Leveraging Fast I/O of Unikernel in QUIC Protocol

Master's Thesis Defense

Jaeseok Huh

Committee

Prof. Sue Moon | Prof. Dongman Lee | Prof. Jaehyuk Huh

Dec 14, 2020

Advanced Networking Lab
KAIST

Leveraging Fast I/O of **Unikernel** in **QUIC** Protocol

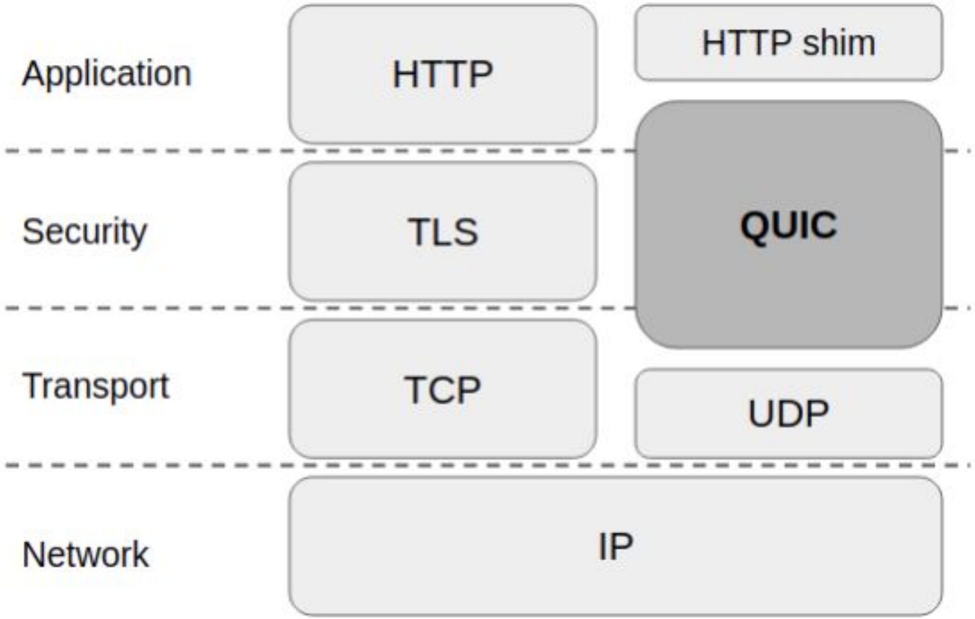
Master's Thesis Defense
Jaeseok Huh

Committee

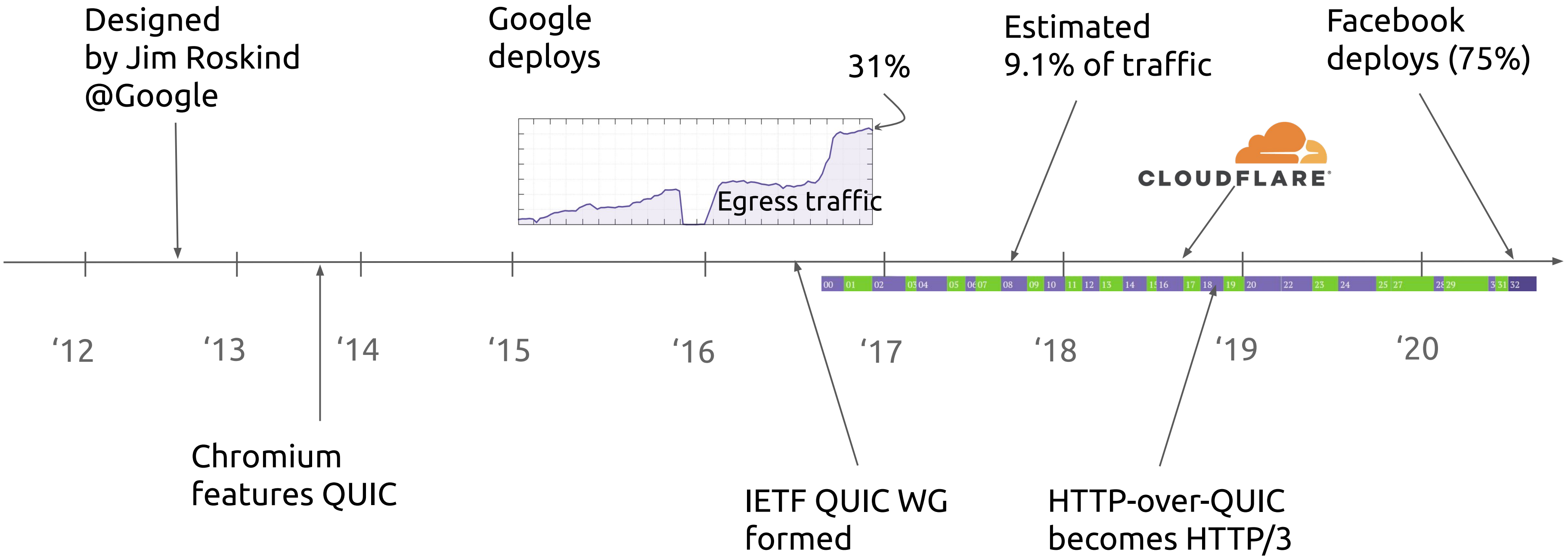
Prof. Sue Moon | Prof. Dongman Lee | Prof. Jaehyuk Huh

