

QUIC is not Quick Enough over Fast Internet

Xumiao Zhang[†], Shuowei Jin, Yi He, Ahmad Hassan^{*}, Z. Morley Mao, Feng Qian^{*}, and Zhi-Li Zhang^{**}

University of Michigan, University of Southern California^{*}, University of Minnesota^{**}

Corresponding author[†]

The Web Conference 2024 (WWW '24)

2024.07.02.

GyeongHeon Jeong(ghjeong@mmlab.snu.ac.kr)

Index

- Introduction
- Background
- QUIC Transport Performance
 - Application Study
- Root Cause Analysis
- Mitigation
- Conclusion

Introduction

- **QUIC** is a multiplexed transport-layer protocol over UDP, poised to be a foundational pillar of the next-generation Web infrastructures (e.g., HTTP/3)
- There are many researches on characterizing QUIC performance
 - Using various QUIC implementations (customized vs. commercial), compute environments (mobile vs. desktop), and network conditions (wired vs. wireless)
- Running QUIC over **high-speed networks**
 - High-speed wired links, WiFi 6/7, and 5G, (500 Mbps ~1 Gbps per connection)
- **QUIC is slow over fast internet**

About QUIC

- QUIC is a user-space transport over UDP
 - It was initially proposed and developed by Google (gQUIC)
 - IETF working group was launched in 2016 to improve the original gQUIC design (IETF QUIC)
- HTTP/3 was structured to make the HTTP syntax as well as existing HTTP/2 functionalities compatible with QUIC
- QUIC design
 - **Pro)** 0/1-RTT fast handshake, stream multiplexing for the removal of head-of-line blocking, and connection migration
 - **Con)** processing and copying data between the kernel space and user space

NIC Offloading

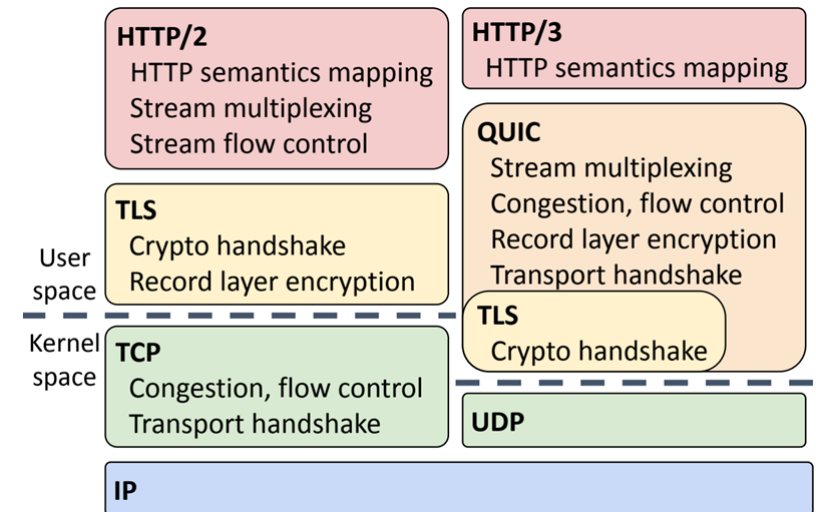
- Optimization technique that reduces the CPU overhead in network operations
 - Allowing the relevant protocol stack to transmit packets that are larger than the **MTU**
 - With offloading, a network interface controller (**NIC**) splits large data chunks into smaller segments
 - Without offloading, the segmentation is performed by the **CPU**, which creates an overhead
- Types
 - Generic segmentation offload (GSO)
 - Uses the TCP or UDP protocol to **send** large packets
 - TCP segmentation offload (TSO)
 - Uses the TCP protocol to **send** large packets
 - Generic receive offload (GRO)
 - Uses the TCP or UDP protocol to **receive** large packets

Methodology

- Comparing the **UDP+QUIC+HTTP/3 (QUIC)** stack with the **TCP+TLS+HTTP/2 (HTTP/2)** stack
 - Set up the testbed to ensure a fair comparison, that is, the observed performance gaps originate solely from the differences in the protocol themselves

