube, each with a duration of 45 mins to 1 hour 15 mins. The videos are of different genres, such as *News*, *Entertainment shows*, *Education*, *Talk shows*, *Comedy*, *Stanford online lectures*, *British TV series*, etc. The maximum quality available for these videos is 720p, and the minimum is 144p. We played videos across different geographical locations: Delhi, Bangalore, New York, Germany, and Singapore. One of the testbeds is set up inside our campus premises, and the rest are set up using Digital Ocean machines, each having *x86\_64* architecture with the processor running at 2494.136 MHz frequency, 4 GB RAM and with Ubuntu 18.04 OS.

**Browser Setup:** YouTube app uses Cronet [24], a chromium networking stack made available to Android apps as a library. Google Chrome also uses the same library; hence, we use it to stream YouTube videos on the desktop platform. In addition, we stream over Firefox as well. We have used an I-Frame SRC URL [25] for embedding a YouTube player inside the browsers by creating an element of I-Frame and then appending the video ID at the end of it. We set the auto-play parameter inside this I-Frame to play the video automatically in Chrome/Chromium. In Mozilla Firefox, we set the preference for autoplay: `media.autoplay.default` to 0 to allow audio and video autoplay (disabled by default). To ensure that video is not played from the cache, we created a new temporary user to open a new browser window after clearing the cache (such as opening a new window in incognito mode for private browsing). Therefore, a browser window opens and starts running a particular video directly from YouTube. Both browsers support the QUIC protocol stack. In the Chrome/Chromium browser setup, we enable QUIC by setting the flag `--enable-quic`. For Firefox, we enable QUIC by setting the `network.http.http3.enable` to `true`. (ii) `network.http.http3.enable_0rtt`, (iii) `network.http.http3.priority`, (iv) `network.http.http3.support_version1`, and (v) `network.http.http3.enabled`. During this setup, we ensured that the QUIC packet sizes were smaller than the path MTU. Additionally, we used an ad-block extension to

remove advertisements during the video playback.

**Network Setup:** We use *Mahimahi*, a network emulation tool Link Shell (`mm-link`) to emulate specific network behavior [26]. It uses a user-defined packet delivery trace file to emulate a network. We create two types of network setups: (1) *semi-controlled setup* where we replay a total of 161 packet traces. Out of which 56 traces are from our volunteers, and for the rest, we use traces available on public sources [27]. This dataset is collected while streaming video over YouTube through WiFi on a laptop. The collected dataset contains both when the user was stationary and mobile. From this, we used 105 mobility traces. We converted the pcap traces to Mahimahi-supported trace files [6]. (2) *controlled setup*: where we emulate various bandwidth patterns. We start with a static bandwidth pattern to understand the minimum and maximum bandwidth at which the minimum and maximum video quality can be attained. At  $< 128$  Kbps, the lowest video quality of 144p was observed throughout the video; at  $> 1500$  Kbps, the quality level of 720p was observed. Each pattern initiates a loop from an initial bandwidth, jumps to the next bandwidth level, and maintains it for a duration, which goes up to the final bandwidth as mentioned in Table II. 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 and the packet size (which equals 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 quick check using parallel streaming without applying any traffic shaping, as shown in Fig. 3. Such a setup always attains 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 setup. This IP belongs to NKN (National Knowledge Network) of Delhi, which maintains a local cache of Google. This way, we ensured no other parameter except the network played a role in the performance measurement.

Table II: Bandwidth Patterns used for Network Emulation

Emulated Bandwidth Patterns	Initial	Final	Jump	Jump Dur. (sec)
<i>Dynamic High (DH)</i>	1152Kbps	896Kbps	-256	240
<i>Dynamic Low (DL)</i>	640Kbps	128Kbps	-256	240
<i>Dynamic Very Low (DVL)</i>	64Kbps	256Kbps	+64	60
<i>Semi-controlled (Real)</i>	"in-the-wild" data and mobility traces [27]			

**Log Collection Setup:** We collected logs at two layers: the *application layer* and the *network layer*. For the application layer, we downloaded all the video-related information from the requests made by YouTube clients and the responses received. Since the connections were encrypted, we created

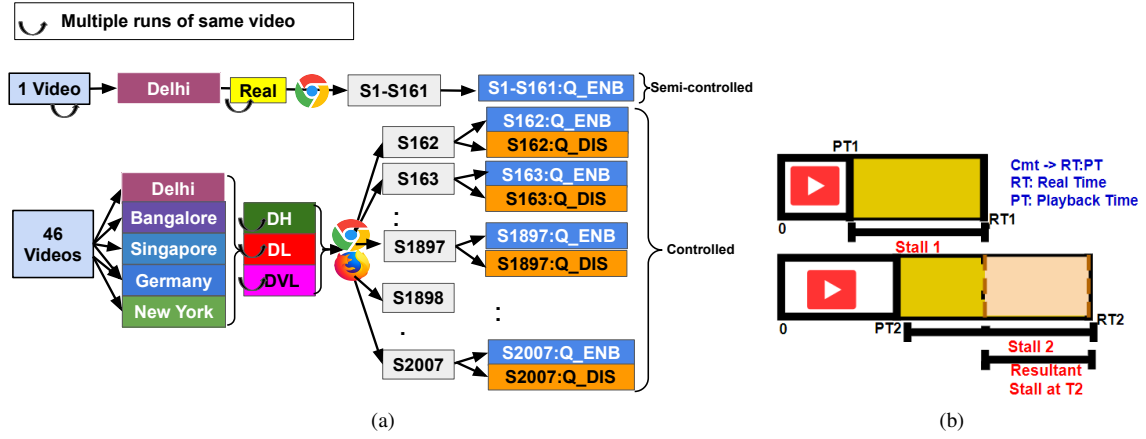


Figure 4: (a) Dataset Organization across 5 locations, 2 browsers, and 4 bandwidth patterns, (b) Stall Computation

our application log extension. Inside it, we have used the `console.log` API for the logs. We defined four events – `BEFORE_REQUEST` (YouTube client prepares an HTTP Request), `SEND_HEADERS` (YouTube client sends HTTP Request which contains various DASH parameters), `RESPONSE_STARTED` (YouTube client starts receiving the response) and `COMPLETE` (YouTube client completes receiving the response, including the video segment data). These events log all the HTTP Request messages sent from the browser to the server. For this event log, we observe two different requests – a `videoplayback` request (request URL is like `https://r6---sn-o3o-qxal.googlevideo.com/videoplayback`) and a `qoe` request, (Request URL is like `https://www.youtube.com/api/stats/qoe?...)` where both are HTTP POST Requests. 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 the protocol used (QUIC). Once the YouTube server receives this request, it sends the corresponding HTTP Response and the segment data.

