QUIC++: Protocol for video streaming over QUIC

August 20, 2023

Team Details

- **Team Members:** Vidur Goel (B.Tech CSAM) and Manas Narang (B.Tech CSAM)
- **Faculty Mentor:** Dr. Mukulika Maity

Abstract

This paper presents the development of QUIC++, an extension of the QUIC protocol optimized for media streaming applications. Carried out during the summer period, the project had the objective of enhancing the user experience in media streaming by refining the process of protocol switching within web browsers like Chromium and Firefox.These prominent browsers have integrated support for the QUIC protocol. In order to overcome potential disruptions stemming from middleboxes hindering QUIC performance, these browsers have implemented a fallback mechanism. This fallback approach facilitates the transition from the QUIC protocol to TCP in scenarios where QUIC encounters challenges due to limitations imposed by middleboxes, such as blocking or rate-limiting.
However, a notable challenge arises for browsers in distinguishing whether the encountered issues stem from network congestion or middlebox-induced rate-limiting. Such ambiguity can consequently impact the quality of media streaming. To address this concern, our research focuses on a specific solution: the seamless transfer of the QUIC congestion window to TCP, executed at the server's end. This endeavor aims to enhance the overall user experience during media streaming sessions. By addressing this challenge, the study aimed to elevate the Quality of Experience (QoE) for users engaged in media streaming over the QUIC protocol. Through innovative modifications to the protocol's behavior, this research seeks to contribute to the advancement of media streaming efficiency and user satisfaction within the QUIC framework.

Introduction

HTTP/3, a novel application protocol, has gained significant adoption among major online entities such as Google, Facebook, Akamai, and CloudFlare. These entities have reported substantial utilization of HTTP/3, with CloudFlare noting that over 30% of its traffic is attributed to this protocol. Unlike its predecessor HTTP/2, HTTP/3 leverages the QUIC transport protocol, offering advantages like a swift 0-RTT handshake, resolution of the Head-of-the-Line Blocking problem, and efficient stream multiplexing.
However, the broad integration of HTTP/3 has encountered a noteworthy challenge. The protocol relies on QUIC which uses UDP as a substrate. This enables both userspace implementation traversal of middleboxes and NAT without requiring updates. While there is no evidence of widespread, systematic disadvantage of UDP traffic compared to TCP in the Internet [1], somewhere between three [2] and five [3] percent of networks simply block UDP traffic. All applications running on top of QUIC must therefore either be prepared to accept connectivity failure on such networks, or be engineered to fall back to some other transport protocol.[4]

Consequently, browsers have implemented a fallback mechanism to ensure connectivity continuity in the presence of these middleboxes. This mechanism triggers a shift from QUIC to TCP if a stable connection cannot be maintained through QUIC. A report by Cloudflare[5] observed a decrease in the share of HTTP/3

traffic, dwindling to approximately 26% by November 2022. Correspondingly, studies[6] in Russia and Iran revealed substantial reductions in HTTP/3 usage, which declined to meager percentages due to the introduction of middleboxes that hinder QUIC traffic.
During fallback events, application streams originating from QUIC transition to TCP as the underlying transport protocol mid-stream. This transition can result in the establishment of a new TCP connection or the reuse of an existing one. However, this switch adversely affects application performance due to the incongruence between the congestion state reached under QUIC and the state TCP initiates with. Research findings[19] underscore fallback as a necessary but detrimental measure for maintaining application connectivity in the presence of middleboxes.

To mitigate this, we propose QUIC++ that minimizes the adverse effects of protocol switching on application performance. This approach transfers congestion state information during protocol transitions from QUIC to TCP, aiming to minimize disruption. Despite the unavailability of access to the YouTube server implementation, we emulated congestion state transfer by merging pertinent sections from HTTP/2 traces with HTTP/3 traces. Evaluating this strategy across 252 YouTube streaming sessions revealed an 84% performance enhancement compared to pure HTTP/3 streaming.
In summary, this study investigates the impact of fallback on application performance within the HTTP/3 context. It proposes an approach QUIC++, which transfers congestion state during protocol switching to mitigate performance disruption. We believe this transfer could be implemented using eBPF and are currently working to achieve this. This study contributes to enhancing application Quality of Experience (QoE) during fallback scenarios, despite limitations in accessing server-side codebases.

