LiteQUIC: Improving QoE of Video Streams by Reducing CPU Overhead of QUIC

Pengqiang Bi
Shandong University
Qingdao, China
pq.bi@mail.sdu.edu.cn

Yifei Zou*
Shandong University
Qingdao, China
yfzou@sdu.edu.cn

Mengbai Xiao*
Shandong University
Qingdao, China
xiaomb@sdu.edu.cn

Dongxiao Yu
Shandong University
Qingdao, China
dxyu@sdu.edu.cn

Yijun Li
Baishan Cloud
Guiyang, China
yijun.li@baishan.com

Zhixiong Liu
Baishan Cloud
Guiyang, China
zhixiong.liu@baishan.com

Qun Xie
Baishan Cloud
Guiyang, China
qun.xie@baishan.com

Abstract

QUIC is the underlying protocol of the next generation HTTP/3, serving as the major vehicle delivering video data nowadays. As a userspace protocol based on UDP, QUIC features low transmission latency and has been widely deployed by content providers. However, the high computational overhead of QUIC shifts system knobs to CPUs in high-bandwidth scenarios. When CPU resources become the constraint, HTTP/3 exhibits even lower throughput than HTTP/1.1. In this paper, we carefully analyze the performance bottleneck of QUIC and find it results from ACK processing, packet sending, and data encryption. By reducing the ACK frequency, activating UDP generic segmentation offload (GSO), and incorporating PicoTLS, a high-performance encryption library, the CPU overhead of QUIC could be effectively reduced in stable network environments. However, simply reducing the ACK frequency also impairs the transmission throughput of QUIC under poor network conditions. To solve this, we develop LiteQUIC, which involves two mechanisms towards alleviating the overhead of ACK processing in addition to GSO and PicoTLS. We evaluate LiteQUIC in the DASH-based video streaming, and the results show that LiteQUIC achieves 1.2× higher average bitrate and 93.3% lower rebuffering time than an optimized version of QUIC with GSO and PicoTLS.

CCS Concepts

• **Networks** → **Transport protocols**; **Network performance analysis**; • **Information systems** → **Multimedia streaming**.

Keywords

HTTP/3, CPU overhead, DASH, ACK, GSO

*Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MM '24, October 28-November 1, 2024, Melbourne, VIC, Australia
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0686-8/24/10
https://doi.org/10.1145/3664647.3681670

ACM Reference Format:
Pengqiang Bi, Yifei Zou, Mengbai Xiao, Dongxiao Yu, Yijun Li, Zhixiong Liu, and Qun Xie. 2024. LiteQUIC: Improving QoE of Video Streams by Reducing CPU Overhead of QUIC. In *Proceedings of the 32nd ACM International Conference on Multimedia (MM '24), October 28-November 1, 2024, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3664647.3681670

1 Introduction

Video traffic has accounted for 65.9% of the Internet traffic as of 2022 [57], where HTTP-based adaptive bitrate (ABR) streaming is the de facto standard [60]. As the underlying protocol of HTTP/3 [8], QUIC [39] has been widely deployed by content providers. In Facebook, QUIC delivers 75% of its traffic and higher quality of user experience (QoE) is received [57]. YouTube stands out as the largest user of QUIC, experiencing a reduction of over 9% in rebuffering time and an increase of over 3% in throughput [19].

Despite the superior performance over TCP observed under poor network conditions [55, 56, 58, 64, 67], QUIC is noticed to have inferior transmission throughput in high-speed networks due to its high computational cost [34, 38, 68]. We try exposing this problem with a preliminary experiment, where we run DASH-based streaming sessions using HTTP/1.1 based on TCP and HTTP/3 based on QUIC. We measure the average bitrate and rebuffering time of streaming sessions, and present the results in Figure 1. With increasing clients connected to the server, HTTP/3 sessions experience degrading throughput and increasing rebuffering time while HTTP/1.1 ones do not. When there are 100 clients simultaneously fetching video data from the server, HTTP/3 has only 40% average bitrate and 21.5× more rebuffering time compared with HTTP/1.1. It is worth noting that in the experiment, the available bandwidth is always greater than the demands of all clients so that the QoE degradation is attributed to the excessive CPU overhead of QUIC.

The computational cost of QUIC can be mainly categorized into three parts: **(1) ACK processing:** To ensure reliable transmission, QUIC clients send acknowledgments (ACKs) upon receiving packets. However, as a userspace transport protocol, QUIC needs to frequently read ACKs through system calls, which introduces non-negligible overhead. Moreover, as QUIC packets are small (~1KB), processing packet-wise ACKs also demands a large proportion of the computational cost. ACK processing plays a major role in incurring excessive CPU overhead compared with TCP. **(2) packet**

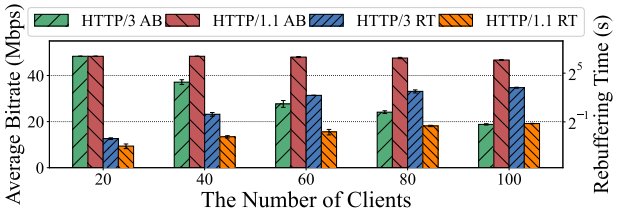

Figure 1: The average bitrate (AB) and rebuffering time (RT) of DASH-based streaming sessions using HTTP/3 and HTTP/1.1. The experimental setup is described in Section 5.2.

sending: Unlike TCP that does not limit the data length written through a system call and assemble packets in the kernel, QUIC has to organize the small QUIC packets in the userspace and sends one each system call. The frequent system calls result in significant overhead as well. **(3) data encryption:** In HTTP/1.1, data encryption occurs at the TLS layer above TCP, allowing to encrypt a large bulk of payload (up to 64KB) at a time. On the other hand, QUIC integrates the encryption module itself, and it encrypts packets individually. Switching from bulk-based encryption to packet-based means more function calls thus higher CPU overhead. Additionally, QUIC encrypts packet headers while TLS does not, which also introduces additional overhead.

In this paper, we first attempt to reduce the high CPU overhead of QUIC by reducing ACK frequency, activating UDP generic segmentation offload, and incorporating a high-performance encryption library, PicoTLS [25]. we find that simply reducing ACK frequency can impair the transmission performance in poor network conditions because the long queue at the client throttles the in-flight data. This also requires modifications to the client, which is often not feasible as most clients are browsers. Therefore, in addition to GSO and PicoTLS, we design a server-side ACK merging mechanism and an adaptive mechanism of adjusting ACK frequency at the client-side. In the CPU-limited state, LiteQUIC reads multiple ACKs that have arrived in a batch from the kernel space and merges them for processing. The ACK merge mechanism could also benefit GSO, which does not work when the ACK frequency is high. Additionally, if the client is allowed to install LiteQUIC, the ACK frequency is adaptively adjusted to further reduce the overhead and processing ACKs without degrading transmission throughput. LiteQUIC achieves up to 1.78× higher transmission throughput than picoquic [51], a QUIC implementation featuring high-performance, in the CPU-limited scenarios, and the throughput of LiteQUIC does not decrease in weak network environments as vanilla HTTP/3. When deployed in DASH-based video streaming sessions, LiteQUIC improves the average bitrate by 20.5% and reduces the rebuffering time by 93.3% compared with an optimized version of QUIC with GSO and PicoTLS. The contributions of this paper are as follows:

- By carefully analyzing QUIC, we discover that its excessive computational cost results from ACK processing, packet sending, and data encryption, and simply reducing ACK frequency impairs the transmission performance in poor networks.
- We design and implement LiteQUIC, which involves two ACK processing mechanisms towards alleviating the overhead of ACK processing without impairing transmission performance.
- We evaluate LiteQUIC in both data transmission and DASH-based streaming sessions, and the results show that LiteQUIC outperforms the state-of-the-art QUIC implementations, and it improves the QoE as a large number of clients are connected.

