석 사 학 위 논 문

Master's Thesis

유니커널의 고성능 I/O를 활용한 QUIC 프로토콜의 성능 개선

Leveraging Fast I/O of Unikernel in QUIC Protocol

2020

허 재 석 (Jaeseok Huh)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

석 사 학 위 논 문

유니커널의 고성능 I/O를 활용한 QUIC 프로토콜의 성능 개선

2020

허 재 석

한 국 과 학 기 술 원

전산학부

유니커널의 고성능 I/O를 활용한 QUIC 프로토콜의 성능 개선

허 재 석

위 논문은 한국과학기술원 석사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2020년 12월 14일

Leveraging Fast I/O of Unikernel in QUIC Protocol

Jaeseok Huh

Advisor: Sue Moon

A dissertation/thesis submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Daejeon, Korea
December 14, 2020

Approved by

Sue Moon
Professor of School of Computing

The study was conducted in accordance with Code of Research Ethics[1].

1) Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my dissertation contains honest conclusions based on my own careful research under the guidance of my advisor.

MCS 허재석. 유니커널의 고성능 I/O를 활용한 QUIC 프로토콜의 성능 개선. 전산학부. 2020년. 27+iii 쪽. 지도교수: 문수복. (영문 논문)
Jaeseok Huh. Leveraging Fast I/O of Unikernel in QUIC Protocol. School of Computer Science. 2020. 27+iii pages. Advisor: Sue Moon. (Text in English)

초 록

Google에서 작은 프로젝트로 시작한 QUIC이 기존의 TCP/HTTPS 프로토콜 스택을 대체할 프로토콜로 등장하였다. 지연시간 감소 및 HOL (Head of Line) Blocking의 제거 등 QUIC의 여러가지 장점에 힘입어, QUIC은 현재 최소 9.1%의 인터넷 트래픽을 차지하고 있는 것으로 추정된다 [3]. 하지만, QUIC이 user space에 자리하게 되면서, QUIC의 높은 CPU 사용량에 대한 논의가 꾸준히 제거되어 왔다. 이 논문에서는 유니커널을 활용하여 QUIC I/O의 CPU 사용량을 줄이는 접근 방식을 논의한다. 구현한 2개의 프로토타입에서 실험 환경에 따라 QUIC I/O가 1.29%에서 35.94%의 CPU 사용 감소를 보였으나, QUIC의 다른 부분에서 그러한 성능 개선이 상쇄되었다.

핵 심 낱 말 QUIC, 유니커널, I/O

Abstract

Beyond its inception as a toy project at Google, QUIC has emerged as a capable replacement for the traditional TCP/HTTPS protocol stack. Due to its manifold benefits including latency reduction and the elimination of HOL (Head of Line) blocking, QUIC is now estimated to account for at least 9.1% of the Internet traffic [3]. However, there has been a concern with its high CPU consumption as it is placed in user space to avoid entrenchment. In this thesis, we examine a Unikernel approach to alleviate QUIC's higher CPU usage with I/O. Our two prototypes show reduced CPU time in the I/O of QUIC implementations--ranging from 1.29% to 35.94% depending on scenarios--while the reduction is offset by increased CPU time in other parts.

Keywords QUIC, Unikernel, I/O

Contents

List of Tables

List of Figures

Chapter 1. Introduction

The Internet has become not only faster but also more reliable over 50 years [17]. It has been thanks to decades of painstaking efforts from networking communities, since its birth. Today's Ethernet, TCP/IP, and HTTP(S) are hard-earned fruits of such a decades-long endeavor.

On top of IP, TCP is placed at the transport layer as a connection-oriented protocol. To ensure the reliability, it mandates the three-way handshake at the beginning [18]. To further provide security, it is often topped with a "secure" protocol such as SSL and TLS in the security layer. At the application level, HTTP(/1, /1.1, and /2) serves the vast majority of web contents.

While they have taken firm root in the networking literature and infrastructure, they have faced constant pressures for change. As for TCP, many transport protocols have been proposed to fulfill demands beyond its capability; however, their wide deployment has not been seen due to protocol and implementation entrenchment [2]. SSL and its successor TLS have a long history of deployment as well. Under the TCP/TLS scheme, it is impossible to achieve bona fide "security" and backward compatibility at the same time with the existence of discovered flaws. Ceasing compatibility support for outdated versions not only poses the chicken-and-egg-problem with clients and servers but also comes at significant business costs. Thus, even the deployment of necessary modifications took years [19], if ever completely, and that of unnecessary ones has been discouraged [2].

In the meantime, QUIC (Quick UDP Internet Connections), which was initially born as a toy project [1] in Google, has emerged as an alternative to the TCP/TLS stack. QUIC—as its full name suggests—uses UDP as a substrate. Instead, it incorporates the TCP's services including reliable delivery, multiplexing, and congestion control into itself in user space, rather than kernel space as is the case with TCP. Still, this difference allows its faster deployment and evolution, which was the main goal of [2]. Furthermore, the adoption of UDP eliminates the Head-Of-Line (HOL) blocking at the transport layer, which has been a concern for HTTP(S) services in the presence of packet loss. Also, QUIC provides a mechanism to establish a 0-RTT secure connection to the known servers, provided that the client has once thereto. This is because the three-handshake is no longer required, while we can count on the availability of the Internet.

Nevertheless, it too comes at cost. Chiefly because it runs in user space, there have been concerns with its higher CPU usage [20, 9] compared to the TCP/TLS stack. In this thesis, we shed light on the Unikernel approach to QUIC in order to alleviate the CPU burden that arises from the design of QUIC.

Unikernels [21] refer to a highly-specialized, single-purpose approach to build kernel VMs (Virtual Machine) or the VMs itself for cloud computing. The application binary and configuration settings and underlying runtime libraries are integrated into the kernel and sealed at compile-time against modification. Unikernels can run directly on hypervisor and employ single address space. No separation between user and kernel space contributes to improved performance owing to fewer data copy and quicker context switches and privilege transitions across the two spaces.

We port two QUIC implementations to a Unikernel named “OSv,” and present the CPU breakdown comparison in three performance testing scenarios. Compared to the Linux counterparts, our Unikernels showed the decreased CPU time for packet I/O, from 1.29% to 35.94% depending on the scenarios, while the other parts of the implementations exhibited increased CPU time (3.48%-87.10%).

The rest of the thesis is composed as follows. Section 2 introduces QUIC and Unikernels in further detail. Then, Section 3 provides i) justifications for our Unikernel approach with QUIC, ii) a brief summary of the performance of existing QUIC implementations, and iii) an explanation for our choice among many Unikernels and its porting process. Section 4 describes the methodology and settings of an experiment and compares the results between Linux and OSv servers. Section 5 leads to related work. Lastly, Section 6 concludes.

Chapter 2. Background

2.1 QUIC

QUIC was released in 2012 by Jim Roskind at Google [1]. When we say "QUIC," strictly speaking, it refers to the specification of the QUIC protocol [14]. It replaces the TCP/TLS stack while relying on UDP for middlebox delivery and TLS (v1.3 according to [14]) for encryption (see Figure 1). It also supports stream multiplexing within a single connection, absorbing part of HTTP.