Impact of Fallback on QOE and Congestion state

It has been proven statistically that QUIC's fallback option to sustain across middleboxes is one of the root causes for poor performance of a QUIC-enabled stream.[19]
The QUIC protocol, constructed upon the UDP protocol, confronts challenges concerning the potential rate limiting or obstruction of packets attributed to security considerations imposed by internet middleboxes such as Network Address Translators (NATs), Proxies, Firewalls, and Intrusion Detection Systems. These intermediaries, predominantly TCP-oriented, often engender impediments that hinder the seamless transmission of QUIC packets. Notably, contemporary web browsers like Firefox and Chrome/Chromium mitigate this issue by deploying connection racing mechanisms. This entails the simultaneous initiation of a TCP connection alongside the QUIC connection. When the QUIC protocol experiences perturbations, connection racing activates, facilitating the employment of the winning protocol for the ongoing application. Consequently, instances of QUIC encountering difficulties due to packet rate limiting or blocking by Internet middleboxes prompt data transfer via the alternate protocol i.e,TCP.

However, when a transition from the QUIC protocol to the TCP protocol transpires, a critical concern emerges concerning the preservation of the congestion state. Owing to the segregated nature of protocol implementation, with QUIC operating in the user space and TCP in the kernel space, the establishment of a coherent connection between these domains is absent. Consequently, the transfer of the congestion state from QUIC to TCP is thwarted. Subsequent to this transition, the TCP protocol confronts two plausible scenarios: initiating a new TCP connection or resuming an existing one. Both scenarios entail substantial temporal overhead.

- **New connection:** In the case of a new connection, the requisite 3-way handshake introduces latency before data transmission can commence. Additionally, the time taken by the TCP protocol to reinstate the previous congestion state, equivalent to that during the QUIC phase, results in delayed optimization of network utilization.
- **Existing Connection:** In the case of existing connection also time taken by the TCP protocol to reinstate the previous congestion state, equivalent to that during the QUIC phase, results in delayed optimization of network.

Hypothesis

QOE in case of Quic++ is better than QOE in case of Quic.

QUIC++ (Hypothesis Testing):

Implementation and Design:
Due to the unavailability of access to the Youtube server implementation, the practical transmission of the congestion state during protocol switching is unfeasible. Thus, our approach revolves around simulating the transfer of congestion state.
Our solution approach is based on the following assumption:
Assumption: "The congestion state reached by a pure TCP trace at a time 't' will be similar to that achieved by QUIC at that time."

Our above assumption is motivated by the fact that QUIC and TCP use the same congestion control algorithm (CUBIC [9]). This commonality underpins our premise that the congestion state attained by QUIC at a given point "t" will be mirrored in TCP under parallel conditions encompassing factors like application data, network characteristics, and device attributes.

Our approach is based on a simple idea. We first streamed videos using both HTTP/3 and HTTP/2 (which uses only TCP). Then, we look for moments when the protocol changes in the HTTP/3 network data. After that, we find the corresponding parts of the HTTP/2 data using timestamps. We carefully combine these TCP parts from HTTP/2 into the right places in the HTTP/3 data. This careful process ensures that when TCP starts, it feels like it has been running smoothly right from the start. This strategy makes it seem like the congestion information from QUIC is being transferred to TCP. This way, the network stays consistent and everything works well.