In Section 2, We discuss related work, and in Section 3, we analyze the CPU overhead as well as the impact caused by ACK frequency in HTTP/3. The design of LiteQUIC is presented in Section 4, along with its evaluation in Section 5. Finally, Section 6 concludes our work.

2 Related Work

2.1 High Performance Network Stack

Nowadays, the Linux network stack are struggling to keep up with rapidly growing link bandwidth, and the CPU resources become the bottleneck of data transmission. There are a number of studies devoted to solving this problem, including network stack optimization [27, 30, 41], hardware offloading techniques [3, 23], and kernel-bypass approaches [46, 66]. There are also optimization schemes for different application scenarios, such as SPRIGHT [52] proposed for improving serverless computing and dcPIM [9] for data transmission in datacenters. A recent work of Cai et al. [10] analyzes the overhead of kernel stacks and reports that TCP can achieve ~42Gbps throughput by leveraging techniques such as segmentation and receive offload, jumbo frames, and packet steering with commodity NICs. In their follow-up work [11], they propose NetChannel, which leverages the modern multicore architecture to saturate link bandwidth more than 100Gbps over a single socket.

2.2 QUIC Protocol

Recently, Yang et al. [65], compare four QUIC implementations for dissecting their CPU overhead, and propose offloading operations such as partial encryption and packet reordering to NICs to reduce the overhead. Other researches [29, 31, 50] also explore offloading TLS encryption to hardware. The performance issue of QUIC caused by ACKs has been studied [16, 17, 37, 43, 44], where Marx et al. [44] demonstrate that most implementations currently send an ACK for every 2-10 received packets instead of the recommended 2, and other studies [16, 17, 37] show that the ACK threshold of 2 imposes significant CPU overhead on the sender, while setting it to 10 can alleviate this issue. Liu et al. [43] design a strategy of dynamically adjusting ACK frequency based on Bandwidth-Delay-Product (BDP) to reduce the overhead. In addition to the aforementioned works, we not only show the impact of different ACK frequencies in various network environments, but also analyze the reasons for these differences. Moreover, the server-side ACK merging approach we designed does not require client-side support.

2.3 Video Streaning over QUIC

Since QUIC is proposed, a large body of studies explore if it could benefit video streaming. A study [6] shows that QUIC-based DASH video streaming does not necessarily result in QoE enhancement, while another thread of studies [4, 14, 15] demonstrate that QUIC outperforms TCP in lossy networks. This is because QUIC enables packet multiplexing that removes head of line blocking and it has up

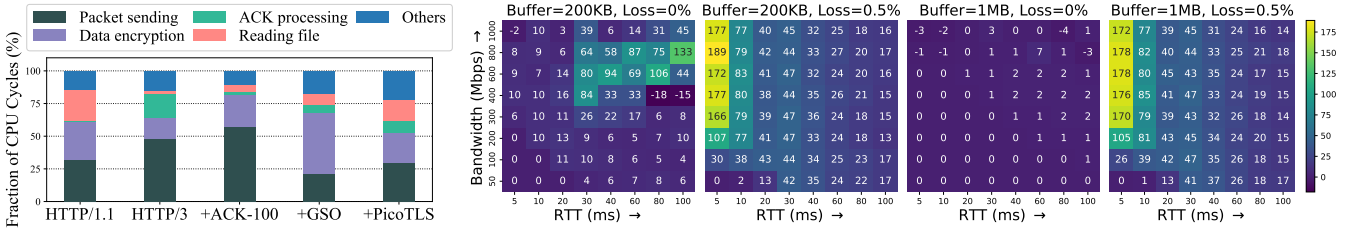

Figure 2: Left: The CPU overhead of different components in HTTP/1.1, vanilla HTTP/3, and HTTP/3 with optimization techniques. +ACK-100 sets the ACK threshold to 100, +GSO activates GSO on top of +ACK-100, and +PicoTLS incorporates the high-performance PicoTLS. **Right:** The throughput gain of setting ACK threshold to 2 over setting to 100.

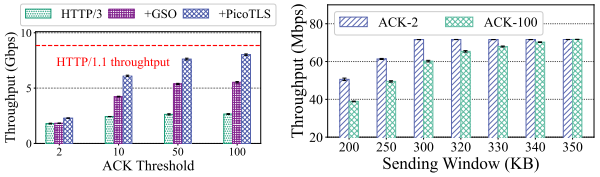

Figure 3: Left: The throughput of HTTP/3 with different ACK thresholds. The red line indicates the throughput of HTTP/1.1. **Right:** The impact of varying size of the sending window on transmission throughput when the network BDP is fixed to 300KB.

to 256 NACKs resulting in fast loss recovery, both alleviating the impact of poor network conditions on video playback [5]. Shreedhar et al. [59] observe that QUIC is faster in establishing connections with video servers compared to TCP+TLS and reduces stall durations, especially in lossy networks. And another study [63] evaluates when multiple connections exist, QUIC reaches higher QoE scores than TCP as the number of connections increases. In our work, multiple connected clients are considered one of reasons imposing excessive computational overhead to the server, which then throttles the network throughput and impairs QoE. We aim to reduce the CPU overhead of QUIC to mitigate this.

3 HTTP/3 Analysis and Optimization

In this section, we test transmission sessions using HTTP/3 for analyzing its CPU overhead and identifying performance bottlenecks.
Testbed: We set up the testbed using two machines directly connected via a 10Gbps link. Both machines are equipped with 10Gbps LREC6880BT NICs and run Ubuntu 20.04 LTS with Linux kernel 5.15.0. We evaluate the performance in the client-server mode. The server uses the nginx-quic [48], which supports both HTTP/1.1 and HTTP/3, and it has an Intel Xeon CPU at 2.90GHz. The client employs nghttp2 [47], a stress testing tool that can evaluate both HTTP/1.1 and HTTP/3, running in a docker environment on the other machine with an Intel Xeon CPU at 2.10GHz.
Experimental metrics: We launch ten clients concurrently to request a 1GB file and analyze the CPU overheads using perf [21]. We measure the network throughput and CPU overhead of HTTP/1.1 and HTTP/3 sessions. The CPU overhead of protocols are summarized as follows: **1) Packet sending:** This is the overhead of sending TCP packets in HTTP/1.1 and sending UDP packets in HTTP/3. For TCP, we measure `sock_write()`, which eventually calls `tcp_sendmsg()`. For QUIC, we measure `ngx_quic_send()`, which eventually calls `udp_sendmsg()`.**2) Data encryption:** This is mainly incurred by encryption modules in the protocol stack. For HTTP/1.1, we measure `tls_seal_record()`. For HTTP/3, we measure `ngx_quic_encrypt()`, which contains `ngx_quic_tls_seal()` and `ngx_quic_tls_hp()` to encrypt the payloads and headers of the QUIC packets, respectively.**3) ACK processing:** This involves the overhead of reading ACK frames from the socket, as well as parsing and processing them. For TCP, we measure `tcp_ack()`. For QUIC, we measure `ngx_quic_recvmsg()` that has `recvmsg()` and `ngx_quic_input_handler()`.**4) Reading file:** This is the overhead of reading files from the disk. We measure `ngx_read_file()` for both HTTP/1.1 and HTTP/3.**5) Others:** Any other overhead.

After the transmission experiments, we find that the average throughput of HTTP/3 is 1.79Gbps, which is only 20% of HTTP/1.1. The dissection of CPU overhead of two protocols is presented as the leftmost two pillars in Figure 2. For HTTP/3, its CPU overhead is mainly composed of packet sending, ACK processing and data encryption. The three modules account for 82.83% of total CPU cycles, where packet sending takes 48.24%, ACK processing takes 18.43%, and data encryption takes 16.16%. Compared to HTTP/1.1, ACK processing is the most increased component.