Dec 14, 2020

The QUIC (Quick UDP Internet Connections) Protocol



A Brief History of QUIC



Langley et al. "The quic transport protocol: Design and internet-scale deployment." SIGCOMM ‘17

Benefits of **QUIC**

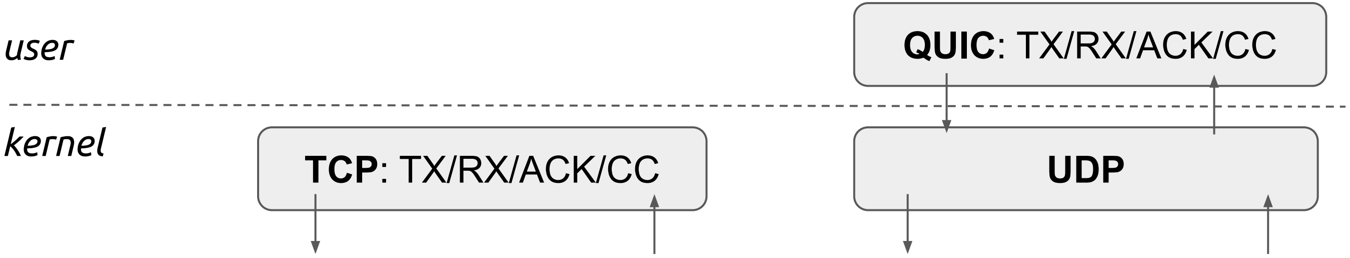
- **Avoids protocol/implementation entrenchment**
  - as implemented in user space and encrypting header

- **Reduces latency**
  - Fewer handshake (i. TCP->UDP; ii. by reusing the server's cipher info.)
  - No transport-level HOL (Head-Of-Line) blocking

- **Improves loss recovery**
  - distinguishes the ACK of a re-TX from that of an original TX

QUIC’s Problem: Higher CPU Consumption

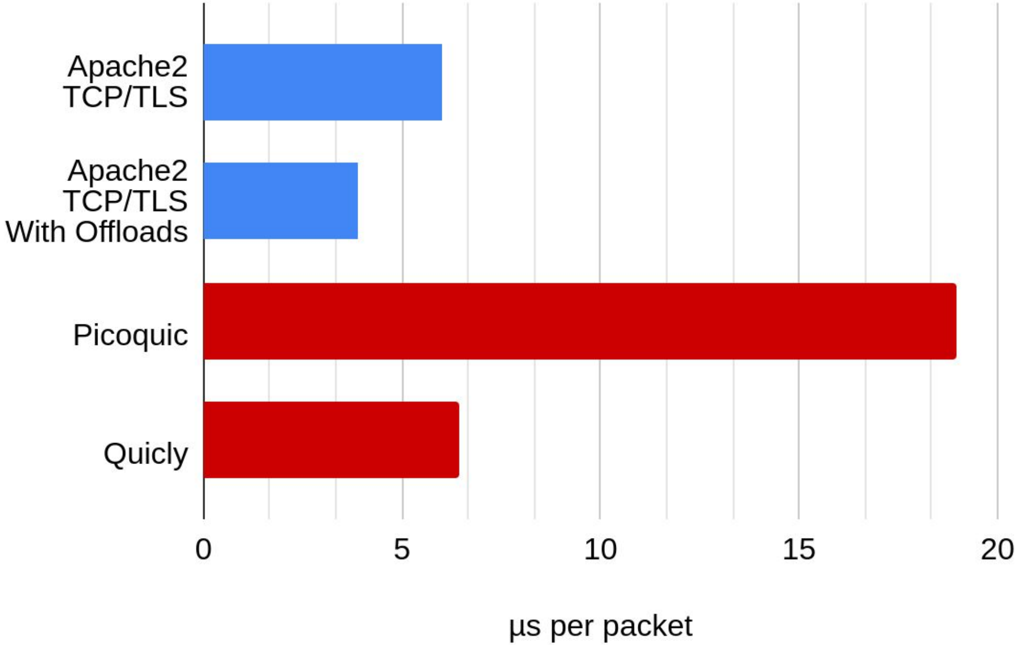
- QUIC is fully implemented **in user space** by design
  - It incurs additional context switches and data copy between user & kernel space
  - Its ACK, Congestion Control (CC), and re-TX take place **in user space**



- Google reported* **2.0x** as high CPU consumption as TCP/TLS stack
- On mobile devices, QUIC’s CC is application-limited for 58% of the time**