On the contrary, the YouTube client periodically sends the `qoe` request to the server on triggering of the event `streamingstat` triggered after a predefined number of frames are rendered over the client. This POST request 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 (tells the requested video quality), *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 two statistics *itag* and *cmt*, to determine when and at what quality a frame has been downloaded and when the frame has been played. In Mozilla Firefox, to run our extension, we have used the command-line tool `web-ext` with `-verbose` flag to print the logs in the terminal along with the follow-

ing preferences: `devtools.console.stdout.chrome` and `devtools.console.stdout.content` set to true, which prints all the logs on the terminal.

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. In addition to the application logs, we collected packet traces to get the network-level exchanges, such as the total number of TCP and QUIC bytes.

**Dataset Organization:** Fig. 4(a) represents our dataset. We streamed a single video multiple times over a QUIC-enabled (Q\_ENB) Chrome/Chromium browser setup for a semi-controlled setup. We obtain a dataset 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 (see Table III).

Table III: Dataset; #sessions: 3853, #duration: 3943 hours

Type	Hours	Type	Hours
Q_DIS	1877	Q_ENB	2066
DH	278	DL	1712
DVL	1650	Firefox	142
Semi-controlled	161	Delhi	3922
Bangalore	490	Singapore	253
Germany	459	New York	218

### B. QoE Metrics used for Evaluation

We characterize the QoE via three performance counters: average bitrate of a video (Bitrate), average bitrate variation

(V\_BRate), and 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 compute the bitrate variation by finding the difference between the previous and current bitrate. To compute stall, we use the *cmt* parameter of the streaming stat. We subtract the playback timestamp from the real timestamp (Fig. 4(b)), which gives the cumulative stall at any 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.

## VI. APPLICATION PERFORMANCE VERSUS CONNECTION RACING: HOW THIS AFFECTS STREAMING QOE

In this section, we study the impact of connection racing on application QoE. First, we investigate the relation between  $RQvs.T_v$  and streaming performance. Then, we study the impact of each configuration in our setup.

### A. Application QoE vs Connection Racing

To study the temporal pattern of the video's QoE with  $RQvs.T_v$  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 provides information in an asynchronous manner. Therefore, we define two derived metrics from the application logs: quality drop ( $B$ ) duration and stall ( $S$ ) duration. 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 summation of all the stalls experienced in that time window of 30-sec.

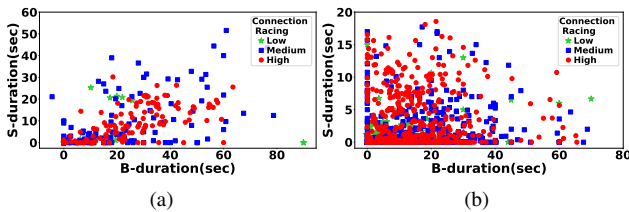


Figure 5: Scatter plot of Quality drop ( $B$ ) duration and Stall duration ( $S$ ) (seconds) at different protocol switching extent due to connection racing (low ( $< 30\%$ ), medium ( $30 - 60\%$ ) and high ( $> 60\%$ )) in (a) semi-controlled (b) controlled setup

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. 5(a), (b) show a scatterplot of quality drop ( $B$ ) duration and stall ( $S$ ) duration for three different protocol switching levels in a semi-controlled and controlled setup respectively. Under both semi-controlled and controlled setups, we observe that as the presence of TCP bytes increases from low to high, the quality drop and stalling duration also increase. Specifically, for the semi-controlled setup, we observe that the median quality drop duration increases from 11s to 25s, and stall duration increases from 1s to 8s from low to high protocol switching. Similarly, median quality drop duration for controlled setup increases from 6s to 13s, and stall duration increases from 0.3s to 1.3s from low to high protocol switching. If the videos were streamed at the lowest quality (as it is done in DVL), the stalling instances are observed more. 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 setup but it might be possible that connection racing adds extra stalls in the case of Q\_ENB browser. Fig. 5 shows that medium to high connection racing events are related either to stalling or quality drop, which made us wonder whether connection racing (apart from the poor/variable network condition) affected the application in terms of its QoE or if it was because of some specific configuration settings in our setup. Hence, we conduct thorough controlled experiments to analyze the impending factors behind quality drops and stalls observed with medium to high connection switching occurrences.

### B. How Connection Racing Affects Streaming QoE

Now, to isolate the impact of poor network, we compare the YouTube performance from two different browser setups, QUIC-enabled (Q\_ENB) and QUIC-disabled (Q\_DIS) browsers for Chrome/Chromium and Mozilla Firefox. We compute the three metrics using the *streaming-stat* information discussed in §. V. 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, resulting the instantaneous moving average value for each QoE metric for each video streaming session. Note that an averaging approach of computing a single value for these three metrics for the entire session is not a good 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. We consider case (b) as the null hypothesis and then perform a two-tail test [28] to check which stream performs better when the alternative hypothesis is true. **Streaming Quality vs Browser Setup:** For Chrome/Chromium, the null hypothesis was accepted for 15%, 30%, and 16% cases in case of average bitrate, average bitrate variation and average stall respectively. For Firefox, the null hypothesis was accepted for 23%, 52%, and 28% cases.