Towards the three major bottlenecks, we first attempt to reduce the CPU overhead by reducing the ACK frequency, activating GSO and incorporating the efficient encryption library PicoTLS.

3.1 Reducing ACK Frequency

HTTP/3 incurs higher CPU overhead of ACK processing because ACKs now are processed in the userspace instead of the kernel space as HTTP/1.1. To process ACKs, QUIC has to frequently read them from the kernel to the application layer. *ACK threshold* is defined in RFC 9000 [33] to alleviate such overhead. The client could determine the frequency of acknowledging QUIC packets with ACK threshold. For example, the default value of 2 in RFC means we send an ACK upon receiving two QUIC packets. We measure how ACK frequency impacts the transmission throughput, and the results are shown in the left of Figure 3. Note that the larger the ACK threshold value, the lower the frequency of ACKs sent by the client. When it is set to 100, we find that the throughput of HTTP/3 is 2.65Gbps, which is 1.48× higher than that setting ACK threshold to 2. We also plot CPU overhead of this setting in Figure 2, where we can find that the cycles required by ACK processing is

reduced to 2.65%, and it means that the throughput improvement results from the reduced CPU overhead.

Reducing the frequency of ACKs decreases the CPU overhead at the server, thus improving throughput when CPU resources are constrained. **However, when CPU resources are abundant and bandwidth resources are limited, what are the impacts of reducing ACK frequency?** To answer this, we use *tc* [2] to set latency from 5ms to 100ms and bandwidth from 50Mbps to 1000Mbps based on values commonly discovered in modern networks [12, 28]. Additionally, we set a 200KB shallow buffer and a 1 MB deep buffer to simulate the bottleneck buffer [7]. To test performance in poor network conditions, we also introduced an additional 0.5% random packet loss ratio. In each scenario, we request a file with a size of 1GB and define $TPGain$ as the throughput gain by comparing the throughput of setting ACK threshold as 2 (ACK-2) to that of setting ACK threshold to 100 (ACK-100):

$$TPGain = \frac{TP_{ACK-2} - TP_{ACK-100}}{TP_{ACK-100}}.$$

We draw the heatmaps of $TPGain$ in different network conditions in the right of Figure 2. We notice that with a shallow bottleneck buffer of 200KB, the throughput of ACK-2 surpasses that of ACK-100 in high-latency and high-bandwidth network conditions. The largest gap occurs at (800Mbps, 100ms), where the throughput of ACK-2 is 1.33× higher than ACK-100. However, in deep bottleneck buffer scenarios, there is little difference between ACK-2 and ACK-100. In networks with random packet loss, the overall throughput of ACK-2 is superior to ACK-100. In the network of (1000Mbps, 5ms), ACK-2 achieves 1.7× higher throughput than ACK-100.

We now analyze these phenomenon. First, when the in-flight data matches BDP of the network, the bandwidth is fully utilized [13]. Additionally, when it surpasses the BDP, the surplus data is queued in the buffer of the bottleneck link. Conversely, when the in-flight data is lower than the BDP, the link bandwidth is underutilized. By setting the ACK threshold to 100, the client delays sending an ACK until 100 packets are received. As a result, the actual in-flight data within the network is

$$inflight = swnd - data_c,$$

where $swnd$ represents the size of sending window at server and $data_c$ denotes the amount of data awaiting at the client. Since $swnd$ is usually converges to a fixed value in a session, $inflight$ is then determined by $data_c$. As more data are queued at the client, the less in-flight data in the network. This can be verified by varying the sending window size. We configure the network with a BDP of 300KB (80Mbps, 30ms), and changes $swnd$ from 200KB to 350KB. We measure the throughput of ACK-2 and ACK-100 and present the results in the right of Figure 3. When the sending window is less than the BDP of 300KB, both configurations fail to fully utilize the bandwidth, and ACK-100 with lower $inflight$ has lower throughput. ACK-2 can fully utilize the bandwidth when the sending window reaches the BDP of 300KB, while ACK-100 requires additional 50KB to cover the data queued at the client. We now interpret the results of the Figure 2. In scenarios with shallow bottleneck buffers, where congestion-induced packet loss leads to a sending window smaller than the BDP, ACK-2 outperforms than ACK-100 because the difference of in-flight data volume. The same explanation applies to scenarios involving random packet loss. In cases with deep buffering, the sending window is approximately the BDP plus the buffer size. Even after accounting for data queued at the client side, the $inflight$ still exceeds the BDP. Consequently, ACK-2 and ACK-100 exhibit the similar performance.

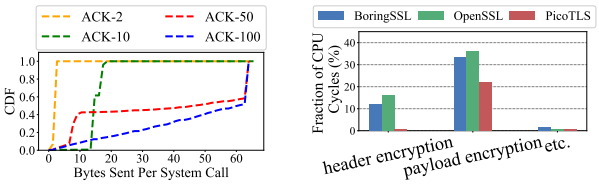

Figure 4: Left: The CDF of how many packets are sent per system call with GSO turned on. **Right:** The CPU overhead of components of BoringSSL, OpenSSL and PicoTLS.

In summary, **while reducing the frequency of ACKs can decrease the overhead of ACK processing, it also leads to inferior transmission throughput in poor network conditions** Additionally, it requires modifications to the client, making it unsuitable for clients those are not open to setup like browsers.

3.2 UDP Segmentation Offload

In Figure 2, setting ACK threshold to 100 shifts the bottleneck of HTTP/3 to packet sending, with the overhead of 57.64% compared to 32.03% of HTTP/1.1. Based on TCP, HTTP/1.1 is allowed to send millions of bytes of data to the socket in a single system call, whereas the QUIC protocol used by HTTP/3 is realized to send one QUIC packet (~1KB) per system call. Such frequent system calls consume more CPU cycles.

We attempt to mitigate the overhead of packet sending by leveraging UDP GSO [20], which can combine up to 64 packets in a single system call. **However, we notice that GSO is not effective when the ACK threshold is low.** During a transmission session, the server moves forward its sending window according to the data acknowledged. As a result, When the ACK threshold is low, the sending window is updated in a small but frequent manner. As a result, every time QUIC sends packets to the kernel, the data allowed to be sent are smaller than the GSO capacity, thus failing to reduce the number of system calls. As the ACK threshold increases, more space is released in the sending window after receiving an ACK, and GSO can starts to show its power. In the left of Figure 3, the transmission throughput is barely improved by activating GSO when the ACK threshold is 2. By increasing ACK threshold to 10, 50, and 100, enabling GSO increases the throughput by 0.74×, 1.04×, and 1.09×, respectively. We also collect the number of bytes sent per system call at different ACK thresholds in Figure 4. When the ACK threshold is 10, over 99.4% of system calls can send more than 10KB of data. As the ACK threshold increases to 100, over 50.2% of system calls send data more than 60KB.

3.3 Efficient Encryption Library: PicoTLS

By setting ACK threshold to 100 and enabling GSO, the throughput of HTTP/3 reaches 62.7% of HTTP/1.1. The component consuming the most CPU cycles is now data encryption. Encrypting the same length of data in HTTP/3 imposes the CPU overhead of 46.91%

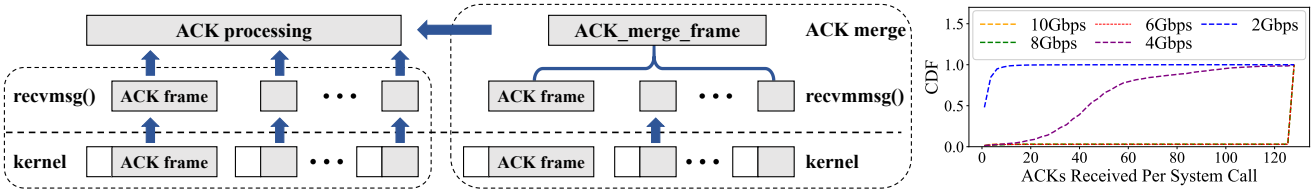

Figure 5: Left: How ACK is processed in QUIC and LiteQUIC. QUIC reads and processes an ACK frame a time, and LiteQUIC reads ACK frames in a batch and merge them into one before processing. **Right:** The number of ACKs received per `recvmmsg()` system call. 10Gbps, 8Gbps and 6Gbps correspond to CPU-limited states, while 4Gbps and 2Gbps correspond to bandwidth-limited states.