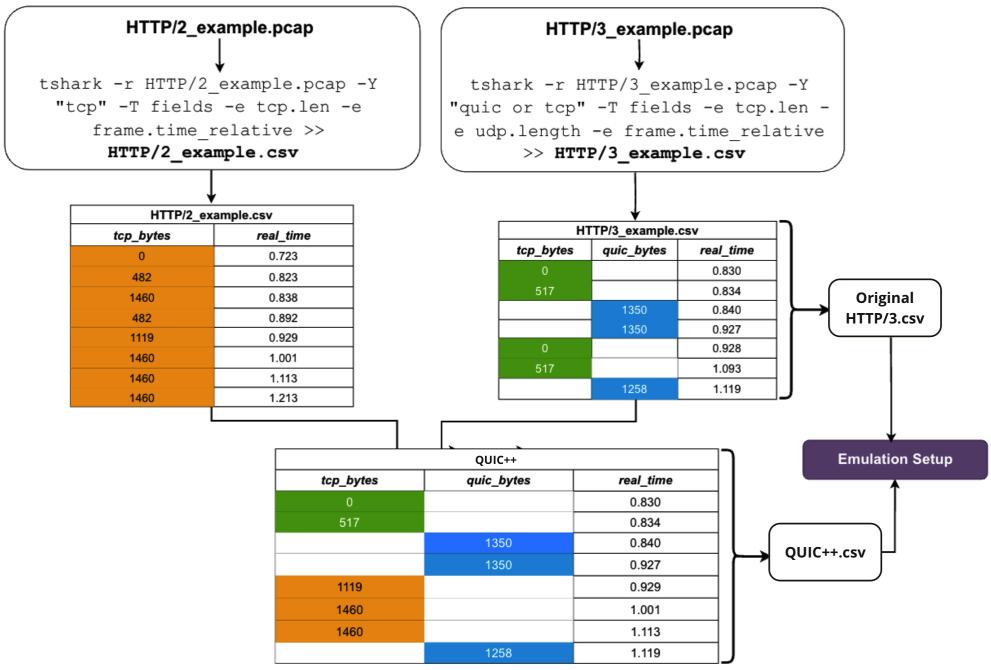

Figure 1: QUIC++ Implementation through an emulated setup

It's important to note that if we had a record where QUIC was used continuously throughout the ses-

sion, we could easily replace TCP parts in HTTP/3 streaming using information from the authentic QUIC trace. However, deciding when to switch protocols (fallback) is controlled by the browser, not us. This makes it difficult to get a completely pure QUIC trace. Because of this challenge, our approach involves using parts of the HTTP/2 trace file to imitate the state transfer instead.

As illustrated diagrammatically in Figure 1,to create this proposed "stitched" trace files, we began by collecting network traces (pcaps) from both HTTP/2 and HTTP/3 streaming activities. These traces were captured using Wireshark software and then converted into csv format. For the HTTP/3 traces, we extracted both TCP and QUIC bytes, along with their real-time timestamps. As for HTTP/2, we specifically focused on extracting TCP bytes and their real-time timestamps. This transformation process was made possible through separate bash scripts, namely "quic.sh" for HTTP/3 pcaps and "tcp.sh" for HTTP/2 pcaps. These scripts utilized tshark, which is the command-line version of Wireshark.

With the csv files ready for both HTTP/2 and HTTP/3 traces, we employed a Python script to process these files. The outcome of this script was the creation of a new "quic++.csv" stitched file. This file was generated by utilizing timestamp information to replace TCP bytes found between QUIC instances in the "http/3.csv" file with the corresponding TCP bytes from the "HTTP/2.csv" file. This replacement process occurred within specific time intervals that aligned with instances of TCP data flanked by two QUIC instances. In other words, whenever a fallback of QUIC to TCP was observed, let it be at timestamp 't1', the congestion state of the HTTP/2 trace at the timestamp 't2', which is just greater than equal to timestamp 't1' was used to replaced HTTP/3 trace and this was done until we reached a timestamp at which the connection switched back to QUIC in the original HTTP/3 trace.

This comprehensive integration of TCP bytes included interactions in both directions—between clients and servers. However, it's crucial to note that the initial TCP bytes preceding the appearance of QUIC bytes were exempt from this integration process. Through this intricate procedure, we successfully produced a synthesized trace file named "quic++.csv." This file seamlessly combines the distinct characteristics of both HTTP/3 and HTTP/2 streams, providing the groundwork for further analysis and evaluation.