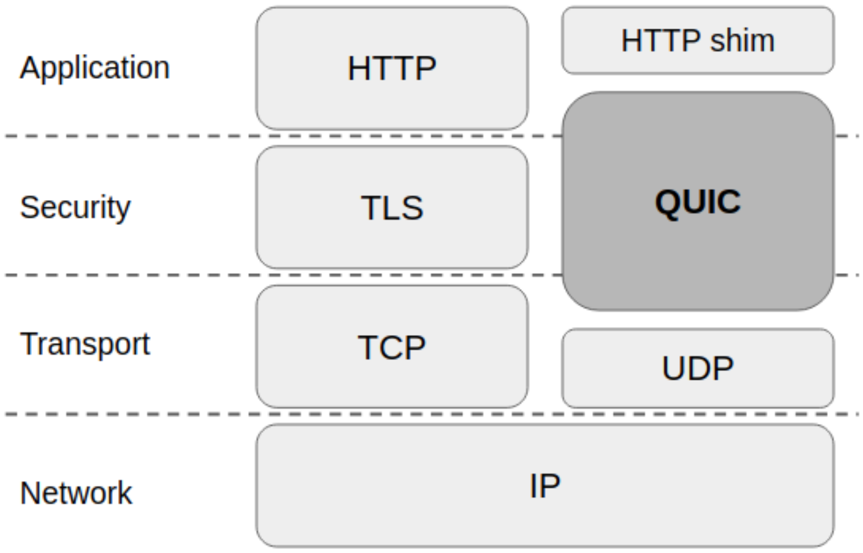

Figure 1. Traditional HTTPS stack and QUIC

QUIC has several benefits. First, as the original paper stresses [2], QUIC resolves the entrenchment of protocols and implementations by putting itself in user space and performing version negotiation at the beginning. Second, it encrypts and authenticates the header. Therefore, it avoids any tempering attempt or discrimination at the middleboxes. The intermediaries often drop packets they do not understand. Third, the client can save one RTT or more by (optimistically) sending requests right after the hello message, without seeing the ACK from the server for it (Figure 2, Left). With this, it is even possible to establish a 0-RTT secure connection, if the client has stored the long-term Diffie-Hellman public value and the certificate(s) of the server from the prior connections (Figure 2, Right). Fourth, its elimination of HOLB adds to reducing (tail) latency from the perspective of service providers. Fifth, it supports connection migration from one IP to another, as the header includes a connection ID, independent of the source and destination IP.

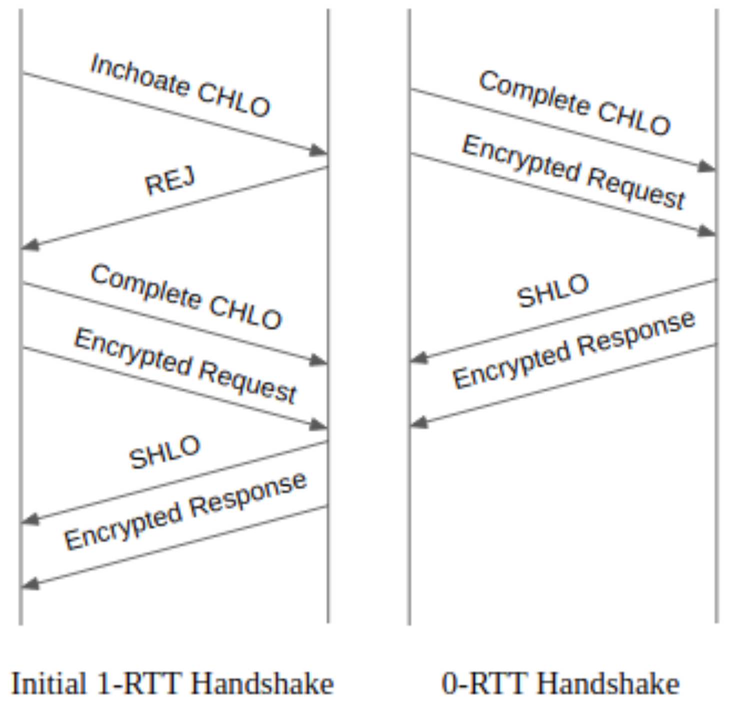

Figure 2. Timelines for QUIC's connection establishment

Since Google tossed it to the QUIC Working Group (WG) in 2016 for standardization, it has been evolving very fast. 32 versions have been drafted in 3 years and more than 20 implementations [7] have been made publicly available for testing. The WG is accepting only minor changes to finalize the protocol.

The industry has started to embrace QUIC. Google, which had extra years of deployment experience, noted that 95.3% of the video users were served with QUIC in 2016 [2]. Uber employed QUIC for its app in 2019. Recently, Facebook (and its subsidiary, Instagram) followed the wave and announced that more than 75% of traffic was QUIC [6]. For CDNs, CloudFlare, Akamai, and Fastly joined the list of deployers [11, 12, 9]. [3] estimated that 9.1% of a large ISP traffic in Europe was QUIC in August 2017. In 2020, the number should be higher, considering the deployment of the service providers.

However, deploying QUIC poses a number of challenges. First, to our best knowledge, the performance of most of the existing implementations have been largely unexamined or documented (The Microsoft QUIC team maintains the same view on the status quo [8]). The early focus of the WG was interoperability between different implementations. [5] includes goodput testing, however, it was done in a ns-3 simulator and capped at 10 Mbps. Second, it is difficult to find an optimal set of parameters. It is not only hardware- or infrastructure-dependent but also requires tweaks across the layers and machines. To our best knowledge, there is no such data publicly available except for [13] in 2017, after which QUIC underwent tens of revisions. As such, we first instrument a number of QUIC implementations that are listed by the WG [7] for performance testing (Section 3.2). In doing so, we attempt to match the parameters to our best effort, rather than extensively finding the optimal values. Third, QUIC consumes more CPU than the traditional TCP/TLS stack does, mainly because the computation is moved up to user space. To relieve the higher CPU consumption of QUIC, we examine a Unikernel approach (Section 2.2 and 4).

2.2 Unikernel

The idea of Unikernel was first introduced in 2013 [21]. Between monolithic and micro kernel, it takes an extreme position toward microkernel. A Unikernel is a highly-specialized VM running atop the hypervisor in cloud environments. It integrates application binary and configuration files—and, when applicable, language runtime and others (see Figure 3)—into a kernel image at compile-time. The compiled image is sealed against runtime modification by any malicious co-tenants occupying the same hardware. The hypervisor ensures the isolation between VMs, while each VM is limited to run only one application.

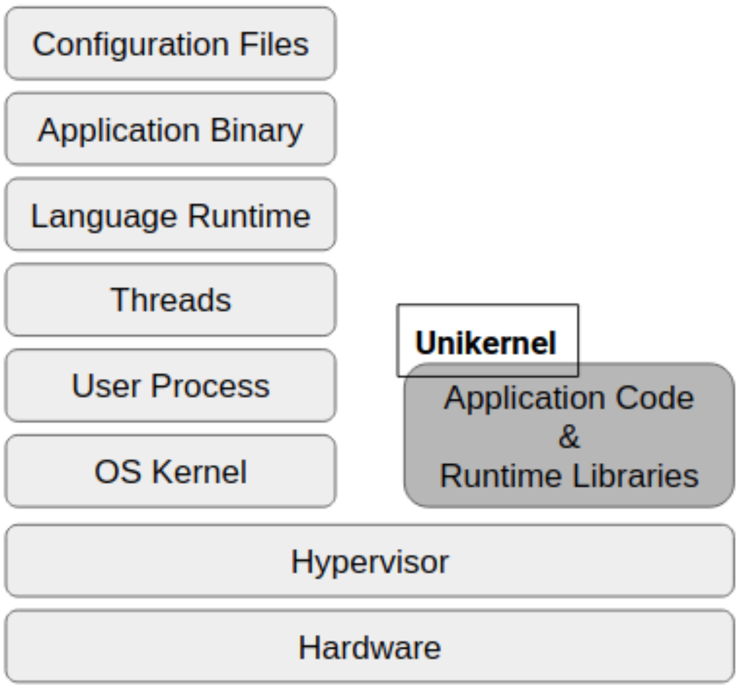

Figure 3. The layers of commonly-used VMs and Unikernel VMs

Along with pushing the protection boundary to the lowest, it exposes smaller attack surface by minimizing the size of the kernel image. The slim size leads to extremely fast booting time, down to microseconds. Cloud users can even entertain the possibility to respond before the time-out of incoming requests. Also, it has no separation between user and kernel space. This results in manifold advantages in performance, for instance, copying data directly from application to hardware and no context switches or privilege transitions between user and kernel space. As a result, a system call becomes a mere function call.