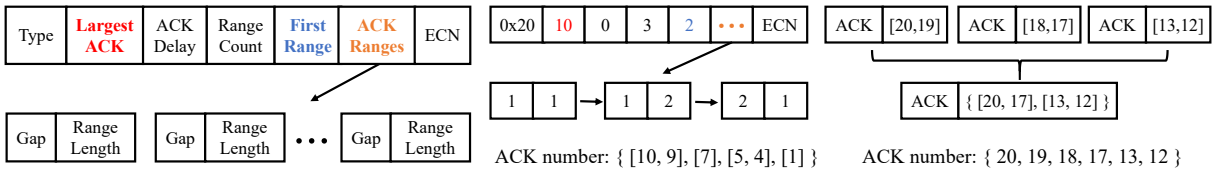

Figure 6: The format of QUIC ACK frame. The left is the format, and the middle is an example, which acknowledges packets starting from 10. The right is an example of merging three ACK frames. Before merging, each ACK frame contains an ACK range, and after merging, the only ACK frame contains two ACK ranges.

compared to 28.80% in HTTP/1.1. This discrepancy arises because HTTP/1.1 encrypts data at the SSL layer, allowing the encryption of up to 64KB of data in one pass. In contrast, QUIC encrypts each packet individually, which means more frequent function calls. It is reported that the performance of OpenSSL degrades by around 30% when encrypting small blocks [36]. Furthermore, QUIC encrypts the header of packets, incurring additional overhead.

PicoTLS is a lightweight library realizing TLS protocol. It eliminates the overhead of encrypting QUIC packet headers and reduces the overhead of encrypting small blocks of data from ~30% to ~10% [36]. We integrate PicoTLS into LiteQUIC and compare its performance with BoringSSL and OpenSSL, two widely used encryption libraries. In Figure 4, we investigate the fraction of CPU cycles of different encryption components. The results show that PicoTLS reduces the overhead of encrypting headers from 12.12% to 0.61% and payload encryption overhead from 33.25% to 21.91% compared to BoringSSL. With PicoTLS, the throughput of HTTP/3 reached 8.02Gbps, which is 90.6% of HTTP/1.1 as shown in Figure 3.

4 LiteQUIC Design

Reducing ACK frequency could reduce the CPU overhead of QUIC, but it fails to adapt to poor network conditions as discussed in Section 3.1. Furthermore, it requires modifications to the client, which is often not feasible. To address these, we design the following two mechanisms to reduce the overhead of ACK processing:

- *ACK merge*: We read ACK frames in batches and merge them into one for processing at server.
- *Adaptive ACK frequency*: If the client is configurable, we dynamically adjust the ACK frequency, only reducing it in CPU-limited scenarios.

With these two mechanisms and by additionally incorporating GSO and PicoTLS, we have LiteQUIC, a lightweight QUIC that works well in various environments.

4.1 ACK Merge

In QUIC, a UDP packet carrying an ACK frame is read from the kernel by `recvmsg()`, which is further parsed for data acknowledgment. However, in high-speed network scenarios, frequent system calls may saturate the CPUs on server. As the CPUs become the bottleneck, a backlog of UDP packets carrying ACKs appears in the server kernel. This could be alleviated by `recvmmsg()` provided by Linux kernel, which can read multiple packets in a single call. With these packets, we then merge the ACK frames parsed from them into one for further processing. The original ACK processing pipeline and the new one are presented in the left of Figure 5.

To merge the ACK frames, we need first understand their format as shown in Figure 6. The fields required in the merge are as follows: 1) **Largest ACK**: The largest packet identifier (starting from 0 in a session) to be acknowledged 2) **First ACK Range**: The number of packets to be acknowledged following the first packet (including the first packet and with decreasing identifier) 3) **ACK Range Count**: The number of acknowledgment ranges. 4) **ACK Range**: It points to the list of acknowledged ranges, Gap means the distance (calculated in packet number) from the previous acknowledgment range, Range Length means the number of packets in the current range. An ACK frame contains multiple acknowledgment ranges. In QUIC, the acknowledgment ranges are processed individually, and by processing a range, the server has to traverse the sending queue for removing packets that have been received. This means even we can retrieve multiple packets using `recvmmsg()`, it still demands excessive CPU cycles to processing ACK range by repeatedly traversing the sending queue. As a result, we merge their ACK ranges before the processing. We give an example of merging ACK ranges in the right of Figure 6, three ACK frames, each with one ACK range, are merged into one frame containing two ranges. With this method, we can quickly clear the backlogged ACK frames in a CPU-constraint scenario.

We test ACK merge by limiting bandwidth with *tc* to 10Gbps, 8Gbps, 6Gbps, 4Gbps, and 2Gbps. We collect the number of ACK frames received by each `recvmmsg()` and present the results in the right of Figure 5. When the bandwidth is limited to 2Gbps, the CPU resources are abundant, and all arriving ACK frames can be handled promptly. In this case, 48% of `recvmmsg()` read only one ACK frame, and 84% read three or fewer ACK frames. If the bandwidth increases to 4Gbps, The CPU resources do not throttle the transmission throughput only if ACK merge is used (without ACK merge, only 2.28Gbps is achieved as shown in Figure 3) In this case, 80% of `recvmmsg()` read 60 ACK frames or fewer. As the bandwidth exceeds 6Gbps, the server becomes CPU-limited, and more than 97% of `recvmmsg()` read 128 ACK frames, which is the maximum set to read each time. We can see that our ACK merge mechanism effectively reduces the CPU consumption of server and thus improves the transmission throughput.

Additionally, as mentioned in Section 3.1, the default ACK threshold of 2 fails the GSO optimization because it pushes forward the sending window in a frequent but small manner. However, with this new mechanism, processing the merged ACK frames allows the sending window to release a wide range of space, enabling GSO to combine enough packets.

4.2 Adaptive ACK Frequency

The server-side ACK merge mechanism does not require any modifications to the client and can be easily deployed. But if the client is allowed to be changed, we design an adaptive ACK frequency mechanism to further reduce the CPU overhead of server. This mechanism dynamically adjusts the ACK frequency based on network conditions, **only reducing it when the server CPU resources are constrained**. We utilize the extension specified in a working IETF draft [32] to dynamically adjust the ACK frequency in a session, which designs a `ACK-FREQUENCY` frame. At the beginning of a session, the server and client exchange the newly defined transport parameter, `min_ACK-delay` to notify each other of their support to this extension. If both parties support it, the server can send a `ACK-FREQUENCY` frame to the client at any time, informing it to modify the ACK threshold. When the server is in the CPU-limited state, most `recvmmsg()` read multiple ACK frames. Therefore, we calculate the average number of ACK frames read by `recvmmsg()` per RTT as ACK_{avg} and use it to determine that

$$state = \begin{cases} CPU\text{-}limited, & if\ ACK_{avg} \geq 2 \\ bandwidth\text{-}limited, & if\ ACK_{avg} < 2 \end{cases}$$

When ACK_{avg} exceeds 2, it indicates that the processing of ACK frames on the server is slower than the arrival of ACKs. We consider this to be a CPU-limited state, and at this point, we adjust the ACK threshold on the client to

$$T_{new} = \lfloor T_{old} \times ACK_{avg} \rfloor,$$

where T_{new} is the updated ACK threshold and T_{old} is the old value. This implies that the client should reduce the ACK frequency because the server expects to receive one ACK per system call, which consumes the least CPU resources. When ACK_{avg} is less than 2, we consider it to be in a bandwidth-limited state, and we update the ACK threshold to

$$T_{new} = \max(\lfloor \frac{T_{old}}{2} \rfloor, T_{min}),$$

where T_{min} is a lower bound set to 2. This method ensures that when the transmission session transitions from CPU-limited to bandwidth-limited state, the frequency of ACKs is updated accordingly to avoid impairing throughput.