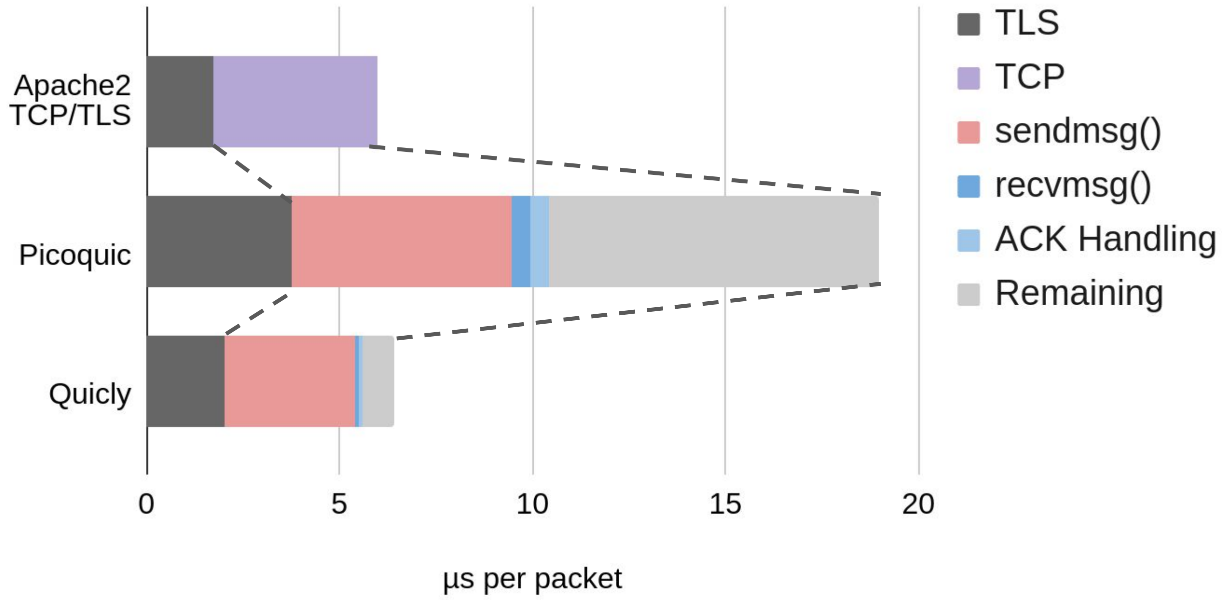
*Langley et al. "The quic transport protocol: Design and internet-scale deployment." SIGCOMM ‘17
**Kakhki et al. "Taking a long look at QUIC: an approach for rigorous evaluation of rapidly evolving transport protocols." IMC ‘17

CPU Consumption: TCP/TLS vs QUIC



* All are the server's CPU-bound / Repeated 2GB File Transfer

CPU Breakdown

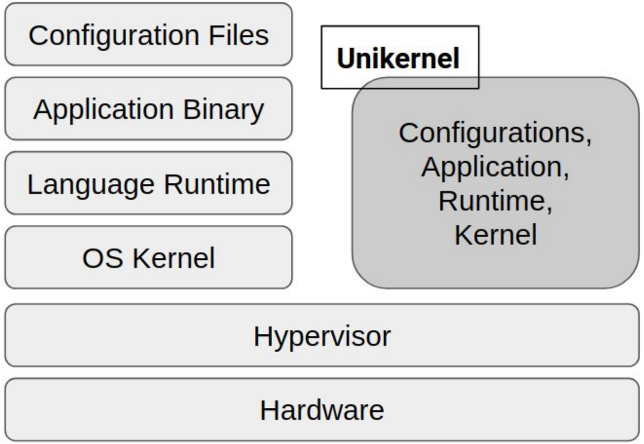


* All are the server's CPU-bound / Repeated 2GB File Transfer

Our Solution: Unikernel

- Highly-specialized, **single-address-space** kernel
  - In **sendmsg**()/**recvmsg**(), one data copy is saved
  - Context switches becomes quicker
  - System calls become function calls

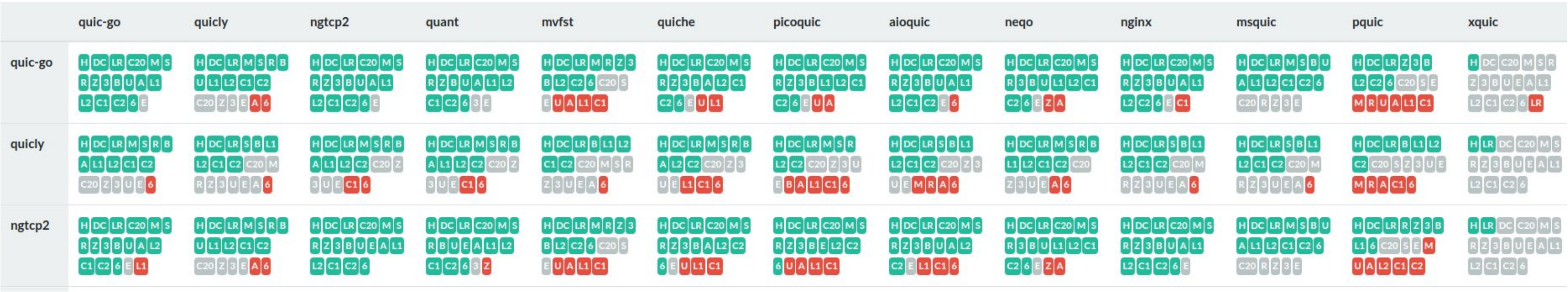
- Includes a minimal set of apps & libs
- Runs directly over hypervisor (or HW)
- Sealed against run-time modification



QUIC Implementations

- **23** are listed by the QUIC Working Group

- <u>Interoperability</u> has been a primary focus; **no performance work so far**



.
.
.

Selection Criteria for QUIC Impls.

