Dissecting QUIC implementation performance

Xiangrui Yang[1], Lars Eggert[2], Jörg Ott[3], Steve Uhlig[4], Zhigang Sun[1], Gianni Antichi[4]
[1]National University of Defense Technology, CN [2]NetApp
[3]Technical University of Munich, DE [4]Queen Mary University of London, UK

Introduction: Recently introduced Network Interface Cards (NICs) with programmable hardware components, e.g., FPGA, can help to save the host CPU cycles from expensive computation tasks. For this reason, nowadays they are becoming commonplace in both datacenters and backbone networks [3]. Recognising this aspect, researchers have been looking into leveraging programmable NICs for TCP offload, load balancing, consensus protocols and key-value stores, to name a few.

In this poster, we explore their role in the context of QUIC [4], which is likely to transport a large fraction of bytes on the Internet at the end of this year [2]. Previous work [1] has studied the impact of just one specific component of QUIC, i.e., crypto. In contrast, we argue that an in-depth understanding of the costs associated to all of its components is a necessary step towards a efficient offloading on programmable NICs.

Experiments: We quantify the CPU overhead of three open source implementations of QUIC, i.e., quant, quicly and picoquic. All of them are written in C and support the latest IETF draft QUIC v25/27. We ran our experiments using two Intel Xeon Silver servers (from now on we call them A and B) connected via dual port 10G Intel NICs. Traffic departing from A would reach B and then return back to A. Both the QUIC server and client were running on A and pinned to a different CPU, while in B we installed TLEM [5] to emulate packet re-ordering and/or loss. The key lessons learned are:

- Data copy between user and kernel space costs around 50% of total CPU usage. This can be avoided by using kernel-bypass techniques as adopted in quant.
- With kernel-bypass optimization, crypto operations become the new bottleneck, by pushing the CPU resources usage up to 40% per connection.
- The amount of packet reordering and/or packet loss have a significant influence on the CPU usage. This is dependent on the specific algorithm being implemented for dealing with out of order packets.

Lesson Learned #1: use kernel-bypass and offload crypto operation. Quant reaches 7x higher throughput than quicly and picoquic using similar amount of CPU resources (Tab. 1). This is the effect of using a kernel-bypass mechanism, i.e., netmap, to deliver packets to the application. Fig. 1 shows the breakdown of the total CPU usage for the three QUIC implementations we studied. While for quicly and picoquic the majority of the cost is associated with the data passing between kernel and user space, with quant the overhead

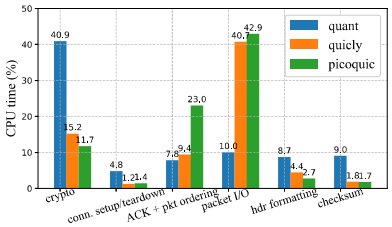

Figure 1. (left) CPU usage breakdown of the client.

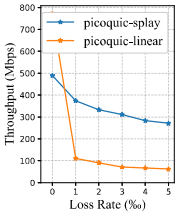

Figure 2. (right) Picoquic throughput under packet loss.

on CPU comes mostly from the crypto functionality. Since quant achieves higher throughput compared to picoquic and quicly, it is reasonable that the CPU overhead introduced by packet-level process (e.g., checksum) goes higher.

Table 1. Maximum achievable throughput vs CPU usage.

	quant	quicly	picoquic
throughput	3696Mbps	463Mbps	489Mbps
CPU usage	58%	54.8%	60.4%

Lesson Learned #2: offload the per-packet reordering process. We ran two versions of picoquic (commit ID: 2e5c3c3 & 50c7e17) under different levels of packet reordering. The former version uses a self-balancing binary search tree while the latter employing linear searching to reorder packets upon arrival. We ran our experiments imposing different percentages of packet reordering. Fig. 2 shows that the throughput drops more rapidly when using linear searching, though it performs better in the best case scenario (no packet reordering). CPU profiling results indicate that this is mainly caused by the complexity difference between linear and splay tree search: $(O(n^2)$ vs $O(logn)$, given n is the number of frames being handled). However, linear search performs better w/o packet reordering since it always hits the target in the first round of the loop $(O(1))$.

References

[1] Deval et al. 2019. Technologies for accelerated QUIC packet processing with hardware offloads. In *Google Patents*. US Patent.
[2] Eggert, Lars. 2020. Towards Securing the Internet of Things with QUIC. In *DISS*. NDSS.
[3] Firestone et al. 2018. Azure accelerated networking: SmartNICs in the public cloud. In *NSDI*. USENIX.
[4] Langley et al. 2017. The QUIC transport protocol: Design and Internet-scale deployment. In *SIGCOMM*. ACM.
[5] Rizzo et al. 2016. Very high speed link emulation with TLEM. In *LANMAN*. IEEE.