5 Evaluation

In this section, we evaluate HTTP/3 with our LiteQUIC in both data transmission and DASH-based streaming sessions.

Implementation: We implement the ACK merge and adaptive ACK frequency mechanism in nginx-quic. We replace `recvmsg()` with `recvmmsg()`. After reading packets from the kernel, packets from different connections are separated and the ACK frames from the same connection are merged for further processing. When merging ACK frames, we only merge their ACK ranges. Occasionally, ACK frames carry processing delays of client and ECN counts, which are handled as the vanilla QUIC. For the adaptive ACK frequency mechanism, both the server-side nginx-quic and the client-side nghttp2 have undergone modifications. We add the new transmission parameter, `min_ACK-delay`, to be exchanged during connection establishment to ensure mutual support. The server sends the `ACK-FREQUENCY` frame only if both sides support this feature. The nghttp2 client is added a module to handle `ACK-FREQUENCY` frame, extracting the updated value from the server to adjust the frequency of sending ACK frames.

5.1 Data Transmission

We first evaluate the performance of LiteQUIC in terms of data transmission, with the experimental setup identical to that in Section 3. We compare it against picoquic, ngtcp2 [49], lsquic [42], quic-go [53], xquic [1], msquic [45], quicly [26] and mvfst [22]. Most of them have participated in the interoperability tests [54] of QUIC and have high stars on GitHub. We evaluate two versions of LiteQUIC in this experiment. LiteQUIC+ means we only incorporate ACK merge, and LiteQUIC++ additionally adopts adaptive ACK frequency. We also test HTTP/3 with optimizations. HTTP/3 is the vanilla version, +GSO means we activate GSO, and +PicoTLS means have both GSO and PicoTLS in HTTP/3.

We first set the ACK threshold to 2 and 10, which is currently adopted by most QUIC implementations [18], and simultaneously initiate 10 client requests for a 1GB file, recording their total throughput. The results are shown in Figure 7, and we see that when the ACK threshold is 2, the throughput of HTTP/3 without any optimization is 1.79Gbps. Even after adding GSO and PicoTLS optimizations, the throughput is only 2.28Gbps, which is only a 27.3% increase. This is because, as mentioned earlier, the server can only release space for two packets in the sending window after receiving an ACK, and GSO cannot effectively combine multiple packets for sending, thus does not work as expected. When ACK merge is used, in CPU-limited states, the server not only reduces the overhead of ACK processing but also releases more sending window, which allows GSO work effectively, leading to a throughput of 5.20Gbps in LiteQUIC+. This is 2.28× of HTTP/3 with GSO and PicoTLS, and a 20.3% improvement over picoquic and quicly. When additionally

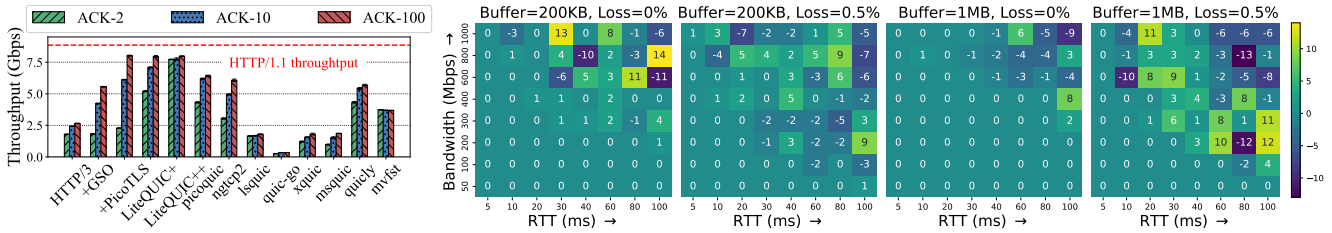

Figure 7: Left: Throughput of different QUIC implementations. **Right:** The throughput gain of LiteQUIC++ compared to itself with the ACK threshold is fixed to 2.

using the adaptive ACK frequency mechanism, the throughput of LiteQUIC++ reaches 7.72Gbps, which is 87.2% of HTTP/1.1, a 48.4% improvement over LiteQUIC+, and a 78.7% improvement over picoquic and quicly.

When the ACK threshold is set to 10, which is the by default configuration of Chrome browsers, HTTP/3 achieves a throughput of 2.43Gbps. At this point, the bottleneck lies in packet sending and data encryption. Significant improvements can be observed when utilizing GSO and PicoTLS, resulting in a throughput of 6.11Gbps, marking a 2.51× enhancement compared to without them. The performance of HTTP/3 closely approaches that of picoquic, which is the best among QUIC implementations at 6.17Gbps. LiteQUIC+ demonstrates a throughput of 7.09Gbps, surpassing HTTP/3 with GSO and PicoTLS by 16%. LiteQUIC++ achieves the highest throughput at 7.75Gbps, a 9% improvement over LiteQUIC+.

When the ACK threshold is set to 100, the overhead of ACK processing is supposed to be significantly reduced. However, with this optimization, the throughput of HTTP/3 remains as low as 2.65Gbps, showing only a 9% increase compared to when the ACK threshold is set to 10. This suggests that the bottleneck at this stage is not primarily due to ACK processing. With the integration of GSO and PicoTLS, the proportion of overhead caused by ACK processing becomes more prominent. Therefore, reducing the ACK frequency leads to substantial benefits. In this scenario, a throughput of 8.02Gbps is achieved, marking a 31% improvement over the performance with an ACK threshold of 10. LiteQUIC+ and LiteQUIC++ achieve throughputs of 7.94Gbps and 7.97Gbps, respectively, closely rivaling HTTP/3 with GSO and PicoTLS, and both outperform other QUIC implementations. However, as mentioned in Section 3.1, simply reducing the ACK frequency can also lead to performance degradation in poor network conditions. LiteQUIC++ only reduces the ACK frequency when CPU is limited, thus not impacting throughput in bandwidth-limited scenarios. Analogous to Section 3.1, we also report the throughput gains of LiteQUIC++ compared to itself but always setting ACK threshold to 2 under different network conditions. The results are shown in the right of Figure 7. The throughput of LiteQUIC++ is similar to setting the ACK threshold to 2, without the performance degradation that occurs when the ACK threshold is set to 100.

5.2 Dash Video Streaming

We also assess LiteQUIC for DASH-based streaming sessions compared to HTTP/1.1 and HTTP/2. Both HTTP/1.1 and HTTP/2 use TLS/TCP, and their performance is almost similar, so we chose HTTP/1.1 as the representative. We deploy our DASH client in the Chrome browser, and the server based on nginx-quic. Since QUIC implemented in Chrome does not support the `ACK-FREQUENCY` frame, only ACK merge is effective in LiteQUIC.

