Implementation and Performance Evaluation of the QUIC Protocol in Linux Kernel

Peng Wang, Carmine Bianco, Janne Riihijärvi and Marina Petrova

Presenter: Kota Yatagai

MSWIM '18: Proceedings of the 21st ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems

Abstract

- QUIC is a UDP-based transport protocol designed to improve HTTPS performance, overcoming problems in TCP
- Current QUIC deployments run in user space, but context switching per packet limits its performance
- This study implements QUIC in the Linux kernel to fairly compare it against TCP
- Experiments show that kernel-space QUIC outperforms TCP in scenarios with low latency and high packet loss

Introduction: TCP and its problems

- TCP, designed in 1974, has evolved but struggles with modern requirements
  - TCP has no built-in support for encryption
    - TLS over TCP (HTTP/2) improved security but added handshake delays
  - TCP’s strict reliable streams limits performance in real-time communications
- Updating TCP is impractical because of the “protocol ossification”
    - it would require global update to OS kernels and middleboxes

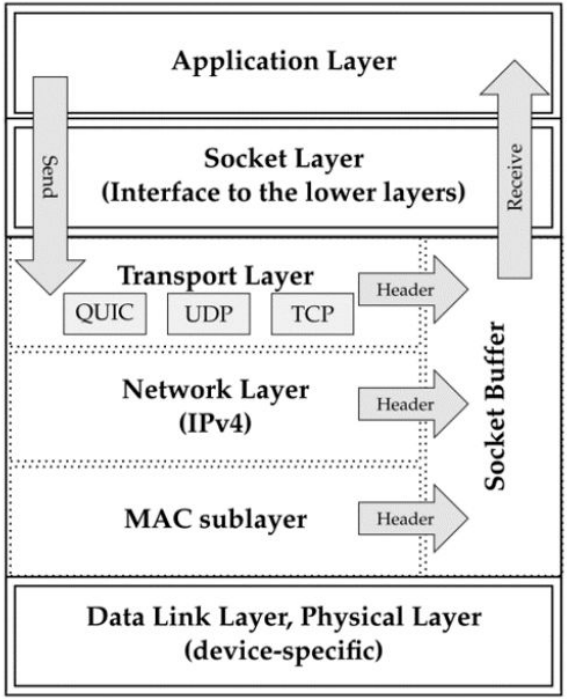
Introduction: Rise of QUIC and unfair comparison

- QUIC is designed to run in user space, making deployment quicker
- QUIC also has TLS1.3 as its required subsystem, ensuring security
  - QUIC-specific information such as initial CID is piggybacked in TLS handshake
  - Whole QUIC handshake will be done with 1RTT at maximum
- Prior studies compared user-space QUIC vs kernel-space TCP, which seemed unfair
- This work implements QUIC inside the Linux kernel for a fair comparison against TCP

Implementation: Protocol registration

QUIC initialization in kernel

- Modify kernel boot process (inet_init) so following steps are executed
  - Register QUIC's socket operations
  - Assign IPPROTO_QUIC
  - Map SOCK_DGRAM and QUIC



Implementation: Connection establishment

- a QUIC connection is established with the berkeley Socket API
  - socket(), bind(), then connect() or listen(), and then accept()

TLS handshake is not implemented in this paper

- Thus following evaluation has to be perceived as TCP vs “QUIC plain text”
  - which is, somehow fair? but definitely impractical

Evaluation: Virtual testbed setup

- Environment: Oracle VirtualBox
  - Both the client and server run on separate VMs
  - Single-core isolated CPU (100% host CPU usage allowed) with 3GB RAM
  - Intel PRO/1000 MT Gigabit Ethernet adaptor
- Host machine: Intel Core i7-6700 (8 cores, 16GB RAM)
- OS: Ubuntu 14.04 with Linux kernel 3.13.11
- Network: emulated with netem (tc)
- Method: 20 repetition of file transfer of size 2.46MB