Bringing QUIC to a Unikernel entails three challenges. First, unikernels implement only a subset of POSIX. Since it assumes a single-process model, fork() is unsupported in some [10]. Some socket options are not implemented. Second, it has a driver for only limited (but general) hardware. These become problematic as we aim for performance testing, which may require leveraging specialized features from kernel and hardware. A Unikernel itself is a highly-specialized kernel, but it does not necessarily mean that specialized features we expect in kernel are already implemented in the Unikernel (see Section 5.2). Third, porting a non-trivial application into a Unikernel takes significant engineering efforts (see Section 3.3).

Chapter 3. Improving QUIC’s I/O with Unikernel

3.1 Why Unikernel

QUIC is, by design, placed in user space to enable fast evolution of the protocol and implementations and forestall the entrenchment thereof. It has allowed Google and other enterprises to successfully deploy QUIC and, supposedly, to benefit from the reduction of latency. However, the higher CPU usage of QUIC remains of concern. Google reported twice as high CPU consumption as the TCP/TLS stack [2]. On mobile devices where CPU is more scarce than desktop, QUIC’s congestion control is application-limited for 58% of the time as opposed to 7% of a desktop case [13].

We compare the CPU footprint of an Apache2 server with TLS enabled and two QUIC servers, in our 2GB file transfer scenario on a Linux machine (Section 4.1). In Figure 4, the QUIC servers, Picoquic and Quicly, consumed more CPU than Apache did, per packet. Excluding the TLS part, the transport layer alone took more time in Picoquic and Quicly than Apache2 (Table 1). The breakdown in Table 2 shows sendmsg() and recvmsg() combined accounted for 40.9% (Picoquic) and 79.0% (Quicly) in CPU time.

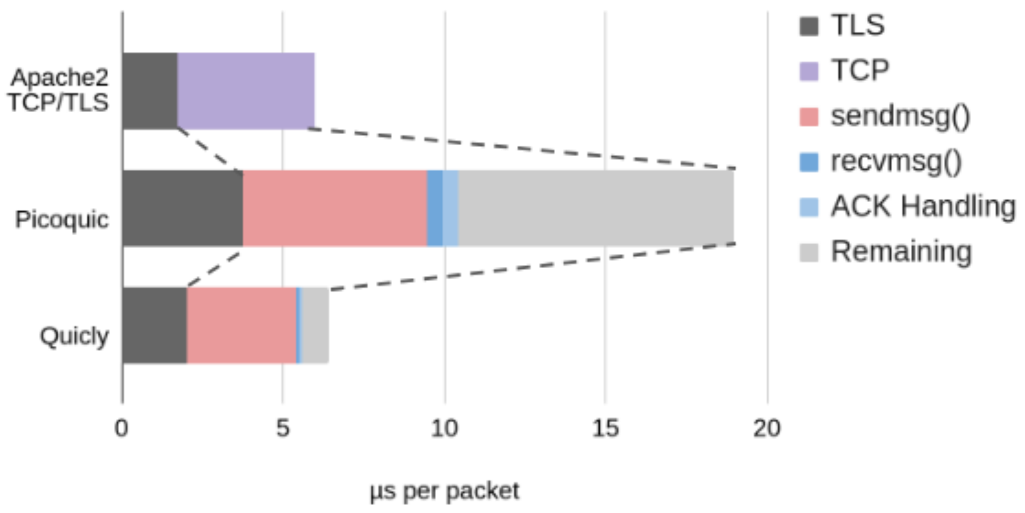

Figure 4. CPU Breakdown of Apache2, Picoquic, and Quicly

µs/pkt	Apache2	Picoquic	Quicly
TLS	1.76	3.76 (+13.6%)	2.02 (+14.8%)
Transport	4.24	15.21 (+258.7%)	4.41 (+4.0%)
Total	6.00	18.97 (+216.2%)	6.43 (+7.2%)

Table 1. Average time spent in TLS and Transport per packet

µs/pkt	Picoquic	Quicly
sendmsg()	5.71	3.39
recvmsg()	0.50	0.09
ACK Handling	0.47	0.08
Remaining	8.53	0.85
Transport Total	15.21	4.41

Table 2. Average time spent within the transport layer of Picoquic and Quicly

As in user space do its (re-)transmission and receive, ACK mechanism, and congestion control all take place, they incur additional data copy and context switches between user and kernel space in QUIC (Figure 5). In this context, we propose a Unikernel approach to relieve the CPU consumption in the I/O of QUIC implementations by exploiting the single address space design of Unikernel. First, packet payload and meta need not be copied or modified across user and kernel. Second, context switches become faster as the page table is unchanged. Third, system calls become only function calls. We examine these aspects in more detail in Section 4.

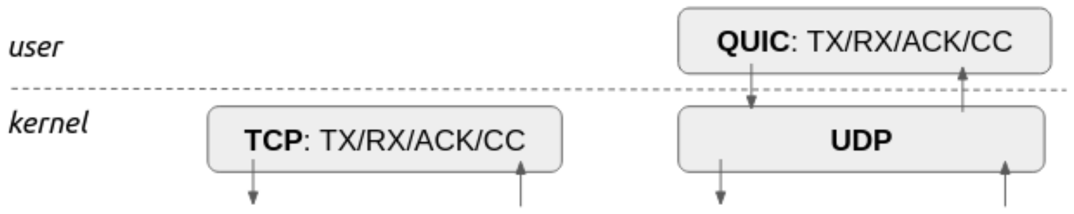

Figure 5. The placement of TCP and QUIC/UDP in user and kernel space

3.2 QUIC Implementations

Despite its rapid reception by the tech giants, there is little work done in the area of performance comparison. In this subsection, we instrument several implementations from the list [7] composed by the WG and choose a few of them to port to a Unikernel.

Among 23 QUIC implementations in the list, we consider only open-sourced ones. We opt for those of draft-29 or later (latest version being draft-32) with minimal interface required for testing, i.e., file transfer, cipher suite selection, and command line interface, as or in the “shim”. If the repository is hosted by GitHub, we pick only those with more than 100 stars. We exclude ones stating themselves “not performant” (Chromium) or “not for production” (Kiwk). Lastly, we specifically exclude Ats, which bases Apache2 for the application layer. Apache2 enforces process-level isolation for requests, thus it is not suitable for Unikernels, which adhere to the single-process design.

For performance testing, we have the client to request a single 2GB file from the server, allowing one CPU core on both sides. The client and server are run on separate machines. We use the same machine settings and compile/running parameters specified in Section 4.2.

Table 1 shows the average goodput of three runs between the client and server of their own in separate Linux machines. The first row (Nginx) is shown for comparison with the traditional TCP/HTTPs stack. For porting, we choose MsQuic and Quicly because of its performance near to that of Nginx and Picoquic as it aims to be "robust first" and is proven so under lossy environment by [4], while still shown fairly performant. We note that the goodput for Mvfst is within 10% of margin with that reported by [4], which used similar testbed settings. We note that Quic-go was tested with a loopback device in a single machine, exceptionally, due to its limited interface.

Name	Goodput (Mbps)	Maintainer(s)	Language
Apache2 (TCP/HTTPS)	1,783	Apache	C/C++
MsQuic	1,250	Microsoft	C
Mvfst	304	Facebook	C/C++
Picoquic	903	Unaffiliated retiree	C
Quiche	797	CloudFlare	Rust
Quicly	1,491	Fastly	C
Quic-go	*(w. loopback setting)* 470	Non-affiliated hobbyists	Go

Table 3. Goodput comparison between chosen implementations

3.3 OSv Unikernel

Among a number of Unikernel libraries, we choose OSv [25] to build a kernel for experiment. It has the ability to run a Linux executable without modification, although with some limitations. Also, it is shown at least as performant as Linux [26].