Experimental setup: We build a DASH-based video streaming system based on HTTP/3 protocol. We use a 5-min video Red Bull[1], which is encoded into {1, 5, 10, 20, 30, 40, 50}Mbps using FFmpeg, and is further split into 10s segments using GPAC MP4Box [24]. We use *dash.js*, the most widely used browser-based DASH player, as the client. We carry out the experiments with different ABR algorithms, including Dynamic [61], BOLA [62], Throughput rule, L2A [35], and LoLP [40]. Throughput rule is a naive method that chooses the highest bitrate lower than the bandwidth measured most recently.

Metrics: We use two metrics, average bitrate (AB) and rebuffering time (RT), to evaluate the QoE.

- *Average bitrate*: The average bitrate (from 1Mbps to 50Mbps) of all played segments. The higher the AB, the higher the QoE.
- *Rebuffering time*: The total time of rebuffering during playback. The higher the RT, the lower the QoE.

We gather QoE metrics across various number of clients, various ABR algorithms, and in poor network conditions. Figure 8 shows the average bitrates, while Figure 9 displays the rebuffering time.

5.2.1 Various number of clients: Using the Dynamic algorithm with 20 clients, even without any optimizations, HTTP/3 can adequately meet video transmission requirements. However, as the number of clients increases to 40, CPU resources become insufficient, resulting in a 23% decrease in AB and introducing a 0.96-second RT. As the number of clients increases to 80, both +GSO and +PicoTLS fail to meet bandwidth requirements. Nonetheless, LiteQUIC still maintains a satisfactory watching experience, achieving an AB of 47.68Mbps and an RT of only 0.17 seconds, marking a 19% improvement in AB and an 83% reduction in RT compared to +PicoTLS. Upon reaching 100 clients, LiteQUIC experiences a decline in QoE, yet it still outperforms +PicoTLS. Its AB reaches 41.24Mbps, which is 88% of HTTP/1.1, surpassing +GSO and +PicoTLS by 48% and 20%, respectively. Moreover, LiteQUIC maintains a low RT of 0.22 seconds, representing a 97%, 96%, and 93% reduction compared to HTTP/3, +GSO, and +PicoTLS, respectively. Notably, although LiteQUIC falls behind HTTP/1.1 in AB, it achieves lower RT, with average rebuffering times of 0.13 seconds for 20 to 100 clients, compared to 0.23 seconds of HTTP/1.1, a reduction of 43%.

[1]Available at https://dash.akamaized.net/akamai/customerconference2013/redbull/

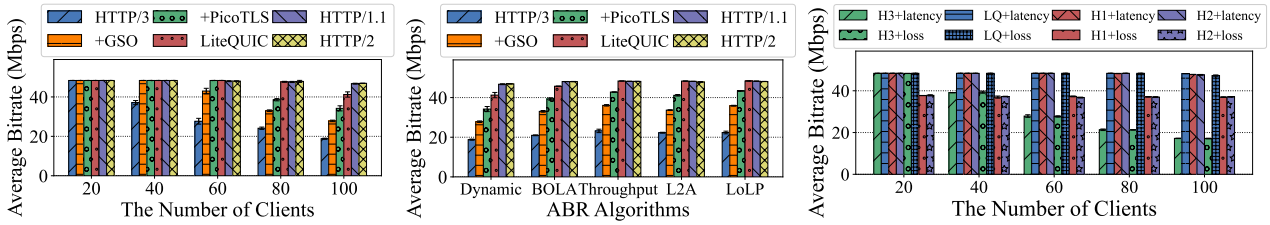

Figure 8: The average bitrate (AB) of DASH-based streaming sessions. +GSO represents the activation of GSO optimization for HTTP/3, and +PicoTLS represents the further utilization of PicoTLS on top of enabling GSO.

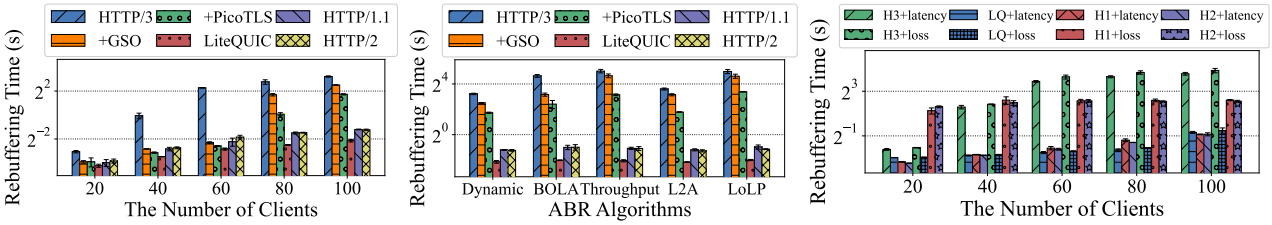

Figure 9: The rebuffering time (RT) of DASH-based streaming sessions. +GSO and +PicoTLS have the same meaning as in Figure 8.

This is primarily attributed to the latency introduced during the startup phase, whereas the 0-RTT handshake of QUIC facilitates rapid connection establishment, resulting in low startup latency.

5.2.2 Various ABR algorithms: At 100 clients, with different ABR algorithms, Dynamic achieves the lowest RT, while the highest RT is observed in HTTP/3 without any optimizations at 9.27 seconds. Throughput rule and LoLP receive the similarly high RTs, which are 32.37 seconds and 31.63 seconds, respectively, leading to inferior QoE. Meanwhile, LiteQUIC continues to maintain the lowest average RT of 0.23 seconds, a reduction of 50% compared to HTTP/1.1 and 96% compared to +PicoTLS. Regarding AB, the performance trend of HTTP/3 with different optimizations is consistent. LiteQUIC consistently outperforms +PicoTLS across all ABR algorithms, with up to a 20% improvement observed in Dynamic. Specifically, in terms of Throughput rule, L2A, and LoLP, LiteQUIC achieves AB values of over 48.30Mbps, marking increases of 17% compared to Dynamic. Overall, when throughput is insufficient, Dynamic performs better by adopting a more conservative strategy, ensuring a lower RT. On the other hand, Throughput rule and LoLP introduce excessively high RTs, severely degrading the user viewing experience. Conversely, when throughput just satisfies requirements, the conservative strategy of Dynamic leads to lower AB compared to other ABR algorithms with LiteQUIC. For example, L2A with LiteQUIC reaches both high AB and low RT.

5.2.3 Poor network conditions: We introduce a latency of 20ms and packet loss of 0.5% to compare the performance in a weak network environment. Using L2A as the ABR algorithm due to its high bandwidth utilization and low rebuffering time. When the latency is introduced, there is no significant change in performance. However, with the introduction of packet loss, a notable deterioration in the performance of HTTP/1.1 is observed. Due to the elimination of head-of-line blocking and wider acknowledgment scope compared to TCP, QUIC exhibits superior performance in high packet loss scenarios. With 20 clients, even unoptimized HTTP/3 outperforms HTTP/1.1. At this point, it achieves 1.28× the AB of HTTP/1.1 and reduces RT by 89.9%. As the number of clients exceeds 60, HTTP/3 enters a CPU-limited state, unable to match available bandwidth due to its overhead, resulting in performance dropping below HTTP/1.1. However, by reducing CPU overhead, LiteQUIC maintains a high QoE even with 100 clients and performs well in weak network environments. At this stage, it achieves 1.28× the AB of HTTP/1.1 and lowers RT by 85.2%. In summary, HTTP/1.1 has low CPU overhead but suffers in high packet loss scenarios. The vanilla QUIC achieves superior performance in poor network conditions but its performance quickly degrades as more clients are connected. With our optimizations on reducing CPU overhead, LiteQUIC achieves high QoE no matter in a poor network scenario or with a large number of clients connected.