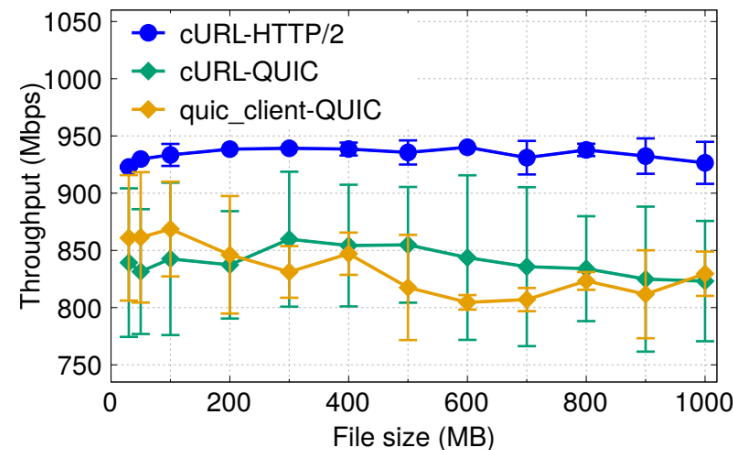
- Setting

- Server machine: Intel Xeon E5-2640 CPU
- Client desktop: Intel Core i7-6700 CPU
- 1-Gbps Ethernet / only two hops away
- Ubuntu 18.04
- Congestion control algorithm: CUBIC
- Controlling bandwidth: Linux tc
- Buffer sizes: 10x the link's bandwidth-delay product (BDP)
- Initial transport settings stay the same

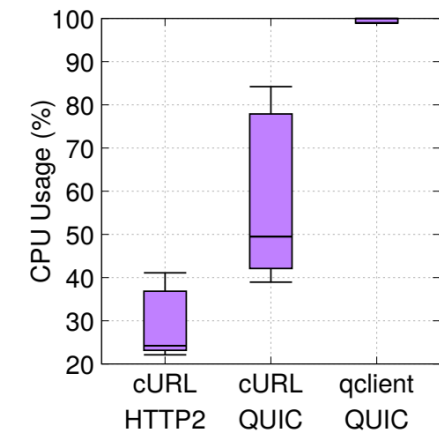


File Download on Lightweight Clients

- Two non-browser download tools: **cURL** and **quic_client**
 - cURL**: a command-line data transfer tool that supports both **QUIC** and **HTTP/2**
 - quic_client**: a standalone **QUIC** client implementation, built with the same QUIC stack as Chrome/Chromium
- Download files of different sizes, ranging from 50 MB to 1 GB
 - cURL running HTTP/2 noticeably **outperforms** both QUIC clients, well utilizing the 1 Gbps available bandwidth
 - The CPU usage when running QUIC is **higher** than HTTP/2