Experimental Setup and Data Collection:

Network Configuration: We used packet traces that originated from a preceding investigation [8], acquired during movement within a corridor to study the YouTube application. To emulate these packet traces, we employed Mahimahi, a network emulation tool. For this purpose, a user-specific trace file is required to mimic diverse network patterns. Our approach involved converting the packet traces into a trace file compatible with Mahimahi emulation, following the methodology detailed in [10]. The user-defined Mahimahi trace file indicates the optimal times for transmitting packets of MTU size. To generate this trace file, we leveraged the throughput derived from the packet traces, calculated based on packet length and reception time. Specifically, we computed the throughput as bytes received divided by received time. Commencing at time t = 0, the first packet is dispatched, marked by a "0" in the trace file's initial line. Subsequent packet transmissions are guided by the throughput, dictating the intervals for sending MTU-sized packets.

Browser Configuration: For YouTube video streaming, we embedded the videos within the Chrome / Chromium browser. Utilizing the I-Frame source URL [11], we appended the video ID to the URL. Additionally, we set the auto-play parameter to ensure automatic playback upon loading the player. Our study encompassed a selection of 10 videos, each lasting between 40 to 50 minutes, with video quality ranging from 144p to 1080p. The streaming took place over the Chrome / Chromium browser using two distinct modes: (1) HTTP/3 with the –enable-quic flag enabled and (2) QUIC++, where the same flag was activated. The entire process was automated, with a new temporary user created for each run.

Log Collection Configuration: The acquisition of application logs facilitated the computation of Quality of Experience (QoE). Employing the formula-based metric introduced in [10], which factors in bitrate, bitrate variation, and stalls, we collected video-related information transmitted via HTTP GET or POST requests from the YouTube client. HTTP responses received from the server were also logged. The streamingstat

data, activated after a set number of frames are rendered, was captured from the client. Key parameters within streamingstat, such as itag and cmt, were extracted. The itag value was mapped to bitrate using a video-info file generated via the YouTube API. This file contained itag, corresponding bitrates, and quality labels. Bitrate variation was computed by assessing the difference between current and previous bitrates. The real-time playback time pt for a video was deduced using the cmt field. Stalls or rebuffering instances were quantified by subtracting pt from rt. Multiple instances of streamingstat were collected over a streaming session, and a moving average was calculated for QoE parameters. The instantaneous QoE at time t was determined as follows: QoE = Avg. Bitrate - Avg. Bitrate Variation - 4.3 × Avg. Stall. For each streaming session, we obtained between 200 to 300 QoE values.

Data Collection: Our data collection process encompassed streaming YouTube videos twice: once over HTTP/3 and once over QUIC++. Application logs were collected to extract QoE parameters. The resultant dataset comprised 252 video session pairs, with 126 sessions conducted via HTTP/3 and the remaining 126 sessions via QUIC++.

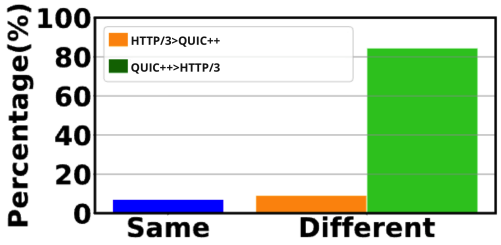

Figure 2: Testing Application QoE for HTTP/3 vs the same for QUIC++

Evaluation: We proceeded with the assessment of QUIC++. Our evaluation encompassed the broader spectrum of streaming sessions. Employing hypothesis testing akin to the approach we compared QoE values between HTTP/3 and QUIC++. As demonstrated in Figure 2, QUIC++ consistently outperformed HTTP/3, registering a favorable outcome in 84% of cases. Ultimately, this substantiates that the transfer of congestion state during fallback instances enhances the Quality of Experience for YouTube streaming.

Challenges Faced: Our dataset was extensive, which made the data processing task time-consuming. To expedite this process, we decided to run the data processing tasks on multiple systems simultaneously. This meant setting up our experimental environment on each of these machines. Our experimental setup primarily consisted of the Chromium browser and the Mahimahi emulation software. However, both of these components relied on various dependencies to function properly.