6 Conclusion

In this work, we identify three major sources of CPU overhead in QUIC: ACK processing, packet sending, and data encryption. By reducing the ACK frequency, activating UDP GSO, and incorporating a high-performance encryption library, the CPU overhead of QUIC could be effectively reduced. However, directly reducing the ACK frequency leads to low throughput in poor network environments and GSO does now work well with high ACK frequency. Therefore, we propose LiteQUIC, a lightweight QUIC for complex network environments that has the server-side ACK merging and the client-side adaptive ACK frequency mechanism. We test LiteQUIC in both data transmission and DASH-based streaming sessions. The results show that LiteQUIC achieves high throughput and QoE, regardless of whether the server CPU is heavily crammed due to an increasing number of connected clients or the network environment is poor.

Acknowledgments

This work is supported in part by the National Natural Science Foundation of China (No.62102229), the Natural Science Foundation of Shandong Province, China (No.ZR2022ZD02), and the Qingdao Municipal Science and Technology Bureau (No.23-1-2-qljh-8-gx).

References

[1] Alibaba. 2021. *Xquic repository.* Retrieved March 21, 2024 from https://github.com/alibaba/xquic

[2] Werner Almesberger et al. 1999. Linux network traffic control—implementation overview.

[3] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 93–109.

[4] Sevket Arisu and Ali C. Begen. 2018. Quickly Starting Media Streams Using QUIC. In *Proceedings of the 23rd Packet Video Workshop* (Amsterdam, Netherlands) *(PV '18)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3210424.3210426

[5] Divyashri Bhat, Rajvardhan Deshmukh, and Michael Zink. 2018. Improving QoE of ABR Streaming Sessions through QUIC Retransmissions. In *Proceedings of the 26th ACM International Conference on Multimedia* (Seoul, Republic of Korea) *(MM '18)*. Association for Computing Machinery, New York, NY, USA, 1616–1624. https://doi.org/10.1145/3240508.3240664

[6] Divyashri Bhat, Amr Rizk, and Michael Zink. 2017. Not so QUIC: A performance study of DASH over QUIC. In *Proceedings of the 27th workshop on network and operating systems support for digital audio and video*. 13–18.

[7] Pengqiang Bi, Mengbai Xiao, Dongxiao Yu, and Guanghui Zhang. 2023. oBBR: Optimize Retransmissions of BBR Flows on the Internet. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 537–551. https://www.usenix.org/conference/atc23/presentation/bi

[8] Mike Bishop. 2022. HTTP/3. RFC 9114. https://doi.org/10.17487/RFC9114

[9] Qizhe Cai, Mina Tahmasbi Arashloo, and Rachit Agarwal. 2022. dcPIM: near-optimal proactive datacenter transport. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 53–65. https://doi.org/10.1145/3544216.3544235

[10] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 65–77. https://doi.org/10.1145/3452296.3472888

[11] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. 2022. Towards μs tail latency and terabit ethernet: disaggregating the host network stack. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 767–779. https://doi.org/10.1145/3544216.3544230

[12] Yi Cao, Arpit Jain, Kriti Sharma, Aruna Balasubramanian, and Anshul Gandhi. 2019. When to use and when not to use BBR: An empirical analysis and evaluation study. In *Proceedings of the Internet Measurement Conference* (Amsterdam, Netherlands) *(IMC '19)*. Association for Computing Machinery, New York, NY, USA, 130–136. https://doi.org/10.1145/3355369.3355579

[13] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2017. BBR: congestion-based congestion control. *Commun. ACM* 60, 2 (jan 2017), 58–66. https://doi.org/10.1145/3009824

[14] Sindhu Chellappa and Radim Bartos. 2022. Adaptability between ABR algorithms in DASH video streaming and HTTP/3 over QUIC: research proposal. In *Proceedings of the 13th ACM Multimedia Systems Conference* (Athlone, Ireland) *(MMSys '22)*. Association for Computing Machinery, New York, NY, USA, 388–392. https://doi.org/10.1145/3524273.3533932

[15] Sindhu Chellappa and Radim Bartos. 2022. Is QUIC Quicker with HTTP/3? An Empirical Analysis of Quality of Experience with DASH Video Streaming. In *2022 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)*. IEEE, 237–242.

[16] Ana Custura, Tom Jones, and Gorry Fairhurst. 2020. Impact of Acknowledgements using IETF QUIC on Satellite Performance. In *2020 10th Advanced Satellite Multimedia Systems Conference and the 16th Signal Processing for Space Communications Workshop (ASMS/SPSC)*. 1–8. https://doi.org/10.1109/ASMS/SPSC48805.2020.9268894

[17] Ana Custura, Tom Jones, Raffaello Secchi, and Gorry Fairhurst. 2023. Reducing the acknowledgement frequency in IETF QUIC. *International Journal of Satellite Communications and Networking* 41, 4 (2023), 315–330. https://doi.org/10.1002/sat.1466 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/sat.1466

[18] Ana Custura, Tom Jones, Raffaello Secchi, and Gorry Fairhurst. 2023. Reducing the acknowledgement frequency in IETF QUIC. *International Journal of Satellite Communications and Networking* 41, 4 (2023), 315–330.

[19] Ian Swett David Schinazi, Fan Yang. 2020. *Chrome is deploying HTTP/3 and IETF QUIC*. Retrieved March 29, 2024 from https://blog.chromium.org/2020/10/chrome-is-deploying-http3-and-ietf-quic.html

[20] Willem De Bruijn and Eric Dumazet. 2018. Optimizing UDP for content delivery: GSO, pacing and zerocopy. In *Linux Plumbers Conference*.

[21] Arnaldo Carvalho De Melo. 2010. The new linux perf tools. In *Slides from Linux Kongress*, Vol. 18. 1–42.

[22] Facebook. 2018. *Mvfst, repository.* Retrieved March 21, 2024 from https://github.com/facebook/mvfst

[23] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking:{SmartNICs} in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 51–66.

[24] Gpac. 2014. *Gpac repository.* Retrieved March 29, 2024 from https://github.com/gpac/gpac

[25] H2o. 2016. *Picotls repository.* Retrieved March 29, 2024 from https://github.com/h2o/picotls

[26] H2o. 2017. *Quicly, repository.* Retrieved March 21, 2024 from https://github.com/h2o/quicly

[27] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. {MegaPipe}: A New Programming Interface for Scalable Network {I/O}. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 135–148.

[28] Mario Hock, Roland Bless, and Martina Zitterbart. 2017. Experimental evaluation of BBR congestion control. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*. 1–10. https://doi.org/10.1109/ICNP.2017.8117540

[29] Xiaokang Hu, Changzheng Wei, Jian Li, Brian Will, Ping Yu, Lu Gong, and Haibing Guan. 2019. QTLS: high-performance TLS asynchronous offload framework with Intel® QuickAssist technology. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 158–172. https://doi.org/10.1145/3293883.3295705

[30] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. 2020. TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 127–140.

[31] Takashi Isobe, Satoshi Tsutsumi, Koichiro Seto, Kenji Aoshima, and Kazutoshi Kariya. 2010. 10 Gbps implementation of TLS/SSL accelerator on FPGA. In *2010 IEEE 18th International Workshop on Quality of Service (IWQoS)*. 1–6. https://doi.org/10.1109/IWQoS.2010.5542723

[32] Jana Iyengar, Ian Swett, and Mirja Kühlewind. 2024. *QUIC Acknowledgment Frequency.* Internet-Draft draft-ietf-quic-ack-frequency-08. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-quic-ack-frequency/08/ Work in Progress.

[33] Jana Iyengar and Martin Thomson. 2021. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000. https://doi.org/10.17487/RFC9000

[34] Benedikt Jaeger, Johannes Zirngibl, Marcel Kempf, Kevin Ploch, and Georg Carle. 2023. QUIC on the Highway: Evaluating Performance on High-rate Links. In *2023 IFIP Networking Conference (IFIP Networking)*. 1–9. https://doi.org/10.23919/IFIPNetworking57963.2023.10186365

[35] Theo Karagkioules, Rufael Mekuria, Dirk Griffioen, and Arjen Wagenaar. 2020. Online learning for low-latency adaptive streaming. In *Proceedings of the 11th ACM Multimedia Systems Conference*. 315–320.