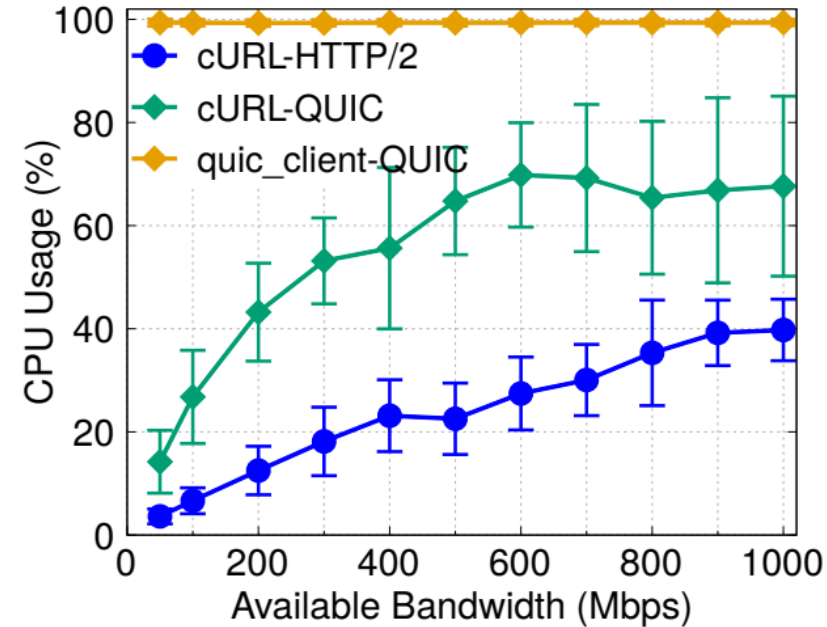
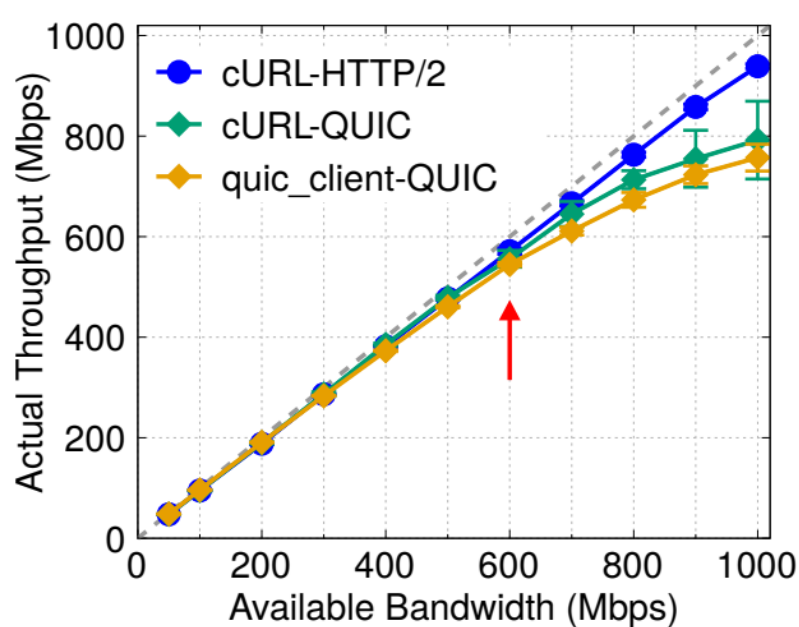
Throughput of lightweight clients during file download



CPU usage of lightweight clients 7/20


File Download on Lightweight Clients (cont.)

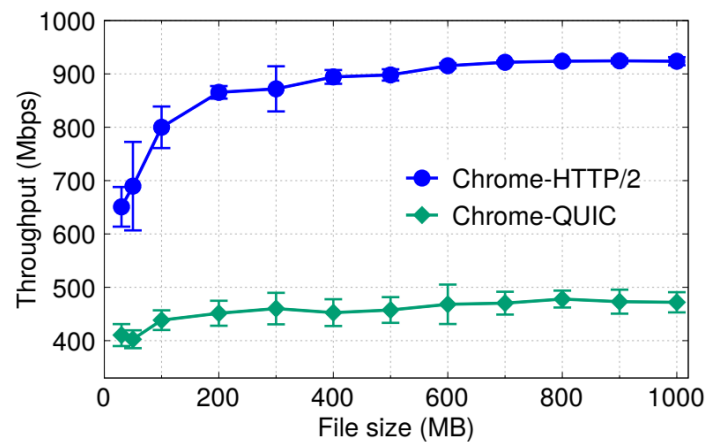
- Limit the available network bandwidth from 50 Mbps to 1000 Mbps
 - When the available bandwidth is low, QUIC and HTTP/2 exhibit similar performance
 - But, beyond around 600 Mbps, QUIC's actual throughput **starts to be bottlenecked**
 - The CPU usage for quic_client is always high and that of cURL QUIC stays around 70%



