A High-Speed QUIC Implementation

Nikita Tyunyayev, Maxime Piraux, Olivier Bonaventure, Tom Barbette
UCLouvain
Louvain-La-Neuve, Belgium
firstname.lastname@uclouvain.be

ABSTRACT

Several implementations of the QUIC protocol exist. Unfortunately, they generally lack behind TCP ones in terms of performance as TCP stacks have been undergoing years of optimizations. In this work, we propose `picoquic-dpdk`, a modified version of picoquic that bypasses the Linux kernel networking stack using the DPDK library, improving the throughput by a 3x factor. We compare our implementation against several QUIC stacks and TCP+TLS and demonstrate that it outperforms all tested QUIC stacks and matches TCP+TLS even with common TCP optimizations.

CCS CONCEPTS

• **Networks** → **Transport protocols**; *Network protocol design*; *Network experimentation*; **Network measurement**.

KEYWORDS

QUIC, DPDK, picoquic, 100GbE

1 INTRODUCTION

QUIC is a feature-rich transport layer protocol developed atop UDP. Compared to TCP, it features a faster handshake, multiplexing through the stream abstraction, connection migration and better flexibility by encoding control and application data in the packet payload. Several QUIC implementations exist [1] but are generally slower than TCP ones, which often benefit from years of optimization. It is not the case for QUIC, which often requires higher CPU utilization to achieve similar goodput[5].

The contribution of this paper is twofold. First, we identify bottlenecks of existing stacks and we present `picoquic-dpdk`, a kernel-bypass version of the `picoquic` implementation which improves performance by a 3x factor. Second, we compare and survey our implementation to several QUIC implementations: `picoquic`, `msquic`, `quicly`, and `quiche`, and a TCP+TLS stack. Our work thus provides a reference on QUIC stacks performance. Beyond previous surveys[6], we design an efficient open-source implementation, specifically address the execution model, and demonstrates how QUIC performance can be on par with TCP+TLS. We conclude with leads on further performance improvements and future research prospects.

[1] https://quicwg.org/

If you cite this paper, please use the ACM CoNEXT reference: Nikita Tyunyayev, Maxime Piraux, Olivier Bonaventure, Tom Barbette. 2022. A High-Speed QUIC Implementation. In CoNEXT Student Workshop 2022 (CoNEXT-SW '22), December 9, 2022, Roma, Italy. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3565477.3569154.

2 BACKGROUND

QUIC relies on UDP and has more limited hardware offload compared to TCP, which can use TCP segmentation offload (TSO) when the NIC supports it. With this technique, large TCP segments are created and passed to the NIC which handles their segmentation, reducing the number of system calls involved. Other TCP optimisations exists such as Generic Receive Offload (GRO) and Large Receive Offload (LRO) to reduce the number of system calls involved in receiving packets. As QUIC encrypts headers, implementing hardware-assisted gather technique such as LRO would need to offload an important part of the stack to the NIC.

Even with software and hardware optimizations, relying on the kernel has drawbacks. The kernel networking has several inefficiencies : (i) the NIC issues interrupt when new packets are available, which causes context switches. Likewise, the application relies on system calls to receive data, which again causes context switches. The cost of the interrupts can be mitigated by using polling, (ii) when a packet is received, the kernel wraps it inside a structure called *sk_buff* for processing. This structure is large and thus has an important impact on the memory footprint of the execution: higher levels of caches that are slower can be required, (iii) there is a copy of the packet between user space and kernel space.

Kernel Bypass. DPDK is designed to accelerate packet processing workloads. It bypasses the kernel networking stack and features performance optimizations such as Poll Mode Drivers (PMD), efficient and NUMA-aware memory allocation, zero-copy, ease the use of huge pages, and more.

When using DPDK, we can receive and process packets without the intervention of the kernel, and thus we avoid its inefficiencies. We avoid the interrupts by relying on polling, the context switches by processing the packets in userspace, and the *sk_buff* structure by replacing it with *mbuf*, the DPDK equivalent. We note that AF_XDP could be an alternative choice but its performance is still lower than DPDK[3]. Moreover DPDK itself supports an XDP-based driver providing both implementations at once.

3 IMPLEMENTATION

We developed `picoquic-dpdk`, a version of `picoquic` that integrates the DPDK library. The goal was to provide a high-speed QUIC implementation. We chose `picoquic` because it is a minimal implementation of the protocol closely following the IETF standard and is therefore easily extendable. It is written in C, as DPDK itself[4] easing our integration. `picoquic` is also well maintained and feature-rich, supporting latest features of QUIC such as multipath and FEC.

UDP stack. The Linux kernel networking stack provides a simple socket abstraction for the developers. The networking stack constructs the packets for the user. As DPDK by-passes the Kernel

stack, it looses address resolution, the management of UDP sessions, as well as inserting headers between chunks of payload. We built a simple UDP stack compatible with both IPv4 (with ARP address resolution) and IPv6 (with ICMP/NDP address resolution) to resolve unknown addresses. It then encode Ethernet, IP and UDP frames when sending packets, track seen addresses when receiving packets and compute fields such as the lengths and checksums.

Main loop. We modified the send/receive loop to work with batches of packet, a technique to enhance packet processing performance [1]. We receive a maximum of 32 packets and process them in the `picoquic` stack. We then construct packets that need to be sent and flush them at once to the TX queues.

4 EVALUATION

We measure the rate at which a client and a server can exchange data over a 100Gbps link connecting two Intel Xeon Silver 4314 CPUs running at 2.4GHz. Data is never read nor written to disk to avoid bottlenecks.