[36] Kazuho. 2020. *fusion AES-GCM engine.* Retrieved March 21, 2024 from https://github.com/h2o/picotls/pull/310

[37] Jana Iyengar Kazuho Oku. 2020. *QUIC matches TCP's efficiency, says our research. | Fastly.* Retrieved March 28, 2024 from https://www.fastly.com/blog/measuring-quic-vs-tcp-computational-efficiency

[38] Michael König, Oliver P. Waldhorst, and Martina Zitterbart. 2023. QUIC(k) Enough in the Long Run? Sustained Throughput Performance of QUIC Implementations. In *2023 IEEE 48th Conference on Local Computer Networks (LCN)*. 1–4. https://doi.org/10.1109/LCN58197.2023.10223395

[39] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. 2017. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the conference of the ACM special interest group on data communication*. 183–196.

[40] May Lim, Mehmet N Akcay, Abdelhak Bentaleb, Ali C Begen, and Roger Zimmermann. 2020. When they go high, we go low: low-latency live streaming in dash. js with LoL. In *Proceedings of the 11th ACM Multimedia Systems Conference*. 321–326.

[41] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. 2016. Scalable kernel tcp design and implementation for short-lived connections. *ACM SIGARCH Computer Architecture News* 44, 2 (2016), 339–352.

[42] Litespeedtech. 2017. *Lsquic repository.* Retrieved March 21, 2024 from https://github.com/litespeedtech/lsquic

[43] Yang Liu, Zhaoxian Yang, Yuxiang Peng, Ting Bi, and Tao Jiang. 2023. Bandwidth-Delay-Product-Based ACK Optimization Strategy for QUIC in Wi-Fi Networks. *IEEE Internet of Things Journal* 10, 20 (2023), 17635–17646. https://doi.org/10.1109/JIOT.2023.3277562

[44] Robin Marx, Joris Herbots, Wim Lamotte, and Peter Quax. 2020. Same Standards, Different Decisions: A Study of QUIC and HTTP/3 Implementation Diversity. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC* (Virtual Event, USA) *(EPIQ '20)*. Association for Computing Machinery, New York, NY, USA, 14–20. https://doi.org/10.1145/3405796.3405828

[45] Microsoft. 2019. *Msquic, repository.* Retrieved March 21, 2024 from https://github.com/microsoft/msquic

[46] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting network support for RDMA. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 313–326.

[47] Nghttp2. 2013. *Nghttp2, repository.* Retrieved March 21, 2024 from https://github.com/nghttp2/nghttp2

[48] Nginx. 2021. *Nginx-quic.* Retrieved March 21, 2024 from https://quic.nginx.org/

[49] Ngtcp2. 2017. *Ngtcp2 repository.* Retrieved March 21, 2024 from https://github.com/ngtcp2/ngtcp2

[50] Boris Pismenny, Ilya Lesokhin, Liran Liss, and Haggai Eran. 2016. TLS Offload to Network Devices. *Netdev 1.2* (2016). https://api.semanticscholar.org/CorpusID:35145139

[51] Private-octopus. 2017. *Picoquic repository.* Retrieved March 21, 2024 from https://github.com/private-octopus/picoquic

[52] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. 2022. SPRIGHT: extracting the server from serverless computing! high-performance eBPF-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 780–794. https://doi.org/10.1145/3544216.3544259

[53] Quic-go. 2016. *Quic-go repository.* Retrieved March 21, 2024 from https://github.com/quic-go/quic-go

[54] Quic-interop. 2019. *Quic-interop-runner.* Retrieved March 21, 2024 from https://interop.seemann.io/

[55] Jan Rüth, Konrad Wolsing, Klaus Wehrle, and Oliver Hohlfeld. 2019. Perceiving QUIC: do users notice or even care?. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies* (Orlando, Florida) *(CoNEXT '19)*. Association for Computing Machinery, New York, NY, USA, 144–150. https://doi.org/10.1145/3359989.3365416

[56] Darius Saif, Chung-Horng Lung, and Ashraf Matrawy. 2021. An Early Benchmark of Quality of Experience Between HTTP/2 and HTTP/3 using Lighthouse. In *ICC 2021 - IEEE International Conference on Communications*. 1–6. https://doi.org/10.1109/ICC42927.2021.9500258

[57] Sandvine. 2023. *2023 Global Internet Phenomena Report.* Retrieved March 29, 2024 from https://www.sandvine.com/phenomena

[58] Tanya Shreedhar, Rohit Panda, Sergey Podanev, and Vaibhav Bajpai. 2022. Evaluating QUIC Performance Over Web, Cloud Storage, and Video Workloads. *IEEE Transactions on Network and Service Management* 19, 2 (2022), 1366–1381. https://doi.org/10.1109/TNSM.2021.3134562

[59] Tanya Shreedhar, Rohit Panda, Sergey Podanev, and Vaibhav Bajpai. 2022. Evaluating QUIC Performance Over Web, Cloud Storage, and Video Workloads. *IEEE Transactions on Network and Service Management* 19, 2 (2022), 1366–1381. https://doi.org/10.1109/TNSM.2021.3134562

[60] Kevin Spiteri, Ramesh Sitaraman, and Daniel Sparacio. 2019. From theory to practice: Improving bitrate adaptation in the DASH reference player. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 15, 2s (2019), 1–29.

[61] Kevin Spiteri, Ramesh Sitaraman, and Daniel Sparacio. 2019. From theory to practice: Improving bitrate adaptation in the DASH reference player. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 15, 2s (2019), 1–29.

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

[63] Van TONG, Hai Anh TRAN, Sami SOUIHI, and Abdelhamid MELLOUK. 2018. Empirical study for Dynamic Adaptive Video Streaming Service based on Google Transport QUIC protocol. In *2018 IEEE 43rd Conference on Local Computer Networks (LCN)*. 343–350. https://doi.org/10.1109/LCN.2018.8638062

[64] Martino Trevisan, Danilo Giordano, Idilio Drago, and Ali Safari Khatouni. 2021. Measuring HTTP/3: Adoption and Performance. In *2021 19th Mediterranean Communication and Computer Networking Conference (MedComNet)*. 1–8. https://doi.org/10.1109/MedComNet52149.2021.9501274

[65] Xiangrui Yang, Lars Eggert, Jörg Ott, Steve Uhlig, Zhigang Sun, and Gianni Antichi. 2020. Making QUIC Quicker With NIC Offload. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC* (Virtual Event, USA) *(EPIQ '20)*. Association for Computing Machinery, New York, NY, USA, 21–27. https://doi.org/10.1145/3405796.3405827

[66] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. 2016. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 43–56.

[67] Alexander Yu and Theophilus A. Benson. 2021. Dissecting Performance of Production QUIC. In *Proceedings of the Web Conference 2021* (Ljubljana, Slovenia) *(WWW '21)*. Association for Computing Machinery, New York, NY, USA, 1157–1168. https://doi.org/10.1145/3442381.3450103

[68] Xumiao Zhang, Shuowei Jin, Yi He, Ahmad Hassan, Z. Morley Mao, Feng Qian, and Zhi-Li Zhang. 2023. Poster: QUIC is not Quick Enough over Fast Internet. In *Proceedings of the 2023 ACM on Internet Measurement Conference* (, Montreal QC, Canada,) *(IMC '23)*. Association for Computing Machinery, New York, NY, USA, 730–731. https://doi.org/10.1145/3618257.3625002