When we began this process, we found that some of the virtual machines (VMs) we used already had these software components preinstalled, as they had been used for similar tasks before. However, others did not, so we needed to install these software components manually. This installation process proved to be time-consuming and challenging because the software had newer releases, necessitating us to resolve dependencies for the updated versions.

As we continued to work through these installation challenges, it became evident that this approach was cumbersome and taking up a significant amount of our time. In response, we decided to explore alternative solutions to simplify this process. Our exploration led us to containers and Docker, a widely-used software for creating and managing containers. Upon further investigation, we realized that by encapsulating our entire experimental setup within a container, we could streamline the setup process and make it much more efficient.

Weekly Progress:

- **Week (1-3): Computer Network Basics and Relevant Research Work**
  During this period, our primary focus was on acquiring a solid understanding of fundamental computer networking concepts. Our goal was to establish a robust foundation of knowledge related to transport protocols crucial for our research, namely TCP, UDP, and QUIC. Additionally, we delved into the intricacies of application layer protocols, specifically HTTP/2 and HTTP/3, along with an exploration of the underlying transport protocols employed within these application layer protocols. Furthermore, we dedicated time to reviewing a pertinent research paper in the context of our project[19].

- **Week (4-5): Data Collection and Data Cleaning**
  During this week, we worked on collecting the network traces (pcaps) of video streams on HTTP/2 and HTTP/3 protocols. We then worked on cleaning the data and converting it into a useful form(csv files) for further processing.

- **Week (6-7): Data Processing**
  We processed the data according to the experimental setup mentioned above. We first created stitched files from original QUIC and pure TCP traces.We then converted these csv files to mahimahi trace files which were subsequently run on mahimahi emulator to generate log files and pcaps. We performed analysis on these files to test our hypothesis.

- **Week (8-9): Streamlining the Data Collection Process using Docker**
  We realised our data collection process was time consuming and needed to be implemented on multiple machines. Since there was a large amount of data to be processed, we were running our mahimahi trace files on multiple virtual machines parallelly. This meant that we needed to install mahimahi emulator and chromium browser, along with their required dependencies on each of these machines. This turned out to be a challenging and cumbersome task and we concluded that it would be best if we containerised our data collection setup. Hence, we spent this time to learn about Docker(tool used to make containers) and tried to build our experimental setup as a containerised application.

- **Week (10-12): Exploring ways to implement our solution**
  After analysing our data, we decided to proceed with the implementation of the transfer of congestion state from QUIC to TCP during fallback and started exploring ways to do this. We came across a research paper where eBPF technology had been used to make TCP stack more extensible[12] by implementing new eBPF callbacks to support user-defined TCP options, and in that paper, one of the use cases they implemented was the "Option to Request Initial Congestion Window". This led us to believe that it might be possible to use eBPF for accessing the congestion state of QUIC and subsequently transferring it to TCP. Thus, we utilised various online resources to gain a better understanding of the network stack and how eBPF works in order to understand the eBPF callbacks relevant to our use case.
  During this time, we also began working on building a client-server model so that we have an interface where we can test out our proposed solution. We came across a research paper[13] where such a network setup was built and used for stall prediction during a QUIC stream. The study used DASH and Go programming language since QUIC protocol is implemented in Go, and Youtube uses DASH. The study exploited the fact that network metrics are stored in qlog logging format in the quic-go implementation of QUIC. We studied this paper and worked on building a client-server model similar to the one given in this paper.