From the list, we **pick** ones that

- are **open-sourced**
- provide **minimal testing interface**
- support **draft-29 (Jun 2020) or later**
- have **100+ GitHub stars**, if hosted by GitHub

And **rule out**

- **“Not performant”** (Chromium) or **“not for production”** (Kiwk)
- **Apache-based** ones (ATS); due to the difficulty in Unikernel-porting

Performance of QUIC Implementations

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

Performance of QUIC Implementations

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

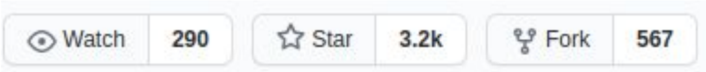
Shown[*] **robust** (Picoquic)

Performant (Quicly)

[*] Yang et al. "Making QUIC Quicker With NIC Offload." EPIQ '20

OSv Unikernel

- From Cloudius in 2014
- Designed for cloud VMs
- Seen wide community support →



- "Can run unmodified Linux executables (with some limitations)"

Porting Picoquic & Quicly into OSv

- **OSv implements only a subset of POSIX ("some limitations")**
  - Thus, we removed setsockopt()'s for ECN, MTU discovery, and IPv6

- All libraries and configuration files must be identified and incorporated
  - The build scripts of all dependencies are re-written

- Transport-layer offloads are off
- Ensure no disk read/write
- Enlarge the buffer size
- Enforce the same CPU/compiler flags and cipher suite

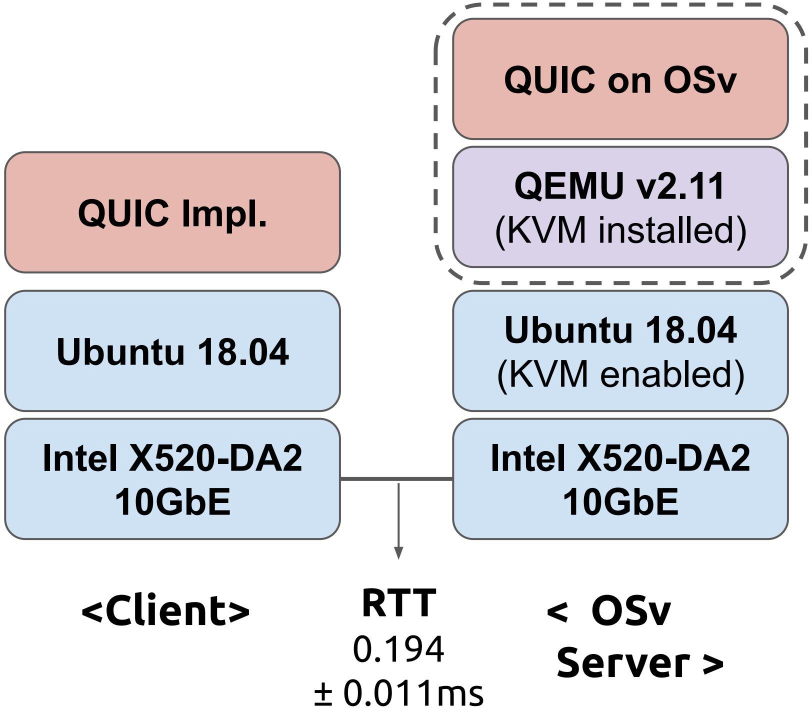
Experiment: Settings

QUIC Implementation

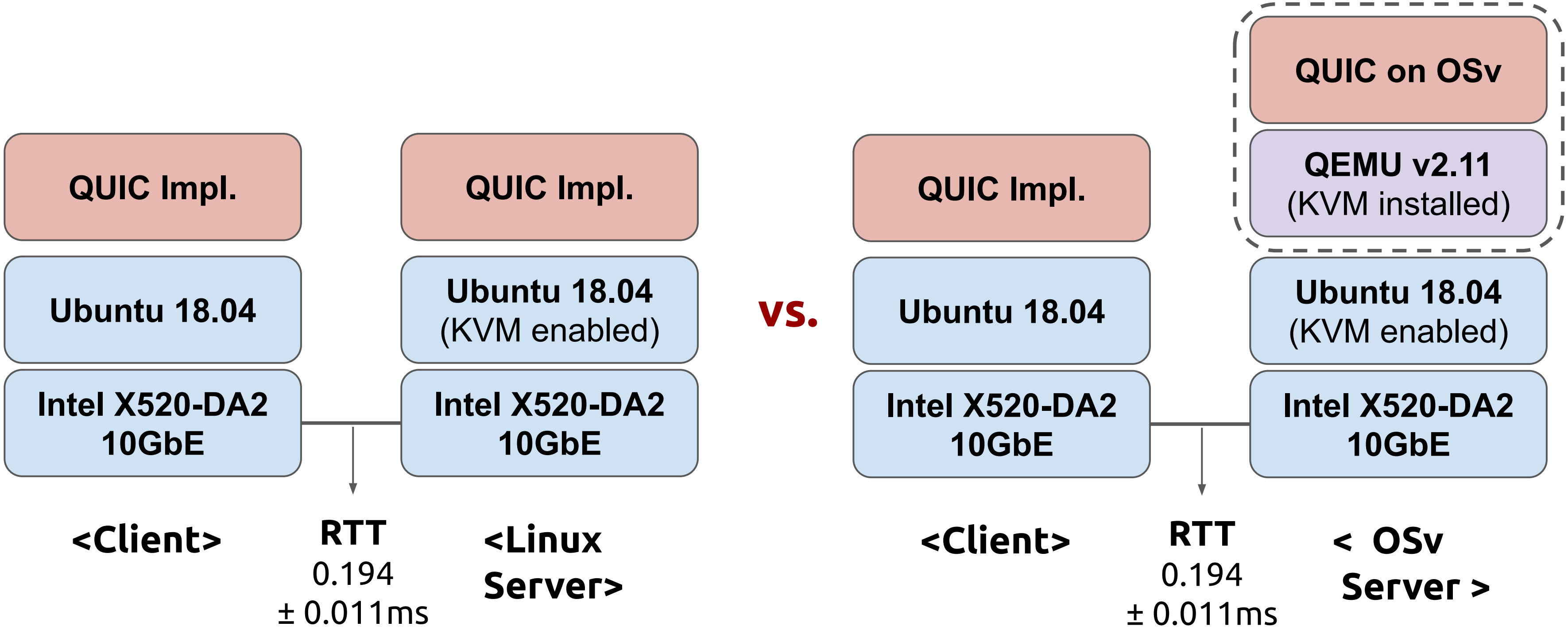
- Draft-29
- AES128-GCM-SHA256
- “-O2”