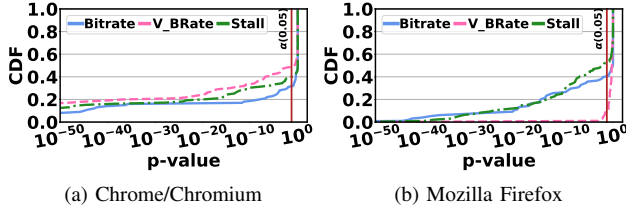


Figure 6: Hypothesis test result for various QoE metrics

For the cases where the null hypothesis was rejected, we performed the two-tail test. Fig. 6(a) and (b) show CDF of p-values observed over the two-tail test for the cases when YouTube over 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. Further, Q\_ENB ones experience higher stall compared to the Q\_DIS ones for 40% (for Chrome) and 52% (for Firefox) of the instances. This raises the question of why YouTube's performance over Q\_ENB browsers suffers.

**Impact of Bandwidth Patterns:** We perform hypothesis tests across data collected over different bandwidths for each QoE performance metric. Fig. 7 (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. Further, for the cases where the null hypothesis was rejected for DH, we observe that YouTube over Q\_ENB browsers performs poorer than Q\_DIS ones in terms of average bitrate, average bitrate variation, and stall for 19%, 17%, and 11% cases, respectively, combined over both the 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 lesser.

**Impact of Geographical Locations:** To answer this, we test the hypothesis on data collected in 5 different locations, namely 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 null hypothesis was rejected, we performed a two-tail test. Fig. 8 (a) shows the CDF plot of hypothesis-testing result for 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. 8 (b)), the similar percentages are 26%, 28%, 30%, 32%

and 24% cases, respectively. In terms of stall (shown in Fig. 8 (c)) the percentages are 49%, 43%, 43%, 46% and 43% cases, respectively. Hence, our observations remain consistent across various locations as well.

We also analyzed whether our observations remain consistent across video genre and the entire data collection duration over the 14 months from September 2021 to December 2022. We find that our observations of YouTube's performance suffer more over Q\_ENB browsers than Q\_DIS browsers remain consistent across these two verticals as well.

**Takeaway 1:** We observed that higher the connection switching occurrences between QUIC and TCP during the racing, the poorer the application QoE. Hence, 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 for the former than the latter. Further, these findings are statistically consistent across browsers, bandwidth patterns, locations, time, and video genres.

## VII. CORRELATION BETWEEN APPLICATION QoE AND CONNECTION RACING INSTANCES

From section VI 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 §. VI (Fig. 6), 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  $\tau(f_t)$  be the playback time for the frame  $f_t$  i.e., the time when frame  $f_t$  is rendered/played and  $\sigma(f_t)$  be the download time for the frame  $f_t$  i.e., 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 §. VI-A ( $144p \rightarrow 1$ ,  $240p \rightarrow 2$ ,  $360p \rightarrow 3$ ,  $480p$



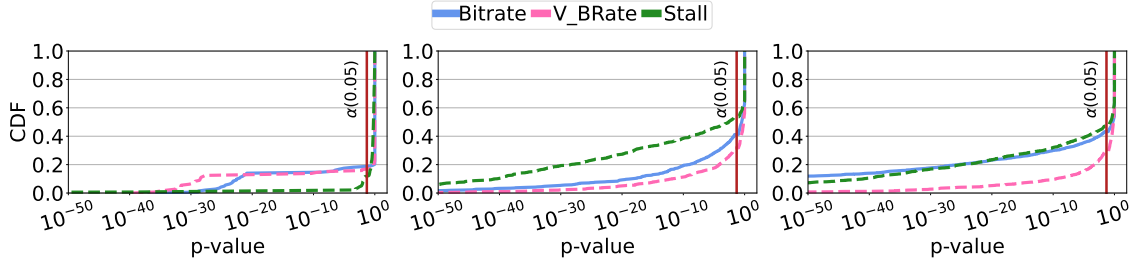


Figure 7: Hypothesis test results for different bandwidth patterns: (a) DH, (b) DL, (c) DVL

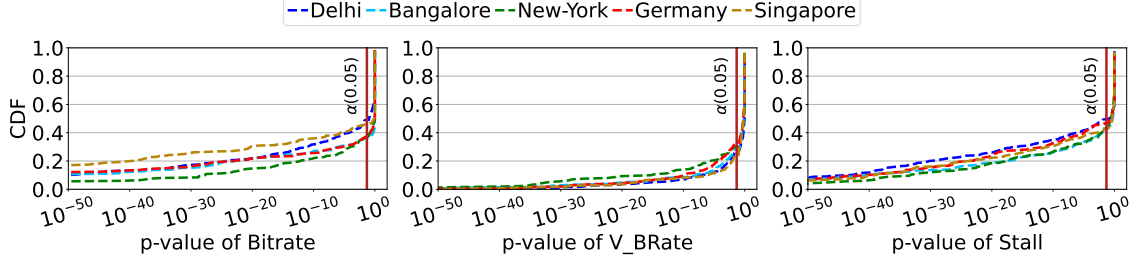


Figure 8: Hypothesis test results for different geographical locations

$\rightarrow 4$ , 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. 4(b) (in §. V).