We compare our improved version of `picoquic` with the original `picoquic`, `quiche`, `quicly` and `msquic`. The last ones were chosen because they are up-to-date industry-oriented implementations. Furthermore, msquic claims to be a fast QUIC implementation focusing on performance. As a TCP+TLS stack we use picoTLS which relies on openSSL for its encryption. All stacks are configured to use AES-128-GCM as the AEAD cipher.

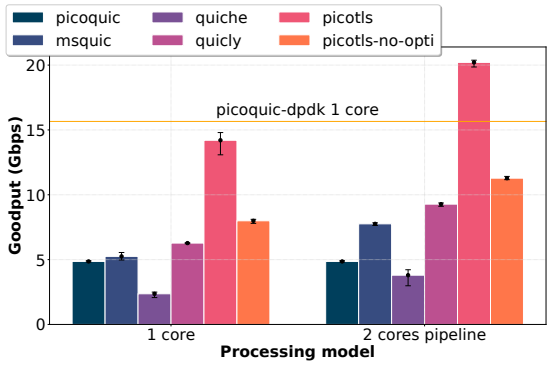

Figure 1: Single-connection performance comparison

Figure 1 illustrates the different stacks' goodput for a single connection with different processing models. The workload is a download of 20GB for the QUIC stacks and a 30 seconds download for `picoTLS`. `picoquic-dpdk` uses a single core for both processing and I/O. However, socket-based stacks may use a pipeline of cores to run the kernel interrupts and processing in addition to a core for the application itself. `msquic` uses a pipeline approach for the application itself, therefore it may use an extra core to separate the user level I/O and the stack processing, further spreading the pipeline on 3 cores. With this approach, `msquic` achieves a goodput of 10.6 Gbps.

Comparison between the QUIC stacks. Figure 1 shows that among the QUIC stacks, using a single core `picoquic-dpdk` performs best. Compared to the original `picoquic` implementation,

Task	CPU utilisation (%)				
	picoquic	picoquic-dpdk	quiche	msquic	quicly
I/O	18.7	5.7	6.54	14.39	35.59
crypto	36.15	44.16	20.49	27.55	42.18
QUIC protocol	37.16	41.24	67.96	52.8	18.88

Table 1: Usage of CPU cycles for the server

we see a 3x improvement. This improvement over `picoquic` is explained by the profiling summarized in Table 1. We profiled the execution of the send/receive loops of the different stacks' servers. The table shows a reduction of I/O from 18.7% to 5.7% between `picoquic` and `picoquic-dpdk`, leaving cycles for the encryption and protocol handling. We see a significant disparity in the distribution of the CPU cycles used for each task among the QUIC stacks. A direct comparison is not straightforward since the goodput achieved varies between the stacks. Some costs, such as encryption, will naturally require a larger part of the CPU cycles for the stacks that reach a higher goodput. Nonetheless, we can learn from the best-performing stacks: `quicly` and `msquic`. Both of them rely on batching to reduce the cost of several tasks: sending packets, encryption, and construction of packets.

The TCP+TLS stack. Figure 1 shows that `picoTLS` is performing nearly as fast as `picoquic-dpdk` with a single core. Disabling the TCP optimizations mentioned in section 2, we show the performance of `picoTLS` is reduced by a factor of 2.

Handshake speed. With requests for a 0-byte payload, `picoquic-dpdk` can handle up to 6000 request/seconds, while `picoquic` performs around half times lower.

CONCLUSION AND FUTURE WORK

Bypassing the Linux kernel improves the performance of a QUIC implementation. We showed that `picoquic-dpdk` outperforms other QUIC implementations, even those focusing on performance, such as `msquic` and `quicly`. In further work we will bring support for multipath[2] and migration, currently unsupported by the sharded scaling approach of our stack.

We showed that `picoquic-dpdk` could slightly outperform the optimized kernel TCP stack with equal CPU resources in a goodput-based workload. In future work, we will address the limitations of our evaluation, such as considering the impact of network delay and more factors like the number of connections and CPU cores on top of the processing pipeline. We will also explore further the handshake performance of QUIC and TCP+TLS stacks.

The `picoquic-dpdk` implementation is available at *https://github.com/UCLouvain-ENSG/picoquic-dpdk*. Our test-suite is fully available to enable other researchers to reproduce the results and observe the evolution of the performance of QUIC and TCP stacks. The test suite is fully automated and can be easily extended to add other test cases and more implementations.

REFERENCES

[1] T. Barbette, C. Soldani, and L. Mathy. 2015. Fast userspace packet processing. In *Proc. ACM/IEEE ANCS'15*. ACM/IEEE.

[2] Quentin De Coninck and Olivier Bonaventure. 2017. Multipath quic: Design and evaluation. In *Proceedings of the 13th international conference on emerging networking experiments and technologies*. 160–166.

[3] T. Høiland-Jørgensen, J. Dangaard Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller. 2018. The express data path: Fast programmable packet processing in the operating system kernel. In *Proc. ACM CoNEXT'18*. 54–66.

[4] M. Piraux, Q. De Coninck, and O. Bonaventure. 2018. Observing the evolution of QUIC implementations. In *Proc. of the EPIC'18 Workshop* (Heraklion, Greece) *(EPIQ'18)*. ACM, New York, NY, USA, 8–14. https://doi.org/10.1145/3284850.3284852

[5] T. Shreedhar, R. Panda, S. Podanev, and V. Bajpai. 2022. Evaluating QUIC performance over web, cloud storage and video workloads. *IEEE Trans. on Network and Service Management* 19, 2 (2022), 1366–1381. https://doi.org/10.1109/TNSM.2021.3134562

[6] 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*. ACM, 21–27.