Evaluation: Wireless testbed setup

- Environment: Up to 3 PCs with (probably) the same CPU, RAM and OS
  - One for the server and one or two for the client depending on the measurement
- Network: IEEE 802.11 (WiFi) with the ASUS RT-AC66U router
- Method: Unknown repetition of file transfer of size 39.69MB

Virtual testbed measurement: Throughput in multiple packet loss rates

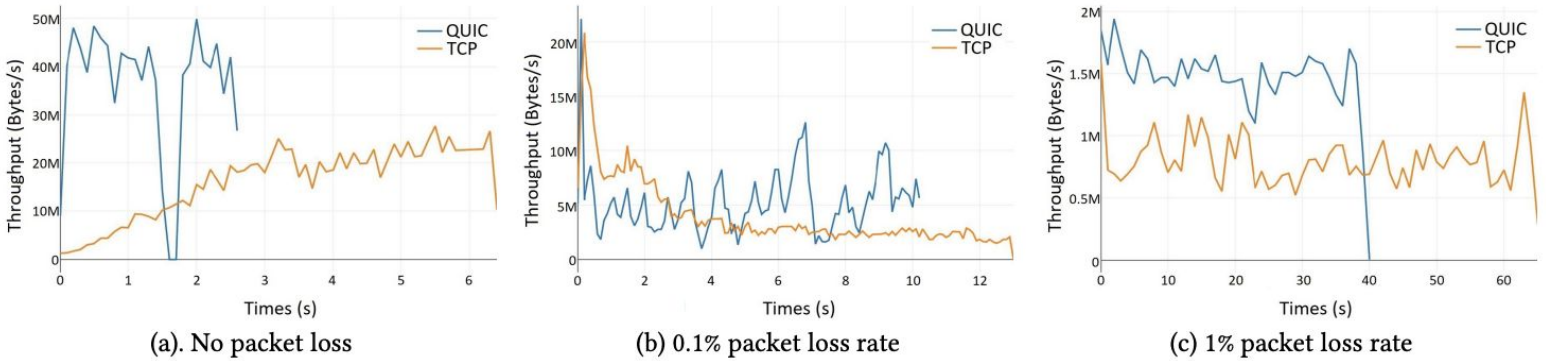
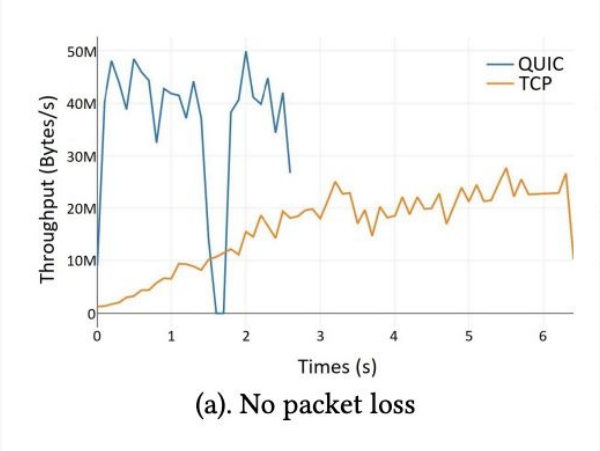


Figure 3: Throughput of QUIC and TCP with different packet loss rate with 10 ms RTT.

- Repetitions of a single file transfer with 0/0.1/1% packet loss rate
- RTT is fixed to 10ms
- QUIC always outperforms TCP but may have a higher jitter for the low loss rate cases

Out-of-paper topic: Why high throughput and high jitter?