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

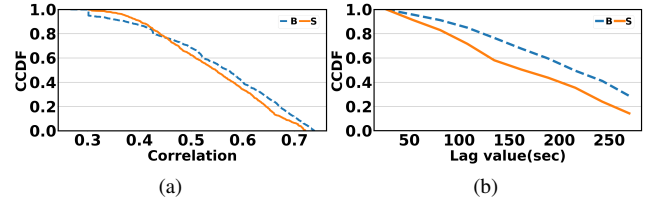
$$(\mathcal{RQvs.T}_v * \mathcal{B}_v)(\mu_b) = \int_{-\infty}^{\infty} \overline{\mathcal{RQvs.T}_v(t)} \mathcal{B}_v(t + \mu_b) dt \quad (1)$$

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

$$(\mathcal{RQvs.T}_v * \mathcal{S}_v)(\mu_s) = \int_{-\infty}^{\infty} \overline{\mathcal{RQvs.T}_v(t)} \mathcal{S}_v(t + \mu_s) dt \quad (2)$$

where  $\mu_s$  is the corresponding lag. Similar to the above, we compute the  $\mu_s$  for which we obtain the maximum cross-correlation ( $\mu_s^{max}$ ). We next plot the CCDF distributions of correlation and the lag values in Fig. 9.

Fig. 9(a) shows the correlation between connection racing ( $\mathcal{RQvs.T}_v(t)$ ) and quality drop ( $\mathcal{B}_v(t)$ ) as well as stalling ( $\mathcal{S}_v(t)$ ). Similarly, Fig. 9(b) shows the lag values  $\mu_b^{max}$  and  $\mu_s^{max}$ . 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. 9(b) that for both  $\mathcal{B}_v(t)$  and  $\mathcal{S}_v(t)$ , the lag values across different streaming sessions are

Figure 9: (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

distributed within 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.

## VIII. DELVING INTO THE DEPTH: WHY CONNECTION RACING AFFECTS STREAMING QoE

In this section, we perform a root cause analysis to understand why connection racing happens even without a middle-box blocking or rate-limiting UDP.

### A. Temporal Analysis of Connection Racing Events and Corresponding YouTube QoE

Here we cherry-pick a few instances from our collected data, as discussed earlier, and perform a temporal analysis to observe the impact on YouTube QoE when connection racing occurs. Fig. 10(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. 10(b) shows one sample server from which the client received the maximum data. Note that the presence of TCP packets in the initial part

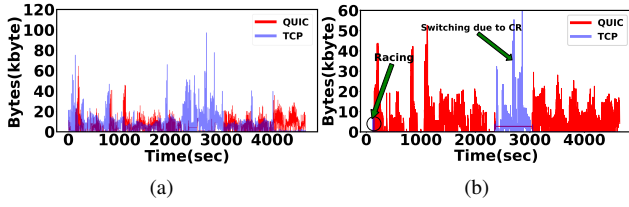


Figure 10: Connection Racing observed over Chrome Browser in terms of bytes transferred over individual protocols, for a YouTube streaming over Q\_ENB browser, (a) Across all IPs from where the client received streaming data, (b) For a single IP on which 89% of total bytes were transferred.

is inevitable. However, if QUIC wins the race, the presence of TCP packets will likely be small. Interestingly, the figure shows a significant presence of TCP, indicating frequent connection switching events between QUIC to TCP and then TCP to QUIC. It also shows the protocol instability or fluctuations where the session starts with TCP, moves to QUIC, then switches to TCP again, and then moves to QUIC. Fig. 10(b) shows that the first two HTTP requests were transmitted over TCP, QUIC wins the race, and all the subsequent requests are transmitted over QUIC. Later at 2200 sec, some error occurs in the stream, either in the existing stream or while creating a new stream. Hence, the browser waits for the QUIC connection to succeed and sends data over TCP.

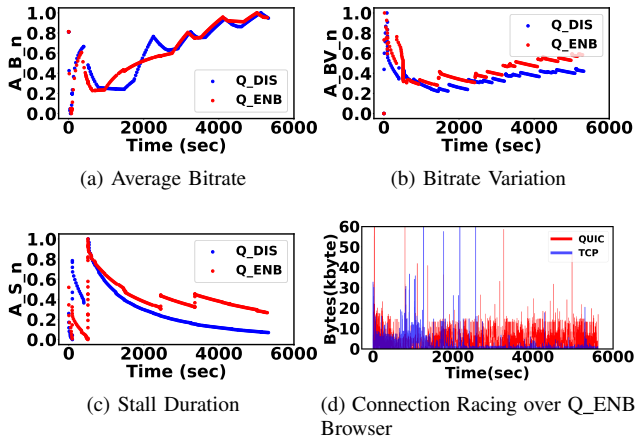


Figure 11: Temporal performance for Scenario S1: YouTube performs better over a Q\_DIS browser

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 the stall values over time for the cherry-picked two pairs of sample video streaming sessions over 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. 11(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. 11(b)) and stalling (Fig. 11(c)), YouTube performs better over the Q\_DIS browser. In the second case (S2), we observe that

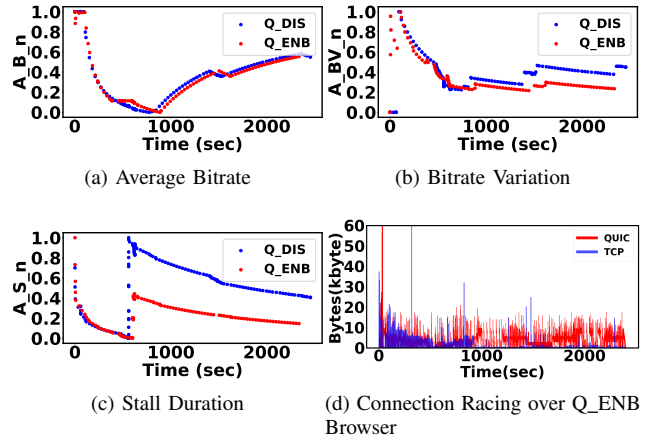


Figure 12: Temporal performance for Scenario S2: YouTube performs better over a Q\_ENB browser

YouTube performs better over the Q\_ENB browser (with  $p < 0.05$ ) for all the three QoE parameters – average bitrate (Fig. 12(a)), bitrate variation (Fig. 12(b)) and stall duration (Fig. 12(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 the TCP bytes only. Interestingly, we observe that TCP traffic is less in S2 (Fig. 12(d)) than in S1 (Fig. 11(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 over 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, introduce network instability, affecting the streaming QoE. Next, we investigate the relation between network and connection racing events.