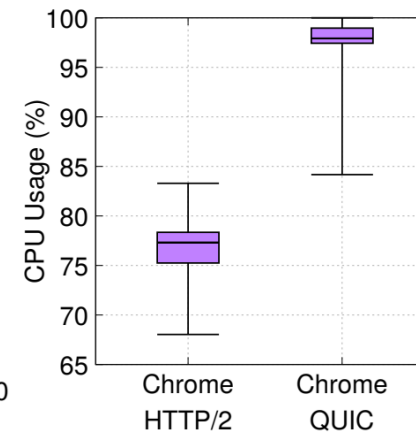
Throughput and CPU usage of cURL and quic_client during file download under limited bandwidth

File Download on Real Browsers

- Experiments on real web browsers: **Chrome browser** 
- Download files of different sizes, ranging from 50 MB to 1 GB
 - The performance gap between QUIC and HTTP/2 is **even larger** than that in our prior lightweight client experiments
 - Different from the lightweight clients, **Chrome** is a full-fledged web browser, so the CPU saturation issue is exacerbated, leading to even lower QUIC performance



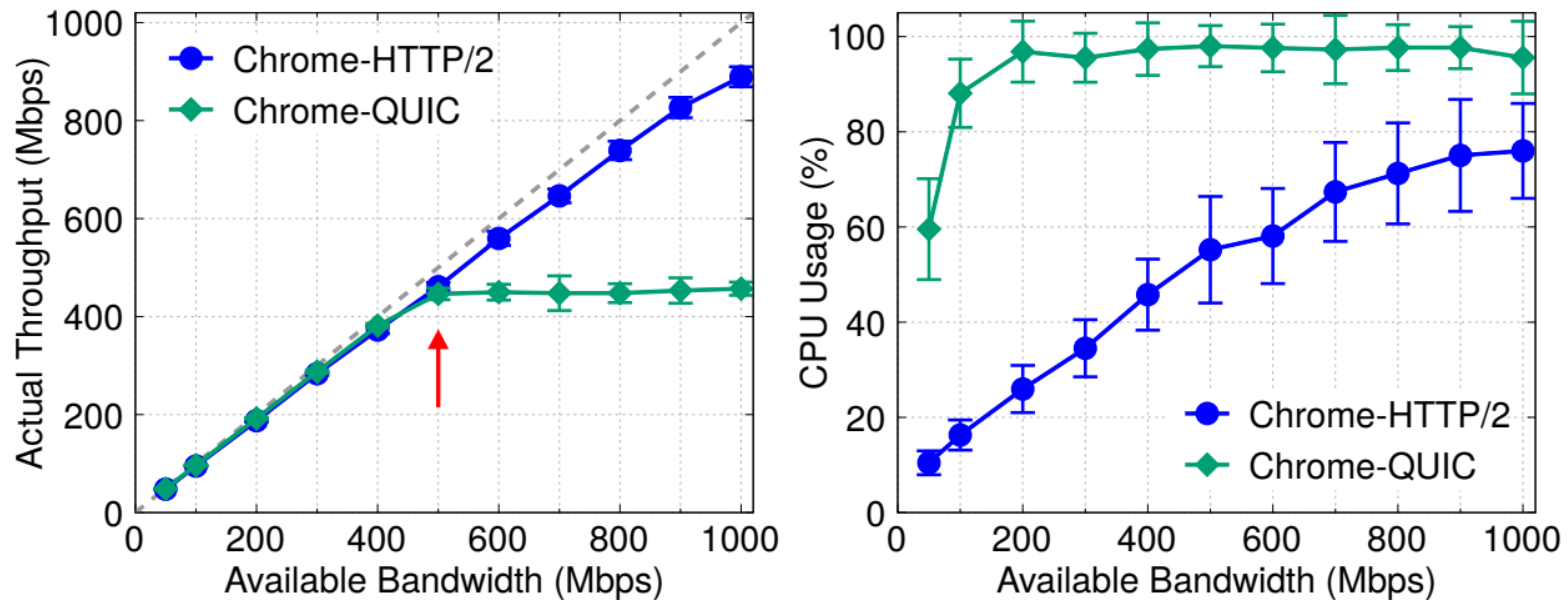
Throughput of the Chrome browser during file download



CPU usage of the Chrome browser

File Download on Real Browsers (cont.)