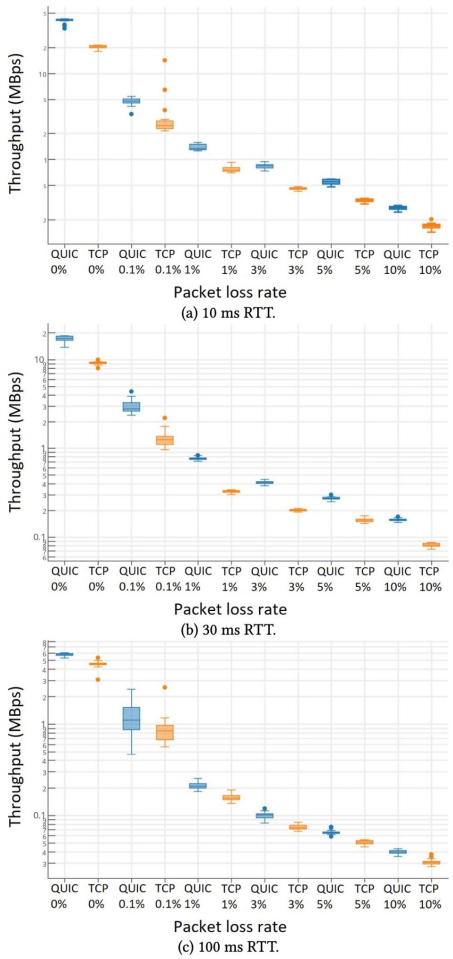
(a). No packet loss

- This paper's implementation does not limit data size of a single transfer
  - as long as cwnd allows, the kernel sends queued packets all at once (bulk)
    - RFC9000 limits this with MAX_DATA or MAX_STREAM_DATA
- Basically not having flow control makes high throughput and high jitter

Out-of-paper topic: What if the implementation is proper?

- Theoretically, QUIC will not overcome TCP in terms of throughput
  - because encryption process is computational intensive.
- QUIC will overcome TCP + TLS1.3
  - quicker handshakes, more flexible SACK (Selective ACK)
- We do not have a real-world measurement of properly implemented QUIC in kernel at this point
  - there are other in-kernel QUIC implementations such as lxin/quic, though not evaluated yet

Virtual testbed measurement: multiple RTTs with multiple packet loss rates



- QUIC always outperforms TCP
  - better loss recovery systems
- The performance deteriorates by roughly a factor of 10 for both QUIC and TCP when just 0.1% loss is introduced into the network

out-of-paper topic: QUIC vs TCP in loss recovery

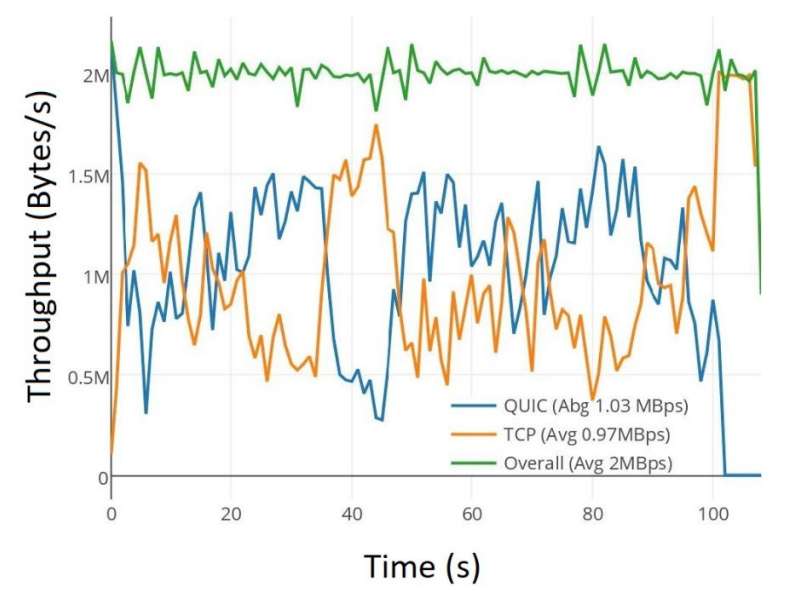
- Linux TCP employs DUPACK and SACK
  - RTO(Retransmission Timeout) is slow (usually 200ms~)
  - Trigger retransmission with 3 duplicate ACKs (DUPACK)
    - also shrink cwnd for congestion avoidance
  - normal ACKs only notify a single packet number…
  - Selective ACK notifies which packets are missing
    - Less insufficient retransmission!