## B. Temporal Analysis of Connection Racing Events and Network Parameters

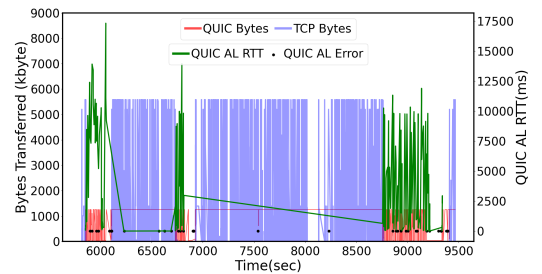


Figure 13: Chromium: Temporal analysis for Application events and network RTT: QUIC application layer(AL) errors, and RTT with bytes transferred from a single IP: 172.217.138.105 over TCP and QUIC over a QUIC-enabled session

The network parameters such as RTT and throughput are utilized to characterize the network from the browser. 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 [29], [30] and, (2)

by monitoring RTT values used by the congestion control algorithm. Since throughput computations are done over a time window, throughput values can provide stale estimates in case of varying network bandwidth (we are emulating the same). 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 QUIC congestion control algorithm. We instrument the browser to get these values. Fig. 13 shows an enlarged view of 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 into from 6000 sec to 9500 sec for an IP: 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* occurs. 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 (\$IV). 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. 13). 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 is continuously high or varying. If the RTT is varying, there will be fewer QUIC retry connection establishment failures which will lead to low-medium connection racing. Otherwise, if QUIC RTT is more than a threshold  $\delta$  ms, then there will be more QUIC connection establishment retry failures and lead to more connection racing. In the sample examples,  $\delta$  is 9000 – 10000. After looking at such individual examples, we perform analysis on an aggregate basis to investigate how the network impacts connection racing instances.

### C. Characterizing Connection Racing Across Networks

Fig. 14 shows the complementary CDF (CCDF) of connection racing ( $RQ_{vs.T_v}$ ) 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, there are fewer connection failures of QUIC. For *DL*, *DVL* and *Real*, 40% or more bytes are transferred over TCP for 20%, 40%

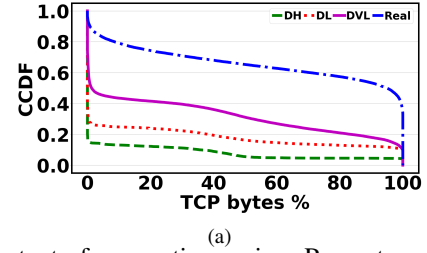


Figure 14: xtent of connection racing: Percentage bytes transferred over TCP across all YouTube over QUIC-Enabled Browser (Q\_ENB) sessions for various network scenarios

and 70% times, respectively. Hence, we observed that Q\_ENB browsers tend to involve in more connection switching events at a low or variable bandwidth network. This observation is also significant over the real network traces. Interestingly, the relative amount of connection switching events correlate 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 can not distinguish between a middlebox and poor network as both of them lead to QUIC connection failure/suffering. Hence, browsers do connection racing unnecessarily.

**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.

## IX. YOUTUBE OVER A PURE QUIC SESSION

We modify the Chromium source<sup>2</sup> 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: February 26, 2024). We perform similar experiments under identical network and traffic conditions over the original Chromium browser and the modified one to understand whether a pure QUIC session helps improve the application performance under the scenarios when the application suffered over the original Chromium browser.

### A. 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 like handshake

<sup>2</sup><https://www.chromium.org/Home/> (Accessed: February 26, 2024)

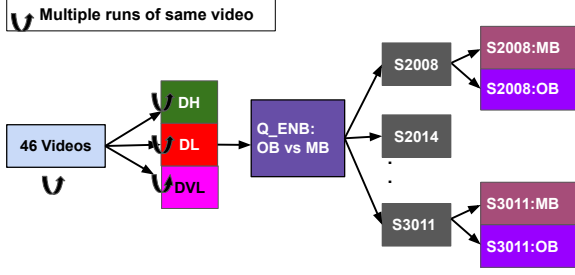


Figure 15: Dataset organization of YouTube over QUIC-enabled modified browser (MB) and original browser (OB)

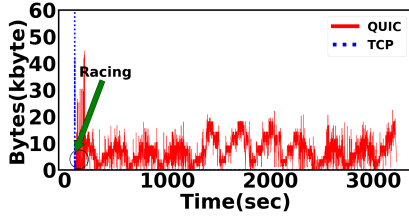


Figure 16: Results of Chromium browser; Number of bytes transferred from a single IP: 180.149.59.14 over TCP and QUIC over a QUIC-enabled modified browser (MB)

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 cant 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 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. 15 shows the dataset of 1004 steaming session pairs, named as *S2008, S2009, ..., S3011*. Each pair contains one session over YouTube with QUIC-enabled original browser (OB) and QUIC-enabled modified browser (MB). The total duration is  $\simeq 1453$  hours ( $\simeq 61$  days), of which original browser is 738 hours, and modified browser is 715 hours.