Porting the Implementations to OSv Unikernel

It requires considerable engineering efforts to port a complex application like QUIC implementations into Unikernels. We describe three major hurdles. First, one needs to rewrite the build scripts for the application. Unikernels put everything from configuration and application to kernel, so every library should be incorporated into the Unikernel image. Also, one should explicitly specify implicit dependencies that are resolved by a build script, for instance, those on pthread or libc. Otherwise, the built Unikernel would crash as running. This includes external libraries as well.

Second, since OSv implements only a subset of POSIX as other Unikernels, some system calls need to be removed. However, they are not all detected at compile-time because OSv does have the interface for some in the C header files without actual implementation. As such, they have to be manually inspected via debugging and then removed or replaced, or one has to implement the system calls in the Unikernel. Consequently, fork() functions are removed; and some socket options–such as ECN, UDP offloading, and IPv6–are not attempted to enable. Despite our efforts, MsQuic is discarded at this step mainly due to its complex abstraction for supporting both Windows and Linux.

Third, in the absence of a high-performance network driver, we use the one in Linux. In order to avoid another isolation overhead from Linux, we leverage KVM, which makes the host Linux act as a hypervisor and yield near-native performance [27]. Atop the KVM-enabled Linux, we use QEMU simulator with KVM components installed. Then, we use a Linux Bridge and TAP device to link the components.

After all, we successfully ported Picoquic and Quicly into an OSv Unikernel. In the next section, we examine the performance of these Unikernels.

4. Experiment

4.1 Methodology

We employed two Linux machines; one for the client and another for the server. We used only one core unless otherwise specified. The ported Unikernels were used only on the server side, and we compare them with their Linux counterparts. We used the client and the server of their own.

Following the benchmark for the traditional TCP/HTTPS stack [28], we measured goodput, response time, and Request Per Second (RPS). For each case, we performed CPU profiling using the Time Stamp Counter (TSC) of the x86 processor. The TSC provides a high-resolution CPU time clock with low-overhead. We placed RDTSC (RD for read) calls before and after function calls in the source code along with memory barriers to ensure no out-of-order-execution.

As for goodput, the client downloaded a single file from the server, as done in 3.2. We varied the file size from 2KB, which is roughly the size of most common requests in web servers, to 2GB. We ensured no disk write on both client and server side, as we observed that it became a bottleneck with the iperf testing. For the purpose of CPU warm-up, routing information insertion, and the application initialization, we discarded the first three runs out of total 33 runs and used only the remaining 30 runs for the average. The CPU breakdown is shown in per packet in this scenario.

For response time, we measured on the client the average time for establishing the connection. We used 144 clients to send--in total--100,000 requests for one byte file. For RPS, the clients requested 2KB files 100,000 times in total. In both cases, the server used only one core on its CPU whereas the clients used all the cores available to saturate the server. The CPU breakdowns are shown in per request in these scenarios.

4.2 Machine Settings

The two machines were equipped with

- Intel(R) Xeon(R) CPU X5650 @ 2.67GHz (flags are shown in suppl.)
- Intel(R) Ethernet Converged Network Adapter X520-DA2 (10GbE)
- DRAM - DDR3 24GB (server) / DDR3 20GB (client)
- Ubuntu 18.04.5 LTS (KVM enabled)
- QEMU v2.11.1 (KVM components installed)

They were connected via a 10GbE link at the same physical rack. The RTT was measured 0.194 ms (SD: 0.011 ms). For the NIC, the transport layer offloadings (i.e., TSO, TRO, GSO, GRO, and LRO) were all turned off for fair comparison. We left the IP-layer offloading (e.g., IP checksum) on. We made sure to increase the buffer size in the kernel and applications enough for gigabits per second transmission. See the supplementary materials for further details. We tested our testbed with iperf and confirmed a throughput of 9.42 Gbps for TCP and 9.52 Gbps for UDP (repeated three times, with no disk read or write).

4.3 Flags, Versions, and Parameters

Both Picoquic and quicly comes with a number of parameters for running. As for the version, we chose draft-29 as it is available in both. The version of OpenSSL was set the same as 1.1.1f. Page cache, dentries and inodes were cleared before running. We used Cubic [29] for congestion control. Note that the QUIC specification is independent of any congestion algorithm. We used AES128-GCM-SHA256 as a cipher suite. We ran all experiments in as idle a state as possible. When building, we made sure to use the same optimization flags ("-O2") and CPU flags across the machines and implementations.

4.4 Picoquic

In this subsection, we present the results with Picoquic in the goodput, response time, and RPS scenario and CPU breakdown for each case.

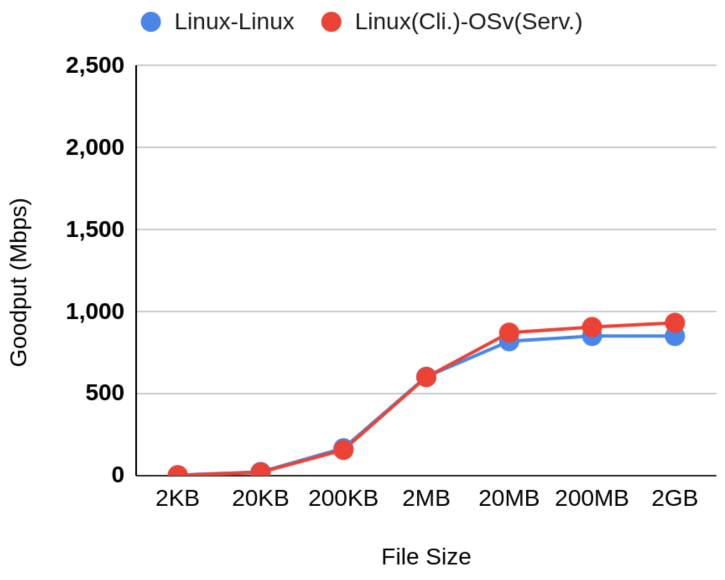

Figure 6. Picoquic Goodput

First, Figure 6 charts the average goodput of Picoquic in the 30 runs for each size. As the size of the file got bigger, the goodput increased up to 852Mbps (Linux-Linux) and 932Mbps (Linux-OSv) in the 2GB file transfer. The OSv server outperformed the Linux server by 6.41% in the 20MB, by 6.40% in the 200MB, and by 9.44% in the 2GB file requests.

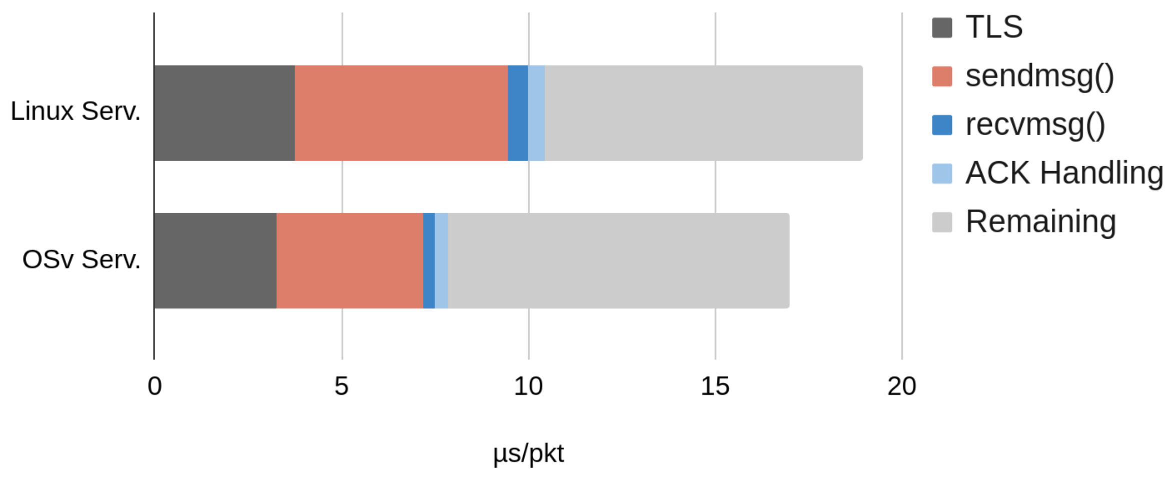

Figure 7. CPU time consumed per packet in Picoquic Goodput

µs/pkt	Linux Serv.	OSv Serv.	Change
TLS	3.76	3.24	-13.99%
sendmsg()	5.71	3.93	-31.17%
recvmsg()	0.50	0.32	-35.81%
ACK Handling	0.47	0.35	-25.55%
Remaining	8.53	9.15	7.24%
Total	18.97	16.99	-10.48%
Goodput (Mbps)	852.04	932.40	9.43%