Conclusion and Future Work:

As we lacked direct access to the YouTube server, we were unable to execute an actual transfer of the congestion state from the user space to the kernel space. To overcome this limitation, we adopted an emulation approach by crafting a trace file that simulates the transfer of congestion state from QUIC to TCP. We envision that achieving such a transfer between user space and kernel space could potentially be realized through the Extended Berkeley Packet Filters (eBPF) framework, as explored in various scholarly

works [12, 14, 15]. eBPF enables dynamic programmability of the kernel space from the user space in real-time. Notable research studies such as [15, 16, 17, 18] have demonstrated that eBPF can be employed to explicitly communicate supplementary data, including network state information, to the TCP layer through an actively executing application. This implies that the browser employing QUIC could gather congestion state details and communicate them to the TCP layer within the kernel network stack via the dynamic execution of an eBPF program. However, a comprehensive implementation of this concept and rigorous testing of its performance, along with the consideration of security implications, require further in-depth analysis.

References:

[1] Edeline, K., Kuehlewind, M., Trammell, B., Aben, E., and B. Donnet, "Using UDP for Internet Transport Evolution (arXiv preprint 1612.07816)", 22 December 2016,
https://arxiv.org/abs/1612.07816

[2] Trammell, B. and M. Kuehlewind, "Internet Path Transparency Measurements using RIPE Atlas (RIPE72 MAT presentation)", 25 May 2016,
https://ripe72.ripe.net/wp-content/uploads/presentations/86-atlas-udpdiff.pdf

[3] Swett, I., "QUIC Deployment Experience at Google (IETF96 QUIC BoF presentation)", 20 July 2016, https://www.ietf.org/proceedings/96/slides/slides-96-quic-3.pdf

[4] QUIC RFC: The necessity of Fallback
https://www.ietf.org/archive/id/draft-ietf-quic-applicability-09.html#name-the-necessity-of-fallback-2

[5] D. B. Lucas Pardue. HTTP RFCs have evolved: A Cloudflare view of HTTP usage trends.
https://blog.cloudflare.com/cloudflare-view-http3-usage/, 2022

[6] M. Stucchi. Mongolia Joins Growing Number of Countries Reducing Openness and Resilience of the Internet. https://rb.gy/zyolpk, 2020

[7] 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.

[8] C. Gutterman, K. Guo, S. Arora, X. Wang, L. Wu, E. Katz-Bassett, and G. Zussman. Requet Dataset. https://github.com/Wimnet/RequetDataSet, 2019

[9] S. Ha, I. Rhee, and L. Xu. Cubic: a new tcp-friendly high-speed tcp variant. ACM SIGOPS operating systems review, 42(5):64–74, 2008

[10] H. Mao, R. Netravali, and M. Alizadeh. Neural adaptive video streaming with pensieve. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication, pages 197–210. ACM, 2017

[11] YouTube. YouTube Embedded Players and Player Parameters.
https://developers.google.com/youtube/player_parameters, 2021

[12] Making the Linux TCP stack more extensible with eBPF
https://inl.info.ucl.ac.be/system/files/tcp-ebpf.pdf

[13] Cross that boundary: Investigating the feasibility of cross-layer information sharing for enhancing ABR decision logic over QUIC
https://dl.acm.org/doi/pdf/10.1145/3592473.3592563

[14] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira. Fast packet processing with eBPF and XDP: Concepts, code, challenges, and applications. ACM Computing Surveys (CSUR), 53(1):1–36, 2020

[15] M. Jadin, Q. De Coninck, L. Navarre, M. Schapira, and O. Bonaventure. Leveraging eBPF to make TCP path-aware. IEEE Transactions on Network and Service Management, 19(3):2827–2838, 2022

[16] V.-H. Tran and O. Bonaventure. Making the linux TCP stack more extensible with eBPF. In Proc. of the Netdev 0x13, Technical Conference on Linux Networking, 2019

[17] V.-H. Tran and O. Bonaventure. Beyond socket options: making the Linux TCP stack truly extensible. In 2019 IFIP Networking Conference (IFIP Networking), pages 1–9. IEEE, 2019

[18] Y. Zhou, Z. Wang, S. Dharanipragada, and M. Yu. Electrode: Accelerating distributed protocols with eBPF. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), pages 1391–1407, 2023

[19] YouTube over Google's QUIC vs Internet Middleboxes: A Tug of War between Protocol Sustainability and Application QoE https://arxiv.org/pdf/2203.11977.pdf