## B. Results

Fig. 16 shows the negligible presence of TCP traffic except only in the beginning, only used to investigate for QUIC support. All the modifications we made restricted the browser from using the `main` job. Hence, the browser allows QUIC to utilize its true potential and stop unnecessary connection racing. We then look at sample video sessions to observe how the modified browser impacts application QoE compared to the original case. Fig. 17(a, b, and c) shows the distribution

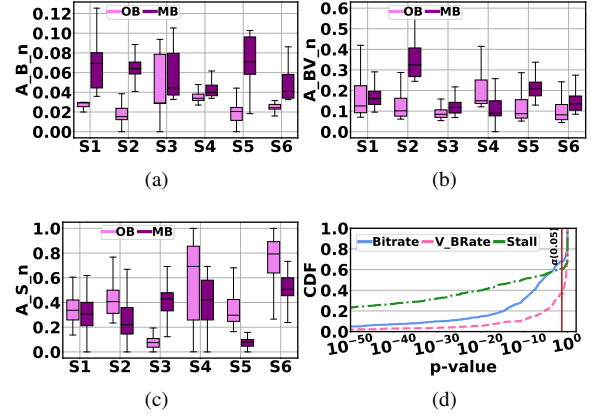


Figure 17: Normalized (a) average bitrate, (b) average bitrate variation and (c) average stall distribution of 6 sample streaming pairs for original vs modified Chromium browser (d) Hypothesis testing on original vs modified Chromium browser

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

Fig. 17 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.

## X. INTELLIGENT CONNECTION RACING MECHANISM

We develop a dynamic solution to take a connection racing decision. Our solution is based upon our observations in §VIII-B where we observed that connection racing is highly correlated with the RTT. We perform experiments under poor network and UDP rate limiting firewall over the original Chromium browser and the modified one to understand whether our solution helps improve the application performance under the scenarios when the application suffered over the original Chromium browser.

### A. Dynamic-solution

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



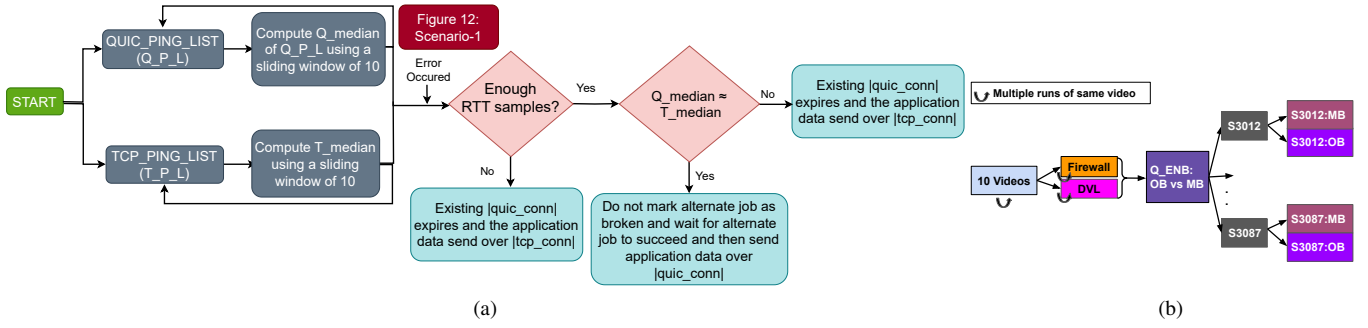


Figure 18: (a) Dynamic connection racing solution work flow (b) Dataset organization of YouTube over QUIC-enabled modified Chromium browser (MB) and original Chromium-browser (OB)

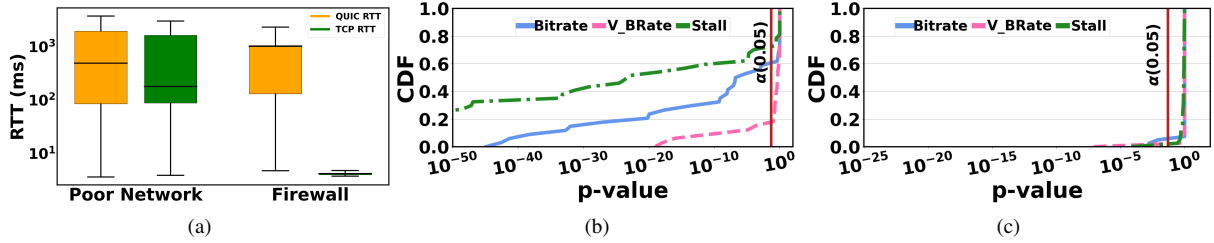


Figure 19: (a) QUIC and TCP RTT under poor network with 170 QUIC RTT samples and 140 TCP RTT samples and rate-limit UDP:443 firewall with 171 QUIC RTT samples and 143 TCP RTT samples. (b) Poor Network: Hypothesis test result for average bitrate, average bitrate variation and average stall of 38 sample streaming pairs for original vs modified browser (c) 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

the following: *we monitor the RTT of QUIC and TCP at the browser. If comparable, indicates that the network is providing the same treatment to both protocols and we should not allow connection racing.* On the other hand, if not comparable and if QUIC's RTT is higher than TCP's that indicates the network is not providing the same treatment to both, specifically QUIC is facing additional challenges, hence we allow connection racing in that case.

Fig. 18 (a) shows the design of our solution. We modify the Chromium source<sup>3</sup> to implement the dynamic solution. We have made this also open-source, which can be accessed from the following link – <https://github.com/sapna2504/Chromium-Modification-V2> (Accessed: February 26, 2024). We monitor the RTT of QUIC and TCP using NQE (Network Quality Estimator) [29]. NQE transmits QUIC frames to obtain QUIC RTT value and monitors the existing TCP sockets to obtain TCP RTT. As the mechanisms used for getting RTTs are not the same (active for QUIC and passive for TCP) and not 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 ping once every 15 sec to reduce overhead due to ping in a poor network. Whenever the QUIC connection is established the browser sends the very first ping for a connection migration check. We utilize this first ping to set further alarms (at every 15 sec) for future pings. The QUIC RTT was computed by subtracting the packet

sent time from the received time (it provides accurate network RTT estimation [7]). Similarly, we used the HTTP/2 ping mechanism for TCP that transmits TCP pings every 15 seconds using *HttpStreamFactory*. We enable this HTTP/2 ping mechanism only for the sessions attached to the YouTube server by checking a hostname. It allows to enable and disable 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 sec. 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 ping mechanism before sending 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

<sup>3</sup><https://www.chromium.org/Home/> (Accessed: February 26, 2024)

and/or timeout(s) we compare the RTT values. Till the point, we don't have enough RTT samples to make a decision, we allow the browser to behave as it would have otherwise.