Machine

- Intel Xeon X5650@ 2.67GHz (24 core)
  - Used only one core unless otherwise specified
- DDR3 20GB (Cli.) / 24GB (Serv.)



Linux-Linux vs. Linux(Cli.)-OSv(Serv.)



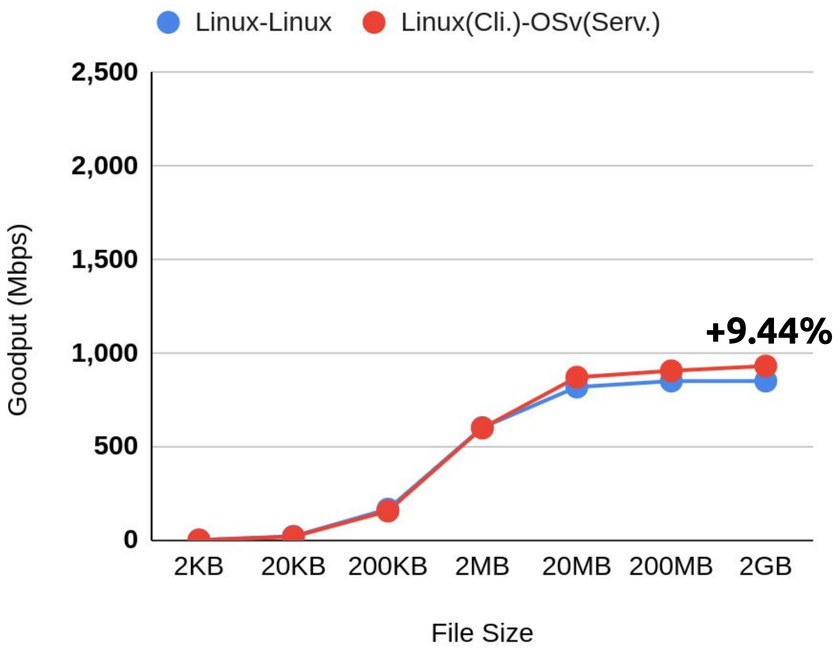
Experiment: Methodology

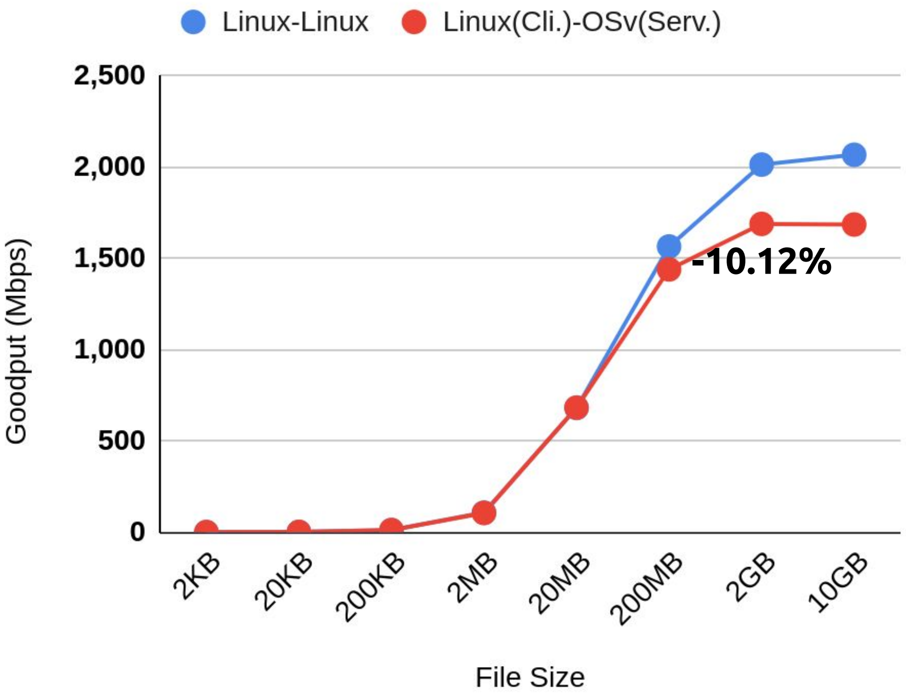
- **Goodput**
  - Requesting a single file of size from 2KB to 2GB
  - 30 runs, after 3 runs discarded for warm-up