Table 4. CPU time breakdown per packet in Picoquic Goodput

Figure 7 and Table 4 detail the CPU consumption in the 2GB file transfer. As we expected, OSv saved CPU time in sendmsg() by 31.17% and recvmsg() by 35.81%. This could be attributed to one data copy saved in the OSv implementation of those functions. We note that the overall CPU time decreased by 10.48%, a value close to the goodput improvement (9.43%), which verifies our CPU accounting. We attest the same result in the following scenarios as well.

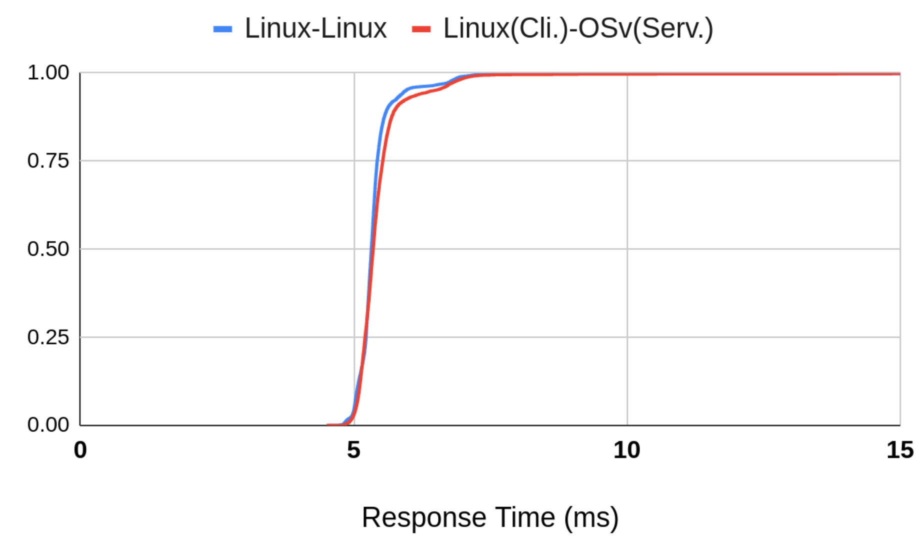

Figure 8. Picoquic Response Time

Next, Figure 8 shows the CDF for response time, defined as the average time on the client for establishing a connection with the server. It was only 0.05 ms (0.89%) slower with the OSv server than with the Linux server. In Figure 9 and Table 5, we can see the relative gains from sendmsg() and recvmsg() were diminished. In this scenario, the average size of the outgoing packets from the server, which is approximately 1 byte plus the header size now, became far smaller than that in the file transfer scenario, which was about the MTU (1,472 in our setting). On the other hand, the offset came from the remaining part, other than sendmsg() and recvmsg(). This could be the emulation overhead from QEMU. Despite the smaller packet size, Picoquic spent longer time in sendmsg() on average. This is probably because in this case the server switches more often between multiple connections and different functions than does in the goodput scenario, in which a large file is transmitted through one connection.

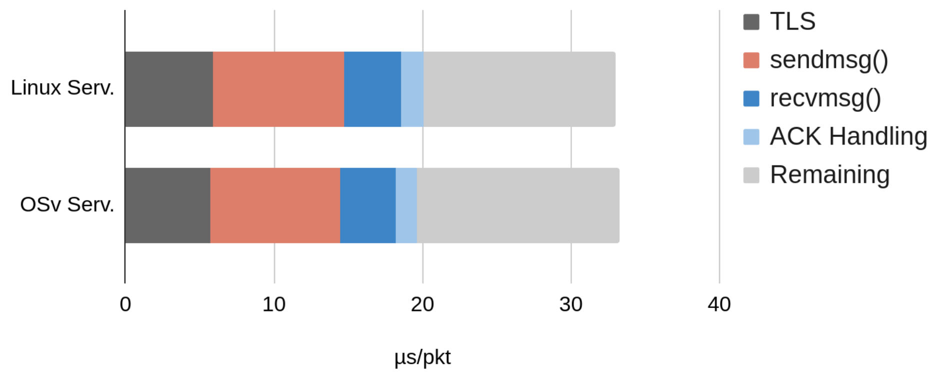

Figure 9. CPU time comparison per request in Picoquic Response Time

µs/req	Linux Serv.	OSv Serv.	Change
TLS	5.86	5.72	-2.46%
sendmsg()	8.83	8.70	-1.46%
recvmsg()	3.85	3.80	-1.29%
ACK Handling	1.50	1.44	-4.11%
Remaining	12.96	13.64	+5.21%
Total	33.01	33.30	+0.88%
Res. Time (ms)	5.42	5.47	+0.92%

Table 5. CPU time breakdown per request in Picoquic Response Time

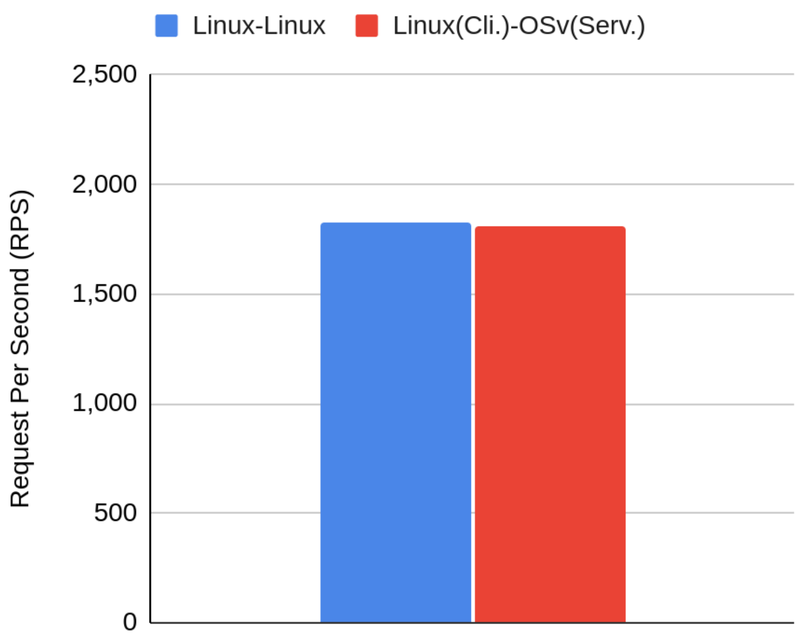

Figure 10. Picoquic RPS

Lastly, we compare the RPS of the Linux and OSv server. The client opened 144 concurrent connections, and requested 2KB file transfer to the server 100,000 times. As shown in Figure 10, the OSv server handled 0.68% less requests than the Linux server did; 1824.38 and 1812.05 rps respectively. Similarly, the OSv did save time in sendmsg() and recvmsg(), while losing in the other parts. But here, the gain was larger in sendmsg() and recvmsg(), as the average size of the packets became bigger (it was approximately 1KB plus the header size, as normal 2KB requests were served in two packets). The average time spent in sendmsg() was also longer than was in the previous scenario.