### B. Dataset & Results

**Dataset:** We stream YouTube videos under a poor network (DVL) and UDP rate-limiting firewall. Note that we have rate-limited the UDP packets at port 443 with a rate of 64Kbps 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 18 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. 19(a) shows the RTT distribution of QUIC and TCP under poor network and rate-limiting firewall. We observe that QUIC and TCP RTT values are similar for poor networks. However, the former's RTT values are higher than the latter for firewalls. We have used this key observation for developing our solution.

**Comparison of QoE** Fig. 19(b) and (c) show the result of hypothesis testing for poor network and UDP rate limited firewall respectively. For 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.

## XI. DISCUSSION AND CONCLUSION

During an HTTP/3 session, we observed repeated protocol switching due to connection racing, hurting the application performance. This fluctuation between protocols is alarming for the streaming session. Finally, our analysis over more than 5474 streaming hours of video data can provide a reliable indication of the impact of connection racing on YouTube over QUIC-enabled browsers' performance from an application perspective. As the application performance gets affected due to the connection racing implementation on the browser, which is independent of what application is run on top of it, we believe that similar observations will be applicable for 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

a failure caused by a middlebox and that due to network congestion, which needs a thorough investigation to support a stable performance of QUIC over the Internet. The sheer variance in the real world makes it impossible to measure precisely. Though we have tried to emulate the real world to understand what happens underneath 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. Further, even when connection racing is manually turned off after modifying the browser, still, for about 40% instances, the original browser outperformed the modified browser. Therefore, further research is required to determine the other reasons that affect the QUIC performance.

**Improving the performance of connection racing:** Another orthogonal solution could be to let the connecting racing happen as it is but rather let the sender be aware of such protocol switching. The sender would first detect when a connection racing happens at the browser level and tunes its parameters to tolerate the transfer of connections from QUIC to TCP or vice versa. Likewise, whenever a protocol switching occurs due to racing, the state of the protocol also gets transferred. For example, suppose the protocol switches to TCP during streaming over QUIC due to a middlebox or poor network. During that time, TCP's congestion state should not start from the beginning; instead, whatever congestion state QUIC had reached could be passed to TCP and vice versa when the browser decides to return to QUIC. Through this mechanism, even if protocol switching happens, the application data transfer rate would remain intact as the sender's congestion state does not change.

## ACKNOWLEDGEMENT

This work is funded by the Science and Engineering Research Board (SERB), India through the Grant Number CRG/2022/004187. We thank Mr. Chirag Bansal for helping us implement the dynamic racing solution in Chromium.

## REFERENCES

- [1] B. T. M. Kuehlewind, "Applicability of the QUIC Transport Protocol," <https://www.ietf.org/archive/id/draft-ietf-quic-applicability-09.html>, 2021, [accessed February 26, 2024].
- [2] M. Stucchi, "Mongolia Joins Growing Number of Countries Reducing Openness and Resilience of the Internet," <https://rb.gy/zyolpk>, 2020, [Online; accessed February 26, 2024].
- [3] H. Li, C. Wu, G. Sun, P. Zhang, D. Shan, T. Pan, and C. Hu, "Programming network stack for middleboxes with rubik." in *NSDI*, 2021.
- [4] T. Böttger, F. Cuadrado, G. Tyson, I. Castro, and S. Uhlig, "Open connect everywhere: A glimpse at the internet ecosystem through the lens of the netflix cdn," *ACM SIGCOMM Computer Communication Review*, 2018.
- [5] K. MacMillan, T. Mangla, J. Saxon, and N. Feamster, "Measuring the performance and network utilization of popular video conferencing applications," in *Proceedings of the 21st ACM Internet Measurement Conference*, 2021.