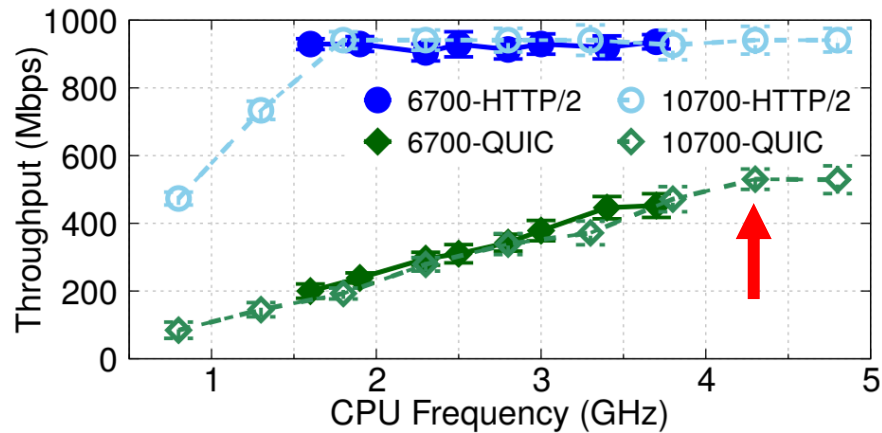
- Limit the available network bandwidth from 50 Mbps to 1000 Mbps
 - QUIC fails to fully utilize the bandwidth starting earlier at approximately **500 Mbps** (< 600Mbps)
 - Chrome with QUIC approaches 100% CPU usage when the throughput is only **200 Mbps**



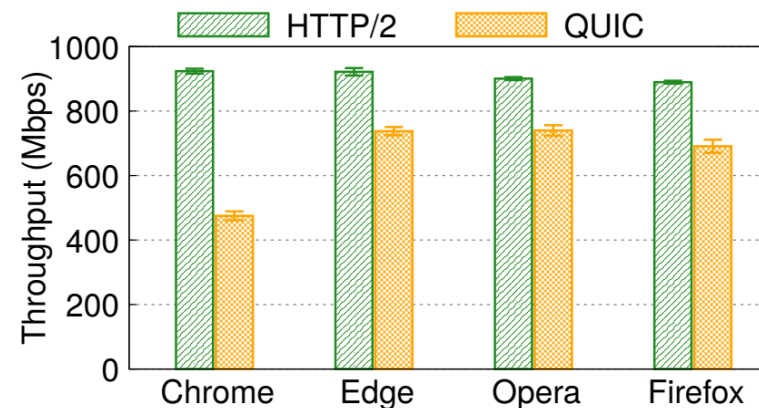
Throughput and CPU usage of the Chrome browser during file download under limited bandwidth

File Download on Real Browsers (cont.)

- Changing the CPU frequency (i.e., CPU clock speed)
 - Intel Core **i7-6700** CPU has a frequency of 3.40 GHz to 4.00 GHz, and **i7-10700** has ~4.80 GHz
 - Increasing CPU computing power can marginally narrow the performance gap between QUIC and HTTP/2
- Comparing different browsers
 - All the browsers have **worse performance** when QUIC is enabled, with **increased CPU usage**



Throughput of the Chrome browser at different CPU frequencies

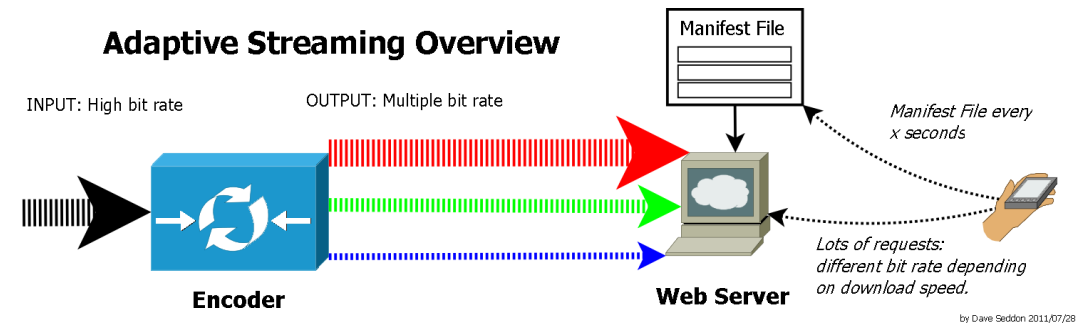



Throughput of four different browsers during file download

Table 2: Browsers' CPU usage (%).