- **Response Time**
  - 100,000 1-byte requests in total, originating from 144 clients

- **Request Per Time (RPS)**
  - 100,000 2KB file requests, from 144 clients

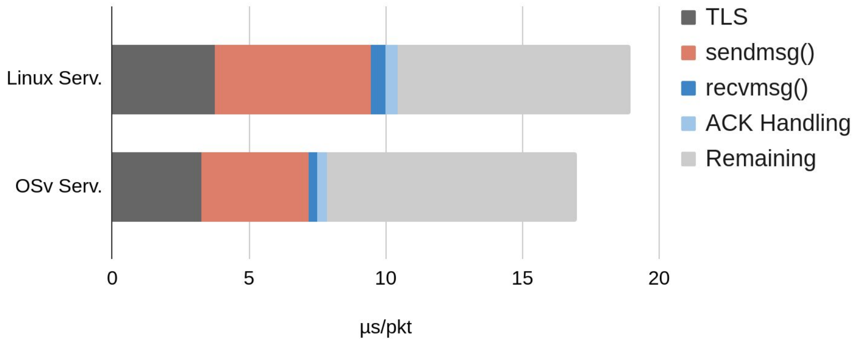
Goodput





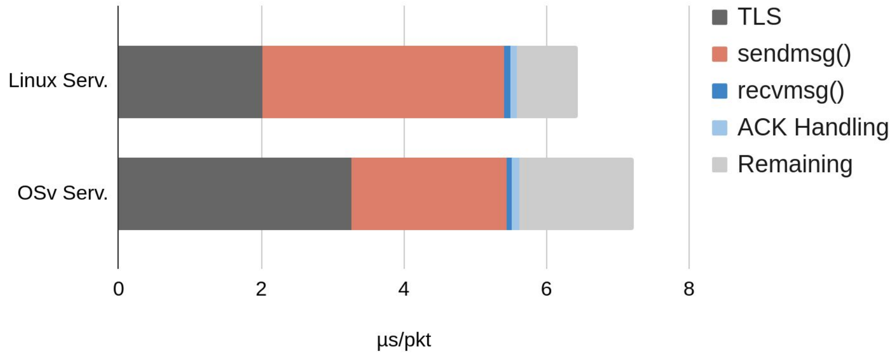
Picoquic **Quicly**

Goodput: CPU Breakdown



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%

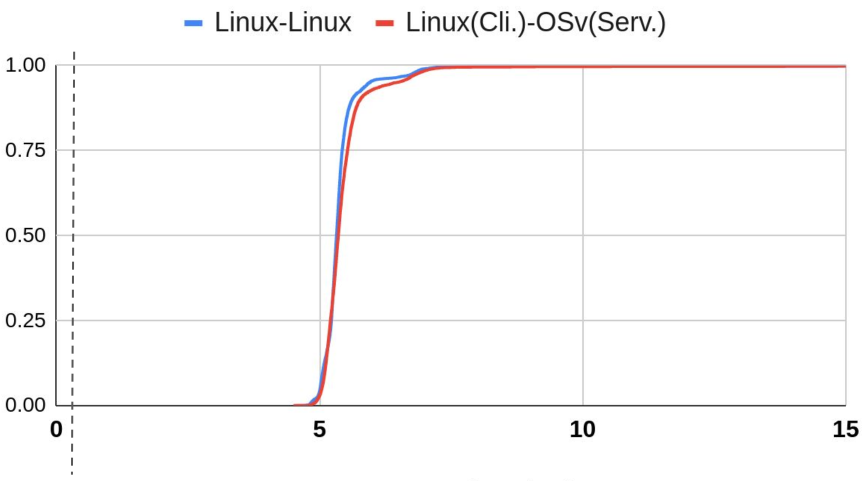
Picoquic



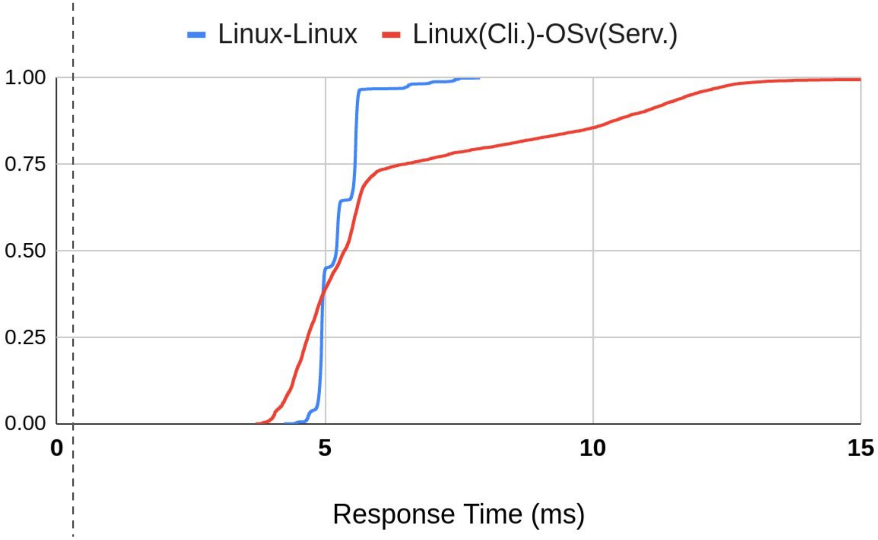
µs/pkt	Linux Serv.	OSv Serv.	Change
TLS	2.02	3.27	61.77%
sendmsg()	3.39	2.17	-35.94%
recvmsg()	0.09	0.07	-27.64%
ACK Handling	0.08	0.12	43.03%
Remaining	0.85	1.60	87.10%
Total	6.4	7.22	12.17%
Goodput (Mbps)	1,783.62	1,603.19	-10.12%