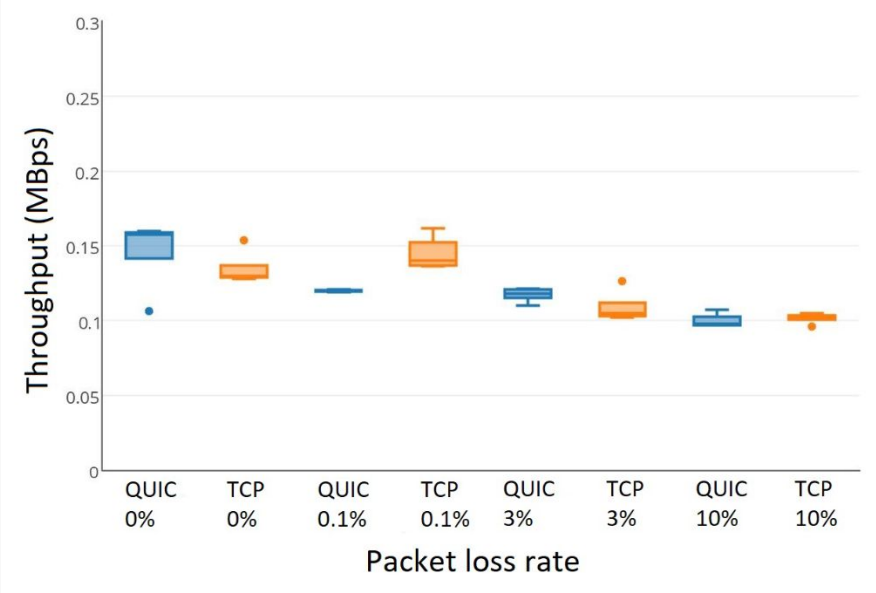
out-of-paper topic: QUIC vs TCP in loss recovery

- QUIC employs PTO and ACK Ranges
  - RTO is slow
  - Set Probe TimeOut (usually 1.125 RTT) when sending packets
    - If an ACK doesn’t come back by PTO, retransmit
      - PTO increases exponentially until the maximum tryout
    - Faster than the TCP’s DUPACK (PTO vs waiting for 3 ACKs)
  - QUIC also has a better version of SACK (no limit for packet indication)
    - even with huge loss, QUIC is expected to recover faster

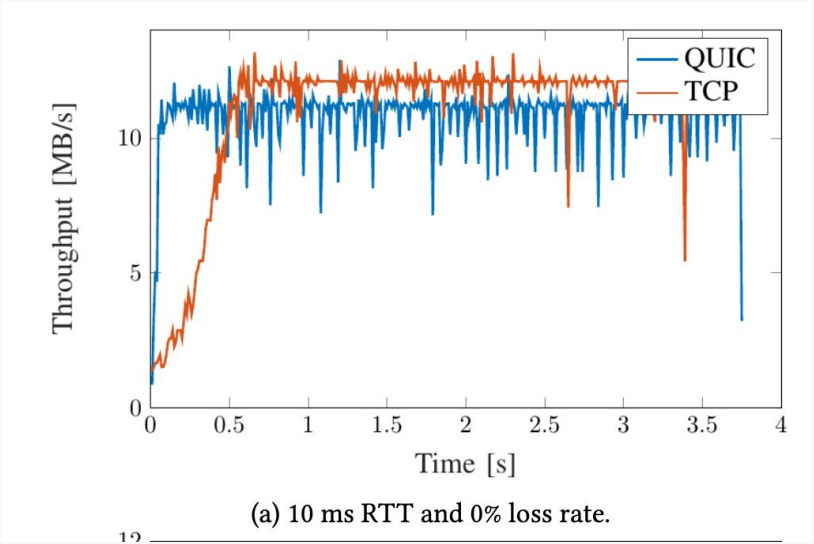
Virtual testbed: Link sharing

- Running QUIC and TCP in two virtual machines
- the bandwidth is fixed to 2MBps
- the link is shared fairly between the two because both use the same CCA