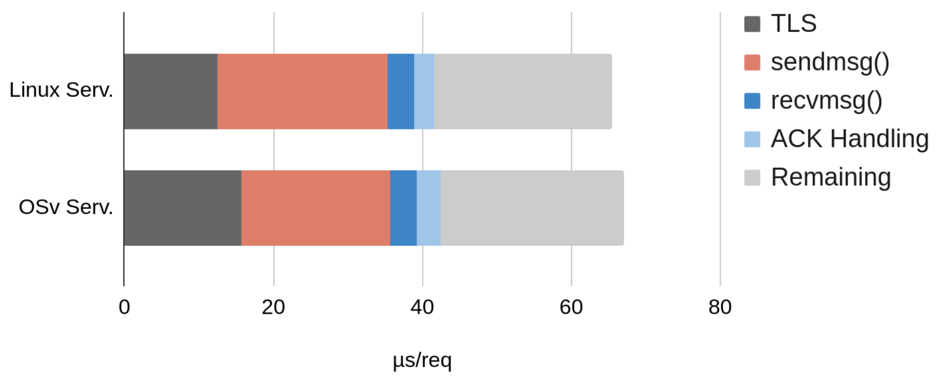

Figure 11. CPU time comparison per request in Picoquic RPS

µs/req	**Linux Serv.**	**OSv Serv.**	**Change**
TLS	12.43	15.78	26.99%
sendmsg()	22.91	19.98	-12.76%
recvmsg()	3.58	3.45	-3.64%
ACK Handling	2.75	3.26	+18.43%
Remaining	23.78	24.61	+3.48%
Total	65.44	67.08	+2.50%
RPS	1824.38	1812.05	-0.68%

Table 6. CPU time breakdown per request in Picoquic RPS

4.5 Quicly

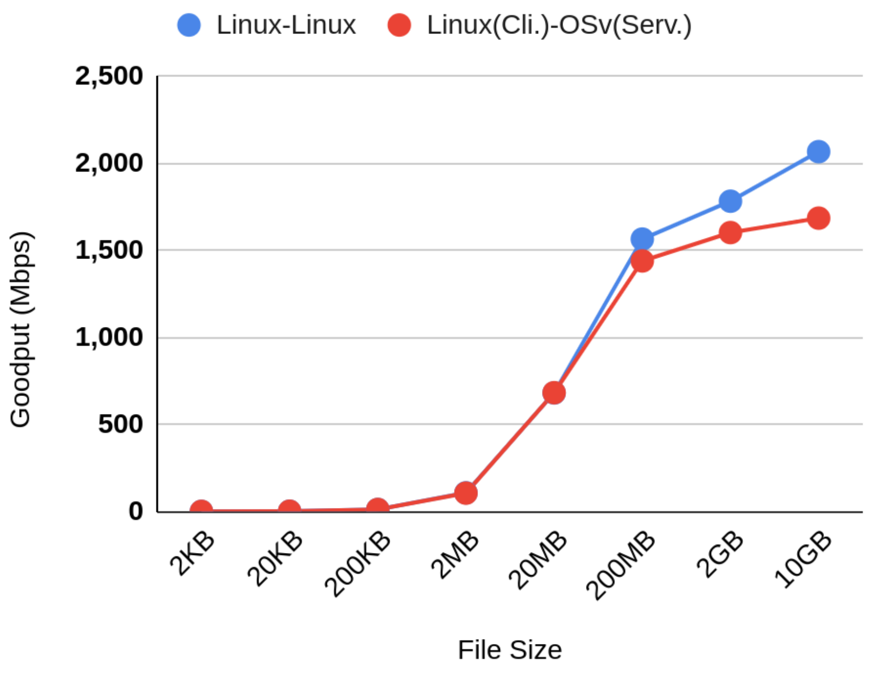

Figure 12. Quicly Goodput

On the other hand, Quicly presents different results. In Figure 8, Linux and OSv server yielded almost the same goodput until 2MB (the difference less than 2%), however, in 200MB (1,566 and 1,440Mbps; -8.02%) and 2GB (1,784 and 1,603Mbps; -10.12%), the goodput of OSv was worse than that of Linux. Additionally, we instrumented 10GB file transfer to see a further trend, but also in that case, OSv performed poorer (2,069 and 1,686Mbps; -18.48%). Similar to the case of Picoquic, Quicly spent less time in sendmsg() and recvmsg() but more in the other parts. However, due to the larger offset in the other parts, the goodput was eventually smaller. It is noted that Quicly still yielded better performance than Picoquic in both Linux and OSv.

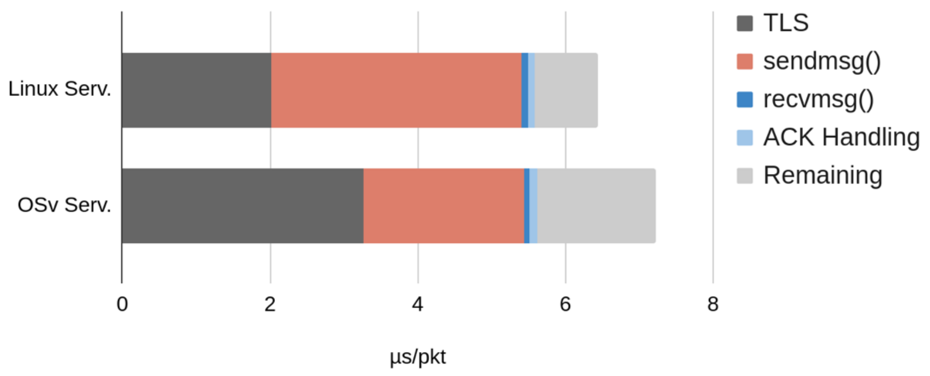

Figure 13. CPU time comparison per packet in Quicly Goodput

µs/pkt	Linux Serv.	OSv Serv.	Change
TLS	2.02	3.27	+61.77%
sendmsg()	3.39	2.17	-35.94%
recvmsg()	0.09	0.07	-27.64%
ACK Handling	0.08	0.12	+43.03%
Remaining	0.85	1.60	+87.10%
Total	6.40	7.22	+12.17%
Goodput (Mbps)	1,783.62	1,603.19	-10.12%

Table 7. CPU time breakdown per packet in Quicly Goodput

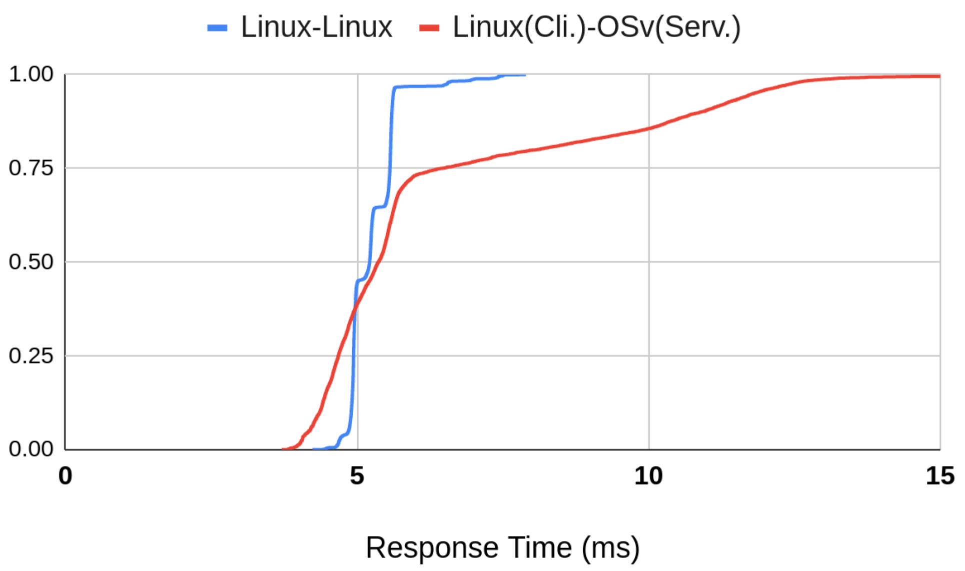

Figure 14. Quicly Response Time

Figure 13 plots the CDF of Quicly's response time. Unlike Picoquic, Quickly saw long tail latency in OSv. We suspect that this is because Picoquic is better equipped to handle multiple connections simultaneously. Quicly uses linear data structures internally for packet sequencing, whereas Picoquic uses tree-based data structures. In OSv, the average time was 6.55 ms, which was 23.2% higher than in Linux (5.32ms). As with Picoquic, the relative gains from sendmsg() and recvmsg() were declined, the increase from in other parts were larger as in Quicly Goodput, and the sendmsg() consumed a longer time on average.