Browser	HTTP/2	HTTP/3
Chrome	77.1±4.7	97.4±4.5
Edge	28.4±3.9	81.1±7.7
Opera	27.9±3.3	84.9±14.5
Firefox	185.8±61.9	213.0±46.5

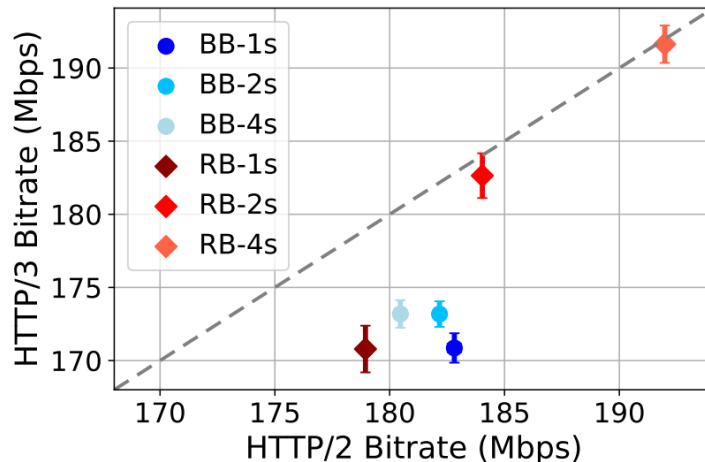
Video Streaming Setting



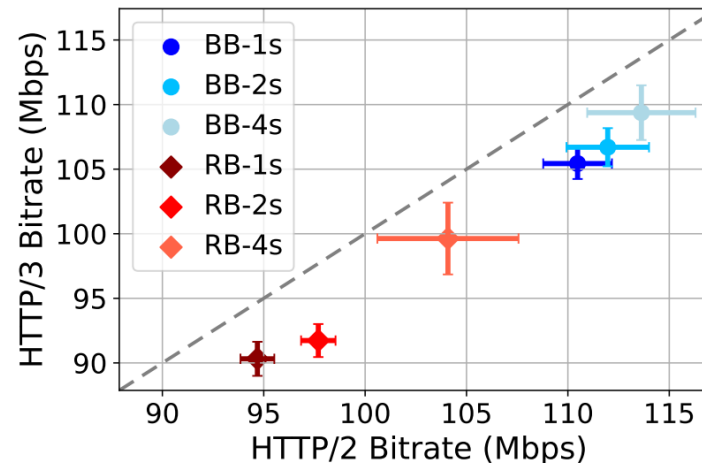
- Adaptive bitrate (**ABR**) video streaming
 - Method of video streaming over HTTP where the source content is encoded at multiple bit rates
- Using **ffmpeg** to encode a custom 4K video 
 - Generate **six tracks** at different bitrates
 - 4K video streaming usually requires 35-100 Mbps (Achieved at 5G)
 - Scale up the video bitrates with the top track bitrate reaching **200 Mbps** (Median throughput of 5G)
 - Encode the video into three different chunk durations, **1s, 2s, and 4s**
 - Bitrate adaptation algorithms:
 - **Buffer-Based (BB)**: Select bitrates with the goal of keeping the buffer occupancy high
 - **Rate-Based (RB)**: Select the highest bitrate below the bandwidth predicted from experienced throughputs during past chunk downloads
- Evaluate ABR video streaming under three types of network conditions
 - Stream over [Ethernet, 5G, 4G]