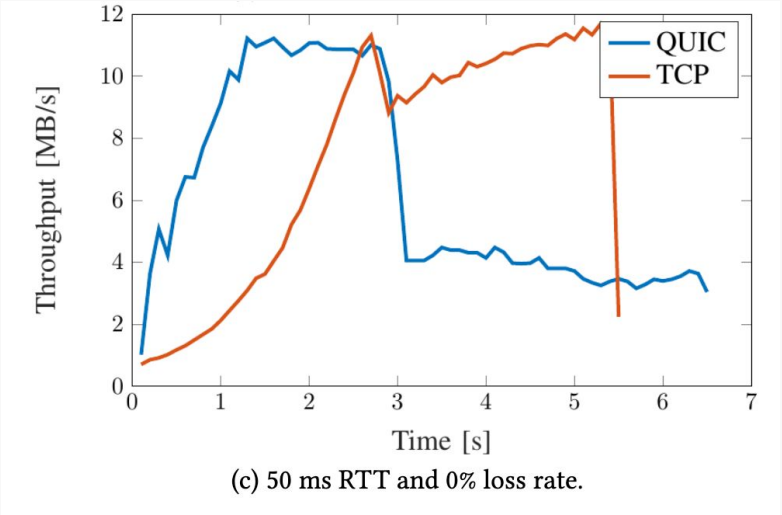




Wireless testbed: 0% loss rate



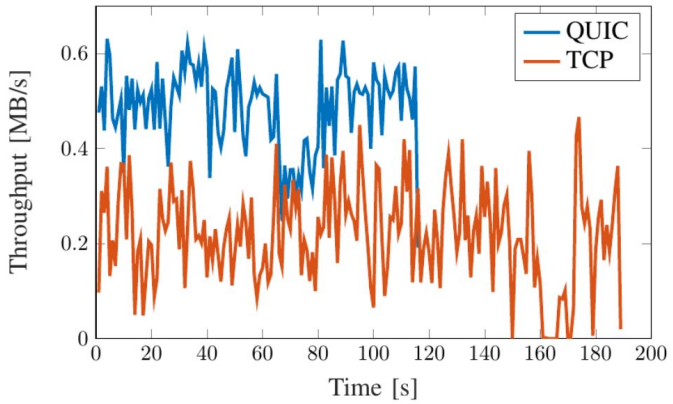
(a) 10 ms RTT and 0% loss rate.



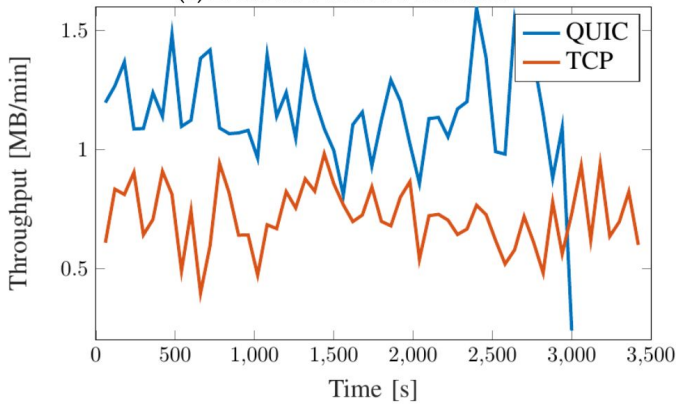
(c) 50 ms RTT and 0% loss rate.

- In a normal network situation (0 packet loss), QUIC reaches its peak throughput quicker than TCP
  - quicker handshake, even including TLS

Wireless testbed: 10% loss rate



(b) 10 ms RTT and 10% loss rate.



(c) 100 ms RTT and 10% loss rate.

- QUIC has higher throughput over TCP
  - better loss recovery (faster retransmission, faster recovery)

Conclusion

- QUIC outperforms TCP in lossy networks
- QUIC also completes handshake quicker
- In-kernel QUIC with no encryption has higher performance in most situations
    - proper QUIC vs TCP+TLS1.3 will be the same result, but there’s no measurement