- [6] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensive," in *Proceedings of the conference of the ACM special interest group on data communication*, 2017.
- [7] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. 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.
- [8] M. Engelbart and J. Ott, "Congestion control for real-time media over quic," in *Proceedings of the 2021 Workshop on Evolution, Performance and Interoperability of QUIC*.
- [9] S. Arisu and A. C. Begen, "Quickly starting media streams using quic," in *Proceedings of the 23rd Packet Video Workshop*, 2018.
- [10] C. Perkins and J. Ott, "Real-time audio-visual media transport over QUIC," in *Proceedings of EPIQ'20*.
- [11] J. Herbots, M. Wijnants, W. Lamotte, and P. Quax, "Cross-layer metrics sharing for quicker video streaming," in *Proceedings of CoNEXT'20*.
- [12] D. Lorenzi, M. Nguyen, F. Tashtarian, S. Milani, H. Hellwagner, and C. 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*.
- [13] M. Seufert, R. Schatz, N. Wehner, and P. Casas, "Quicker or not?-an empirical analysis of quic vs tcp for video streaming qoe provisioning," in *2019 22nd Conference on ICIN*.
- [14] T. Shreedhar, R. Panda, S. Podanev, and V. Bajpai, "Evaluating quic performance over web, cloud storage and video workloads," *IEEE Transactions on Network and Service Management*, 2021.
- [15] A. M. Kakhki, S. Jero, D. Choffnes, C. Nita-Rotaru, and A. 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*, 2017, pp. 290–303.
- [16] Z. Zheng, Y. Ma, Y. Liu, F. Yang, Z. Li, Y. Zhang, J. Zhang, W. Shi, W. Chen, D. 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.
- [17] M. Palmer, T. Krüger, B. Chandrasekaran, and A. Feldmann, "The quic fix for optimal video streaming," in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, 2018, pp. 43–49.
- [18] Chromium, "Chromium Code Search," <https://source.chromium.org/chromium/chromium/src>, [Online; accessed February 26, 2024].
- [19] R. Hamilton, "Re: [QUIC] graceful fallback to tcp," <https://mailarchive.ietf.org/arch/msg/quic/ph1IAVBa5pW1AgDvr8fbD741Auwl/>, 2016, [Online; accessed February 26, 2024].
- [20] C. C. Search, "QUIC error codes," [https://source.chromium.org/chromium/chromium/src/+HEAD:net/third\\_party/quiche/src/quiche/quic/core/quic\\_error\\_codes.h](https://source.chromium.org/chromium/chromium/src/+HEAD:net/third_party/quiche/src/quiche/quic/core/quic_error_codes.h), 2022, [Online; accessed February 26, 2024].
- [21] Firefox, "Mozilla-Central," <https://searchfox.org/mozilla-central/source/netwerk>, [Online; accessed February 26, 2024].
- [22] A. Mondal, S. Sengupta, B. R. Reddy, M. Koundinya, C. Govindarajan, P. De, N. Ganguly, and S. Chakraborty, "Candid with youtube: Adaptive streaming behavior and implications on data consumption," in *NOSSDAV'17*, pp. 19–24.
- [23] T. Perry, "One Port to Rule Them All," <https://httptoolkit.com/blog/http-https-same-port/#wait-what-about-http3>, 2021, [Online; accessed February 26, 2024].
- [24] Developers, "Perform network operations using Cronet," <https://developer.android.com/guide/topics/connectivity/cronet>, 2021, [Online; accessed February 26, 2024].
- [25] YouTube, "YouTube Embedded Players and Player Parameters," [https://developers.google.com/youtube/player\\_parameters](https://developers.google.com/youtube/player_parameters), 2021, [Online; accessed February 26, 2024].
- [26] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan, "Mahimahi: Accurate Record-and-Replay for HTTP," in *2015 USENIX Annual Technical Conference*.
- [27] C. Gutterman, K. Guo, S. Arora, X. Wang, L. Wu, E. Katz-Bassett, and G. Zussman, "Requet Dataset," <https://github.com/Wimnet/RequetDataSet>, 2019, [accessed February 26, 2024].
- [28] D. C. Montgomery, *Design and analysis of experiments*. John Wiley & sons, 2017.
- [29] G. Chrome, "web-networks-Network-Quality-Estimation-in-Chrome," [urlhttps://www.w3.org/2020/02/05-web-networks-Network-Quality-Estimation-in-Chrome.pdf](https://www.w3.org/2020/02/05-web-networks-Network-Quality-Estimation-in-Chrome.pdf), 2020, [Online; accessed February 26, 2024].
- [30] Chrome, "Network Information API Sample," [urlhttps://googlechrome.github.io/samples/network-information/index.html](https://googlechrome.github.io/samples/network-information/index.html), 2020, [Online; accessed February 26, 2024].



research interests lie in networks and systems, particularly in next-generation transport protocols like QUIC.



**Sapna Chaudhary** received her B.Tech degree in Computer Science and Engineering from Indraprastha Engineering College, Ghaziabad, India, in 2016. Subsequently, she pursued her M.Tech degree in Computer Science and Engineering at Guru Gobind Singh Indraprastha University (GGSIPU), Delhi, India, starting in 2018. In 2020, she commenced her Ph.D. studies at Indraprastha Institute of Information Technology (IIITD), New Delhi, India. Her Ph.D. research focuses on improving the Quality of Service (QoS) of video streaming applications. Her broader research interests lie in networks and systems, particularly in next-generation transport protocols like QUIC.

**Naval Kumar Shukla** is a Software Engineer at DP World, based in Bangalore, Karnataka, India. He received his B.Tech degree in Computer Science and Engineering from Indraprastha Institute of Information Technology, Delhi, in 2023. Naval's professional journey in the technology industry began after joining DP World. His interests lie in Communication Networks and Web Development.



**Prince Sachdeva** is a skilled Lead Software Engineer graduated from the prestigious Indraprastha Institute of Information Technology, Delhi (IIIT Delhi), with a Bachelor's degree in Computer Science and Engineering (B.Tech CSE). This academic background plays a crucial role in his successful career, where he applies his insights to real-world networking, mobile computing, and software development challenges.



**Mukulika Maity** is an Assistant Professor at the Computer Science Department of Indraprastha Institute of Information Technology Delhi (IIITD), New Delhi, India. She received her M.Tech+ Ph.D. dual degree from the Computer Science Department of the Indian Institute of Technology (IIT) Bombay in 2016. She received B.E. in 2010 in the computer science department of Bengal Engineering and Science University, Shibpur. Her Ph.D. topic was health diagnosis and congestion mitigation of wireless networks. Her research interests are broadly in the areas of networks and mobile computing. She is currently working on multiple projects funded by both Govt (DST, NSC) and private organizations (Arista Networks). She is awarded an early career research award by DST, best paper awards at multiple conferences, WiFi think fest award, and teaching excellence awards at IIITD.



**Sandip Chakraborty** is an Associate Professor in the Department of Computer Science and Engineering at the Indian Institute of Technology (IIT) Kharagpur. He obtained his Bachelor's degree from Jadavpur University, Kolkata in 2009 and M.Tech and Ph.D., both from IIT Guwahati, in 2011 and 2014, respectively. The primary research interests of Dr. Chakraborty are Distributed Systems, Mobile Computing, and Human-Computer Interactions. He received various awards including the Indian National Academy of Engineering (INAE) Young Engineers' Award 2019, and the Honorable Mention Award in ACM SIGCHI EICS 2020. Dr. Chakraborty is one of the founding chairs of ACM IMOBILE, the ACM SIGMOBILE Chapter in India. He is working as an Area Editor of Elsevier Ad Hoc Networks and Elsevier Pervasive and Mobile Computing journals. He is a senior member of IEEE and ACM.