Quicly

Response Time (CDF)



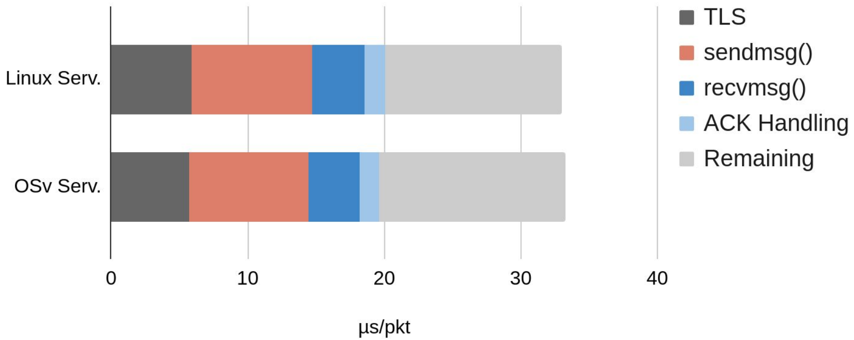
RTT
0.19 ms



RTT
0.19 ms

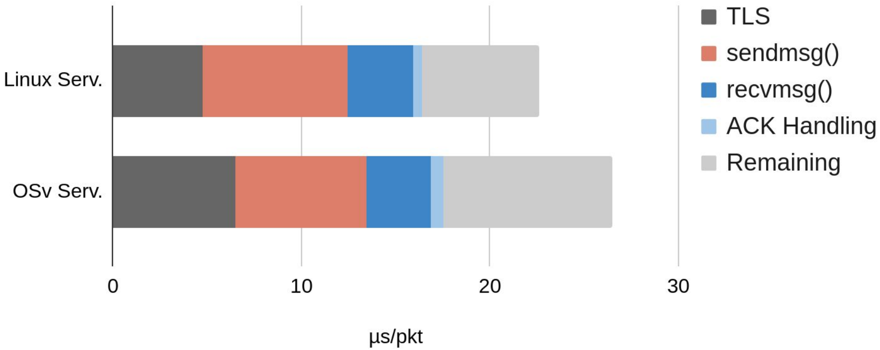
Picoquic **Quicly**

Response Time: CPU Breakdown



µs/pkt	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%

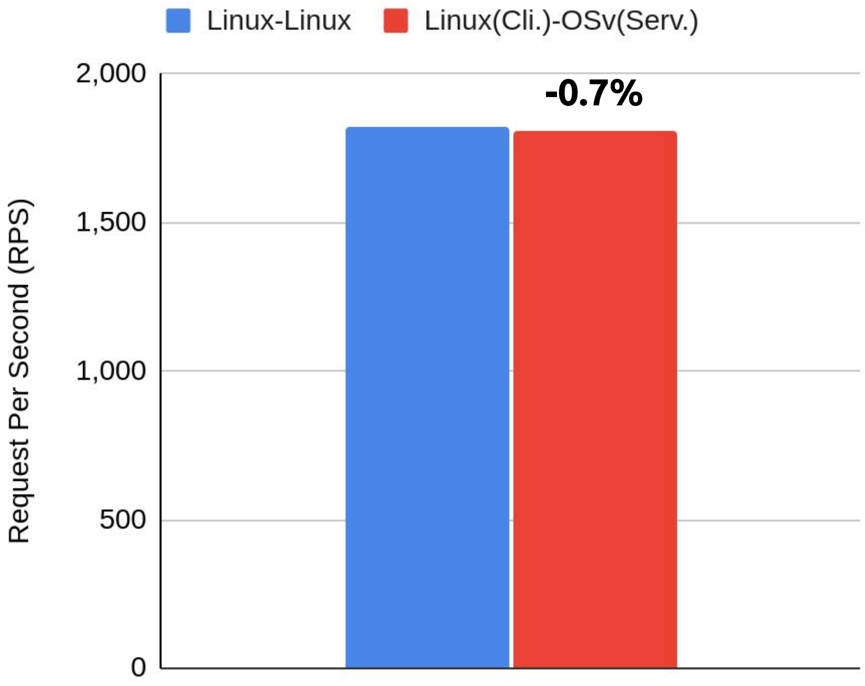
Picoquic



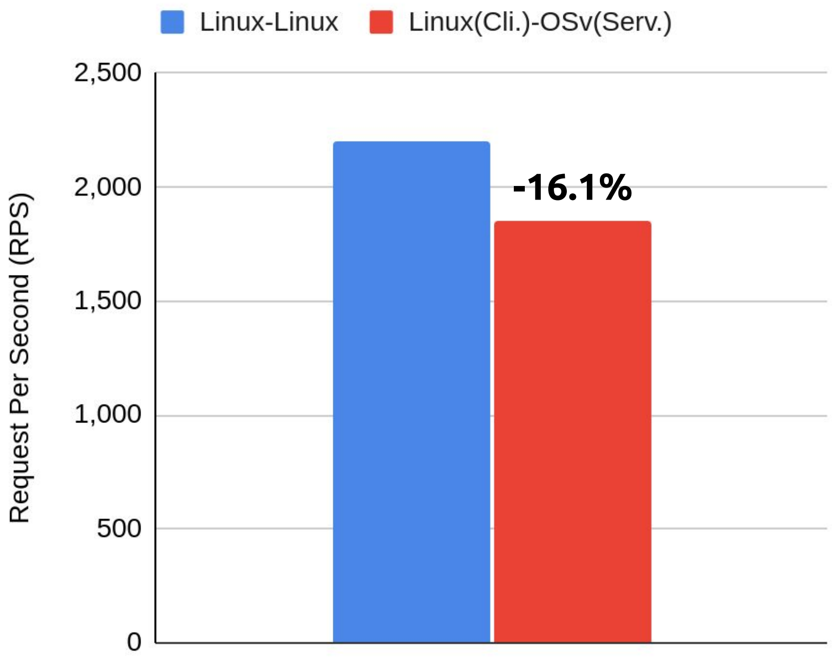
µs/pkt	Linux Serv.	OSv Serv.	Change
TLS	4.78	6.48	+35.55%
sendmsg()	7.65	7.00	-8.56%
recvmsg()	3.46	3.36	-2.84%
ACK Handling	0.48	0.69	+42.82%
Remaining	6.22	8.97	+44.14%
Total	22.60	26.50	+17.26%
Res. Time (ms)	5.318	6.55	23.17%

Quicly

Request Per Second (RPS)

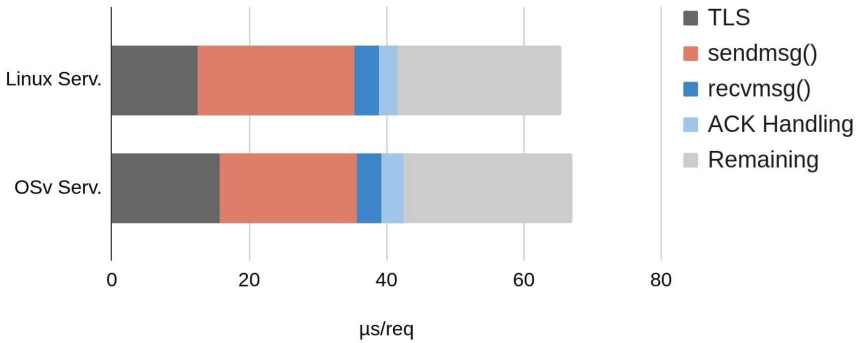


Picoquic

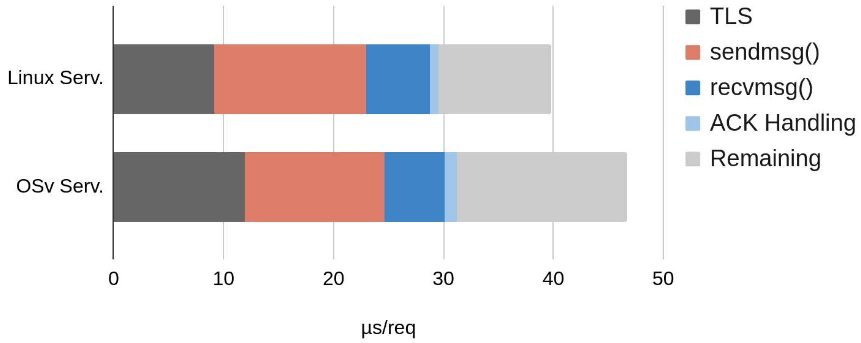


Quicly

RPS: CPU Breakdown



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%



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%

Picoquic

Quicly

Conclusion

- We **instrumented 6** QUIC **implementations and measured goodput**

- We **ported** two of them **into OSv Unikernel**
- In our experiment,

	Goodput	Response Time (avg)	RPS
Picoquic	**+9.43%**	**+0.92%**	**-0.7%**
Quicly	**-10.12%**	**+23.17%**	**-16.1%**

Conclusion (cont.)

- In all cases, **sendmsg() and recvmsg() were faster** in OSv
    - The gain was greater where the avg size of packets was larger
        - Res. Time (1B / 1pkt) < RPS (2KB / 2pkt) < 2GB File (avg pkt size ~ MTU)
        - This is probably because of the one data copy being saved

- It attributed to the **improvement of Picoquic's goodput**, while keeping the avg response time and RPS of Picoquic OSv at a similar level (+0.92%/-0.68% resp.)
- **Quicly OSv was significantly slower (30.6-87.1%) in the other sections** including TLS
    - This offset the gain from sendmsg() and recvmsg()
    - No overall improvement in all of the scenarios

- ACK Handling was faster only in the Picoquic's Goodput and Response Time scenario (25.5%/4.1%, resp.)