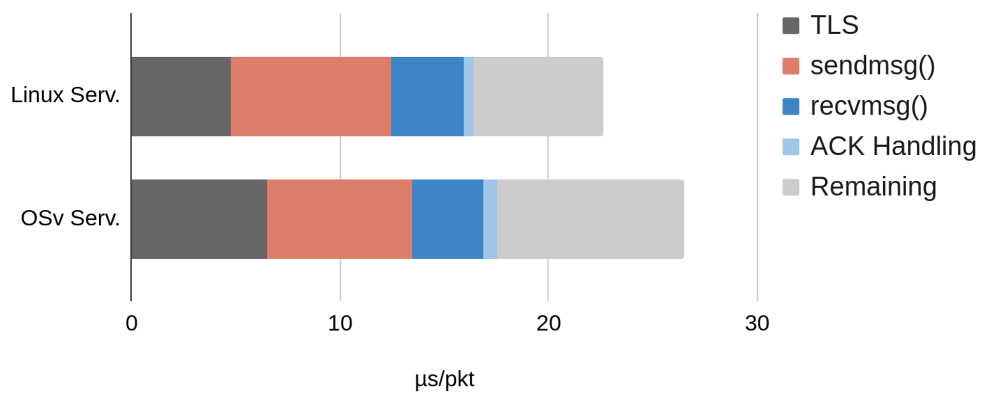

Figure 15. CPU time comparison per request in Quicly Response Time

µs/req	Linux Serv.	OSv Serv.	Change
TLS	4.78	6.48	+35.55%
sendmsg()	7.65	7.00	-8.56%
recvmsg()	3.46	3.36	-2.84%
ACK Handling	0.48	0.69	+42.82%
Remaining	6.22	8.97	+44.14%
Total	22.60	26.50	+17.26%
Res. Time (ms)	5.318	6.55	23.17%

Table 8. CPU time breakdown per request in Quicly Response Time

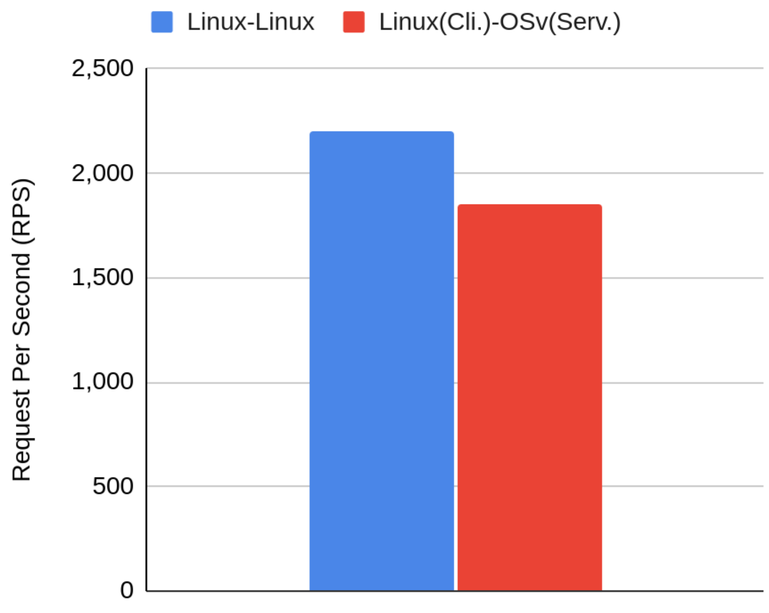

Figure 16. Quicly Request Per Second (RPS)

Finally, Figure 16 compares Quicly's RPS. In OSv, Quickly was able to process 1,849 requests per second, 16.13% less than was in Linux (2,204). The relative changes from sendmsg() and recvmsg() were similar to the previous case, however, the absolute values were higher, probably due to the bigger packet size.

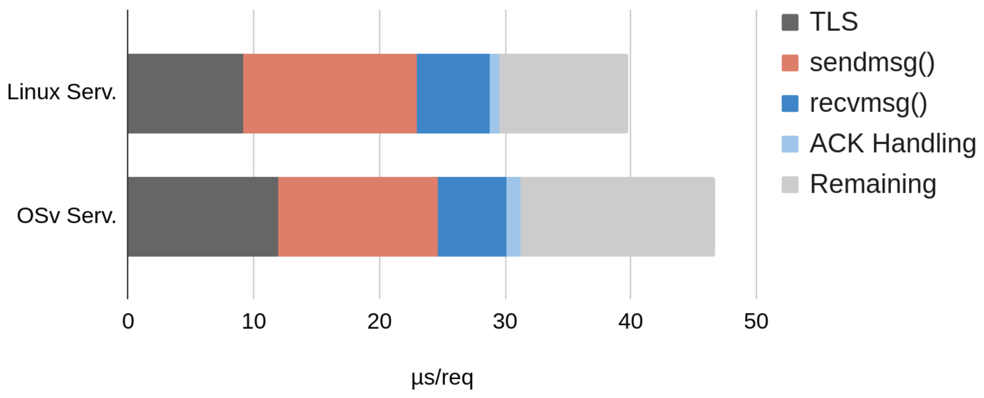

Figure 17. CPU time comparison per request in Quicly RPS

µs/req	Linux Serv.	OSv Serv.	Change
TLS	9.12	11.90	30.55%
sendmsg()	13.87	12.72	-8.31%
recvmsg()	5.74	5.49	-4.23%
ACK Handling	0.79	1.12	41.50%
Remaining	10.31	15.53	50.73%
Total	39.82	46.77	17.44%
RPS	2204.2	1848.76	-16.13%

Table 9. CPU time breakdown per request in Quicly RPS

4.6 Discussions

In all cases, sendmsg() and recvmsg() were faster in OSv than in Linux. The relative gain was greater where the average size of the packets was larger, namely, in descending order of Goodput (~MTU), RPS (~1KB/pkt), and Response Time (~1B/pkt). With Picoquic, OSv yielded the improvement of Goodput, while keeping the average response time and RPS at a similar level (+0.92% and 0.68%, respectively). With Quicly, however, OSv was significantly slower (30.6-87.1%) in the other sections than sendmsg() and recvmsg(), eroding the gain from the two.

On the other hand, ACK Handling was faster only in the Picoquic's Goodput (25.5%) and Response Time (4.1%). Also, the TLS part were generally slower in OSv and in multiple-connections scenarios (i.e. Response Time and RPS). Google noted that critical paths and data-structures were rewritten to reduce the managing cost of the states of multiple connections at a time [2, 20] and Facebook did the same for unacknowledged packets [6]. OSv has its own scheduling and caching system, and the same binary--whether specifically optimized for Linux or not--may well be executed in a different manner in OSv. We did not further investigate the higher CPU consumption in non-I/O parts of the OSv and leave it as future work.