Video Streaming Experiment

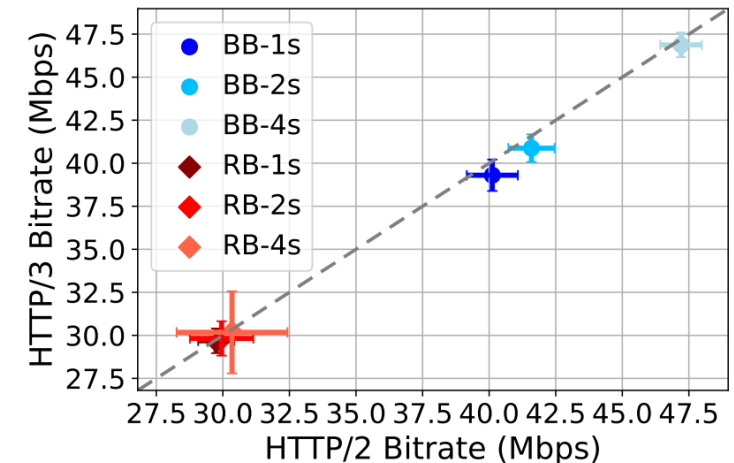
- Measuring video chunk bitrate during the streaming process
 - QUIC performs worse than HTTP/2 in Ethernet and 5G scenarios
 - The bitrate reduction goes up to **9.8%**
 - For the slow 4G networks, the performance difference is not that significant



(a) Stream over Ethernet.



(b) Stream over 5G.



(c) Stream over 4G.

Comparing average video chunk bitrate between HTTP/3 (QUIC) and HTTP/2

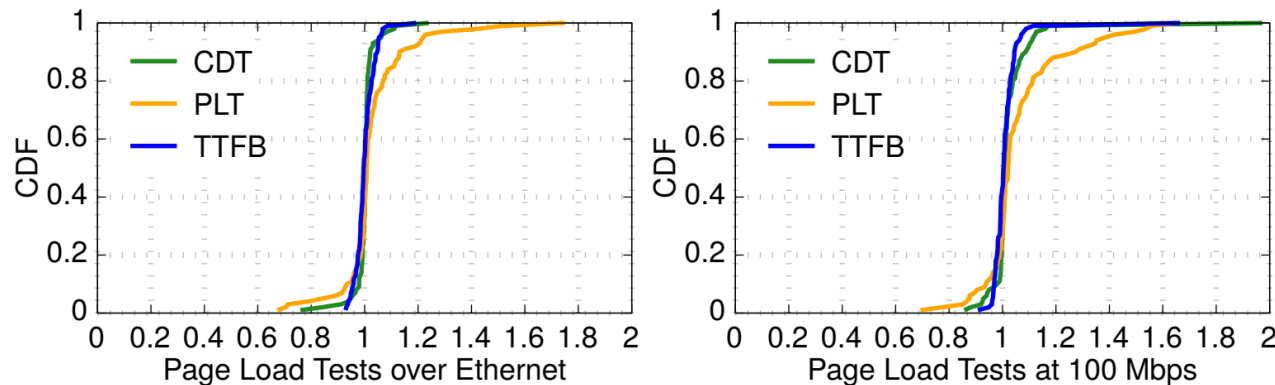
Web Page Loading Setting

- Unlike bulk file download, loading a web page usually involves transferring **multiple small objects**
 - Transferring can be either concurrent or sequential depending on object dependencies
- They conduct an experiment with Alexa's top 100 websites
- First, using the original URLs to directly load the remote websites, with QUIC enabled on Chrome
 - Note that, only **16 websites** exhibit HTTP/3 traffic during page loads
- So download these 100 websites using SiteSucker and host them locally on our web server
 - To make all the websites support HTTP/3



Web Page Loading Experiment

- Comparing the page load performance of QUIC and HTTP/2 with three metrics
 - **Content download time (CDT)**: The time to download all content needed to load the website
 - **Page load time (PLT)**: The rendering time of all components of the page is finished
 - **Time-to-first-byte (TTFB)**: The delay from sending the request to receiving the first byte of the response
- Page load tests for each website over Ethernet and 100 Mbps
 - Data point greater than 1.0 means the corresponding timer is longer in QUIC tests
 - On average, QUIC's PLT is 3.0% longer than HTTP/2's