5. Related Work

To the best of our knowledge, no systematic work has been published for the performance testing of the existing implementations. The industry and academia have mainly focused on the correct implementation of QUIC specifications [32] and the interoperability of QUIC implementation [5, 31].

For deployment, one should consider not only QUIC's higher CPU usage but also the paucity of assistance from the underlying layers. The Generic Segmentation Offload (GSO) for UDP was implemented only in Linux 4.18, which was released in August 2018. NIC and its driver and (para-)virtualized driver are also required to fully enable the feature. In this regard, Yang et al [4] explores the desirable set of primitives that a NIC shall provide by analysing QUIC implementations. Google [20] and Facebook [6] enabled offloading for their deployment, however, UDP offloading has been largely left untouched in the open source community. Despite that, we think QUIC will be seeing wider deployment as HTTP/3 comes into play.

Lastly, we note that Unikernel approach is not the only way to relieve CPU usage. The Intel DPDK [33] and netmap [34] provide means to bypass the kernel and reduce memory copy. Quant [35] adopts netmap and is shown gigabps goodput [4].

6. Conclusion

QUIC is expected to proliferate as the underlying protocol of HTTP/3. In this work, we examined a Unikernel approach to mitigate the high CPU consumption of QUIC compared to the traditional TCP/TLS stack. We instrumented 6 QUIC implementations and measured goodput, and ported two of them, Picoquic and Quicly, into Unikernel. When ported to OSv Unikernel, both did consume less CPU time in the I/O, sendmsg() and recvmsg(), from 1.29% to 35.94%.

In OSv, Picoquic showed 6.40-9.44% increase in goodput, where the requested file size was no less than 2MB. The response time and RPS remained almost the same, exhibiting a difference less than 1% compared to the Linux server. Quicly, on the other hand, the results were no better in OSv for all the three scenarios, as the gains from the I/O were offset by the increased time in the other parts. Nonetheless, both Picoquic and Quicly took less time for their sendmsg() and recvmsg() in OSv. It demonstrates that the single address design of Unikernel can be leveraged to lower the CPU consumption of QUIC.

Bibliography

[1] QUIC: Design Document and Specification Rationale https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit

[2] Langley, Adam, et al. "The quic transport protocol: Design and internet-scale deployment." Proceedings of the Conference of the ACM Special Interest Group on Data Communication. 2017.

[3] A First Look at QUIC in the Wild." International Conference on Passive and Active Network Measurement. Springer, 2018. Langley, Adam, et al.

[4] Yang, Xiangrui, et al. "Making QUIC Quicker With NIC Offload." Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC. 2020.

[5] Seemann, Marten, and Jana Iyengar. "Automating QUIC Interoperability Testing." Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC. 2020.

[6] How Facebook is bringing QUIC to billions (https://engineering.fb.com/2020/10/21/networking-traffic/how-facebook-is-bringing-quic-to-billions/)

[7] https://github.com/quicwg/base-drafts/wiki/Implementations

[8] quic-perf/draft-banks-quic-performance.md https://github.com/nibanks/quic-perf/blob/3183453f90beab68440f3c755d2c3ab143a33641/draft-banks-quic-performance.md

[9] Can QUIC match TCP's computational efficiency? https://www.fastly.com/blog/measuring-quic-vs-tcp-computational-efficiency

[10] Madhavapeddy, Anil, et al. "Unikernels: Library operating systems for the cloud." ACM SIGARCH Computer Architecture News 41.1 (2013): 461-472.

[11] cloudflare/quiche: Savoury implementation of the QUIC transport protocol and HTTP/3 https://github.com/cloudflare/quiche

[12] Introducing QUIC for Web Content https://developer.akamai.com/blog/2018/10/10/introducing-quic-web-content

[13] Kakhki, Arash Molavi, et al. "Taking a long look at QUIC: an approach for rigorous evaluation of rapidly evolving transport protocols." Proceedings of the 2017 Internet Measurement Conference. 2017.

[14] QUIC: A UDP-Based Multiplexed and Secure Transport draft-ietf-quic-transport-32 https://tools.ietf.org/html/draft-ietf-quic-transport-32

[15] Kurose, James F., and Keith W. Ross. Computer networking: A top-down approach. Addison Wesley.

[16] QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2 https://www.ietf.org/mail-archive/web/i-d-announce/current/msg66052.html

[17] SIGCOMM 2020 Keynote: Amin Vadhat: Coming of Age in the Fifth Epoch of Distributed Computing https://youtu.be/27zuReojDVw

[18] Postel, Jon. "Transmission control protocol." (1981).

[19] Google to remove support for SSL 3.0 https://www.infoq.com/news/2014/10/google-ssl3/

[20] QUIC CPU Performance https://conferences.sigcomm.org/sigcomm/2020/files/slides/epiq/0%20QUIC%20and%20HTTP_3%20CPU%20Performance.pdf

[21] Madhavapeddy, Anil, et al. "Unikernels: Library operating systems for the cloud." ACM SIGARCH Computer Architecture News 41.1 (2013): 461-472.

[22] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. 2011. Is It Still Possible to Extend TCP?. In ACM IMC.QUIC

[23] Home - DPDK https://www.dpdk.org/

[24] netmap - the fast packet packet I/O framework http://info.iet.unipi.it/~luigi/netmap/

[25] OSv - the operating system designed for cloud http://osv.io/

[26] Kivity, Avi, et al. "OSv—optimizing the operating system for virtual machines." 2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14). 2014.

[27] QEMU - ArchWiki https://wiki.archlinux.org/index.php/QEMU

[28] h2load - HTTP/2 benchmarking tool - HOW-TO https://nghttp2.org/documentation/h2load-howto.html

[29] Ha, Sangtae, Injong Rhee, and Lisong Xu. "CUBIC: a new TCP-friendly high-speed TCP variant." ACM SIGOPS operating systems review 42.5 (2008): 64-74.

[30] cloudius-systems/osv: OSv, a new operating system for the cloud. https://github.com/cloudius-systems/osv

[31] Rath, Felix, Daniel Schemmel, and Klaus Wehrle. "Interoperability-guided testing of QUIC implementations using symbolic execution." Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC. 2018.

[32] McMillan, Kenneth L., and Lenore D. Zuck. "Formal specification and testing of QUIC." Proceedings of the ACM Special Interest Group on Data Communication. 2019. 227-240.

[33] Home - DPDK https://www.dpdk.org/

[34] Rizzo, Luigi. "Netmap: a novel framework for fast packet I/O." 21st USENIX Security Symposium (USENIX Security 12). 2012.

[35] NTAP/quant: QUIC implmenetation for POSIX and IoT platforms https://github.com/NTAP/quant

Acknowledgments

My earnest gratitude goes first to my advisor, Prof. Sue Moon, for sharing her life-learned wisdom and insight and providing a warm nest for two years. I have been also blessed to have such bright, candid, and sincere people as lab mates: Heeju, Hyunjin, Jisu, Jiwan, Keunhong, and Wonsup. I sincerely thank the administrative staff—Mmes. Oksil Kim and Eunhye Go—for their dedication and seamless support.

My heart goes out to my family in Daegu, numerous people at KAIST, and all my friends in Seoul, Pangyo, and other countries, for their unwavering support and love. I thank the government of Republic of Korea for supporting my academic endeavor for 18 years up to this point. Now that I am in debt of many and society, it is not only my duty but also with deep gratitude and humility to go on a new journey to pay their generosity forward and make the world a better place.

Curriculum Vitae

E-mail - jaeseok.h@kaist.ac.kr

Educations

Mar 2012 – Feb 2015	Korea Digital Media High School, Ansan
Mar 2015 – Feb 2019	B. S. in Computer Science, Hanyang University, Seoul
Feb 2019 – Feb 2021	M. S. in Computer Science, Korea Advanced Institution of Science and Technology, Daejeon