Web page loading results (HTTP/3 over HTTP/2)

Packet Trace Analyses

- QUIC perceives much more packets than HTTP/2
 - For QUIC, the number of packets received by the OS's UDP stack is an order of magnitude higher than the number of packets received by the TCP stack during HTTP/2 downloads (744K vs 58K on average)
 - The numbers of transmitted packets are very close
 - TCP (HTTP/2) uses generic receive offload (**GRO**)
 - The link layer module in the OS combines multiple received TCP segments into a large segment of up to 64 KB
 - However, despite the availability of UDP GRO, it is not used by QUIC
- QUIC has a much higher RTT dominated by local processing
 - Though they have different ACK mechanisms, the average RTT for HTTP/2 download is **1.9ms** while QUIC's RTT skyrockets to **16.2ms**
 - Since the ping RTT between the two machines is only 0.23ms, the endpoint packet processing takes most of the packet latency
 - The performance bottleneck of QUIC appears to be on the receiver side

Root Causes via Kernel Profiling

- Excessive receiver-side processing in the kernel
 - Huge number of calls on ***netif_receive_skb*** which is invoked when a packet is received at the network interface
 - 231K vs 15K (Corresponds to the difference in the number of received UDP and TCP packets)
 - Standard way to reduce packet processing overhead is to involve **NIC offloading**
 - **Challenge** of offloading for QUIC
 - The existing UDP GSO/GRO only supports offloading a train of UDP packets with identical lengths
 - QUIC frames vary in size and are multiplexed after encryption
 - ➔ If a train of UDP datagrams has different packet sizes, existing UDP GSO/GRO cannot offload them
 - The diverse QUIC variants add complexity to realizing the QUIC offloading logic in NIC hardware

Table 3: Download 1 GB file with and without offloading.

Setup	# Sent Packets	# Recv Packets	Time (s)
QUIC (on)	743K	743K	18.60
QUIC (off)	744K	744K	18.82
HTTP/2 (on)	19K	53K	9.36
HTTP/2 (off)	744K	744K	10.84

Root Causes via User space Profiling

- Excessive receiver-side processing in the user space
 - Table 4 provides a breakdown of the time spent by each packet processing stages
 - QUIC consumes 8.7s in user space, and HTTP/2 consumes 4.1s

Table 4: A breakdown of packet processing time.

Chromium Networking Stack	QUIC (8.5s)	HTTP/2 (4.1s)
Read UDP/TCP packets from socket	0.248s	0.037s
Process UDP/TCP packets for payload	0.310s	0.084s
Decode QUIC/TLS-encrypted packets	0.660s	0.814s
Parse decrypted QUIC/HTTP2 frames	3.468s	3.182s
Generate QUIC responses (<i>e.g.</i> , ACK)	2.972s	–
Others	0.859s	0.001s

Mitigation

- Adoption of UDP GRO on the receiver side
 - Most importantly, UDP GRO needs to be deployed on the receiver side to reduce the number of packets handled by the UDP stack
 - However, given the heterogeneity of today's commodity hosts, wide deployment of UDP GRO can be challenging, not to mention supporting it in the NIC hardware
- QUIC-friendly improvements to the offloading solutions
 - UDP GSO/GRO needs to support offloading a train of packets with different sizes
- Multi-threaded download
 - Now, Chromium uses a single thread for receiving network data
 - When fetching large files, using multi-threaded download (each thread running on a separate CPU core) can improve the receive-side performance

Conclusion

- This study highlights, in environments like **fast Internet**, (>500 Mbps in experiments), QUIC's performance may not always live up to its name ("quick")
- Through comprehensive performance profiling, they reveal the root cause to be the pronounced **receiver-side processing overhead**
- The absence of certain offloading techniques like UDP GRO, and the user-space nature of QUIC might complicate its deployment
- Nevertheless, QUIC is still in its early phase, the ongoing efforts and collaborations from multiple stakeholders in the Web ecosystem is needed

Thank you for listening