

École polytechnique de Louvain

Improving the performance of picoquic by bypassing the Linux Kernel with DPDK

Author: **Nikita TYUNYAYEV**

Supervisors: **Oliver BONAVENTURE, Tom BARBETTE**

Readers: **Oliver BONAVENTURE, Tom BARBETTE, Fabien DUCHÊNE**

Academic year 2021–2022

Master [120] in Computer Science

Acknowledgements

I would like to express my appreciation to my supervisor, Prof. Olivier Bonaventure, and Prof. Tom Barbette. Who devoted a part of their time to my thesis and provided guidance and feedback. I want to thank Maxime Piraux and Louis Navarre for participating in our weekly meetings.

I would also like to thank Fabien Duchêne for accepting to be a reader for this thesis.

Finally, I am grateful to my family and friends who helped me through this journey. I want to thank Chloé for always being there for me, Aurélien, Alice, Arnaud, and Jean-Paul for our study sessions and for the motivation they provided me to improve this thesis.

Abstract

QUIC is a recently standardized transport protocol built on top of UDP and designed to resolve shortcomings of the TCP protocol in the modern web. Several implementations exist, but unfortunately, they can not match the speed of the TCP+TLS stack. High-performance NICs becoming a commodity and CPU improving at a slower rate than those NICs worsen the problem because a high percentage of the CPU cycles must be dedicated to QUIC. One way to improve the performance of QUIC implementations is to avoid the Linux networking stack and process packets in user space. The DPDK library is a widely used solution for this purpose. In this master thesis, we integrated DPDK into picoquic, one of the many implementations of QUIC. We evaluated the resulting gain in different use cases, such as a classic client/server where we observed 3x greater goodput and latency reduced by 30%. In the case of a tunneling application, the goodput increased by a factor ranging from 3 to 5. The thesis shows profilings of the executions of the original and modified implementation to explain those gains. Finally, the thesis compares picoquic, msquic, a high-performing implementation of QUIC, and picoTLS, a TLS stack.

Contents

1	Introduction	6
2	State of the art	8
2.1	QUIC	8
2.1.1	History of QUIC	8
2.1.2	Handshake	8
2.1.3	Encryption	9
2.1.4	Multiplexing	10
2.1.5	Congestion control and Loss	11
2.1.6	Monotonically increasing packet numbers	12
2.1.7	Connection migration	12
2.1.8	Flexibility	13
2.1.9	Performance drawback	13
2.2	Tunneling	13
2.2.1	Reliable transfer	13
2.2.2	QUIC's unreliable datagram extension	14
2.2.3	The MASQUE protocol	14
2.2.4	Difficulties	14
2.2.5	Tunneling a UDP packet	15
2.2.6	iCloud Private Relay	15
2.2.7	Wireguard	16
2.3	The need for kernel bypass	16
2.3.1	Limitation of the Linux kernel	17
2.3.2	Kernel-bypass techniques	19
2.4	The DPDK library	19
2.4.1	Environment Abstraction Layer	19
2.4.2	Memory	19
2.4.3	Mbuf	20
2.4.4	User space processing	20
2.4.5	Packet processing model	20
2.5	Scaling a server	21

2.5.1	Receive Side Scaling	21
2.5.2	Limitation of RSS	22
3	picoquic	23
3.1	Main components	23
3.1.1	Developping an application	23
3.2	The case for DPDK	24
3.3	Using DPDK	24
3.3.1	Constructing and inspecting packets	25
3.4	How to integrate DPDK	25
3.4.1	Rx and Tx descriptors	26
3.4.2	Queues and memory pool	26
3.4.3	Main loop	26
3.5	General Segmentation Offload and batching	27
3.6	Limitations	28
4	Measurements	29
4.1	Defining performance	29
4.2	Experimental setup	29
4.3	Evaluating batching	30
4.4	Performance measures	31
4.4.1	Goodput comparison	31
4.4.2	Handshakes	32
4.4.3	Web requests	33
4.5	Finding the bottlenecks	34
4.5.1	picoquic	34
4.5.2	picoquic-dpdk	35
4.6	Cost of select	38
4.7	Interoperability	39
4.8	Cost of encryption	40
4.8.1	Comparing different encryption algorithms	40
4.9	Comparison with other stacks	41
4.9.1	A first comparison	42
4.9.2	picoquic vs msquic	42
4.9.3	A fairer comparison	43
4.10	Exploring multi-threading	44
4.10.1	Experimental setup	45
4.10.2	Workload	45
4.10.3	Analysis	46
4.10.4	Fairness	48
4.11	Conclusion	48

5	VPN prototype	49
5.1	The prototype	49
5.2	Measurements	51
5.2.1	Comparing UDP and TCP workloads	53
5.2.2	Comparison with Wireguard and hybrid relays	54
5.2.3	Other use cases	56
5.3	Conclusion	56
6	Conclusion and future work	57

Chapter 1

Introduction

QUIC [24] is a growing transport layer protocol developed on top of UDP, which offers several advantages over the traditional TCP+TLS stack. Namely, faster handshake, more flexibility, pluggable congestion controller, connection migration, and better integration with HTTP.

Another trend in the data-center world is the fast-increasing performance of Network Interface Cards (NICs). In the last decade, NIC's performance increased faster than CPU performance [69], and the trend for micro-service style applications pushed the need for lower response time.

The need for lower response time means more cores are required to handle incoming requests. It is thus necessary to have networking stacks that are (i) fast and (ii) that can scale well with the number of cores. With high-speed NICs and a high number of cores, the Linux networking stack can become a bottleneck [12].

Several implementations of the QUIC protocol exist, but even those that place performance at the core of their development, such as msquic [44] can not achieve goodput on par with TCP+TLS. In this master thesis, we study the reasons behind the poorer goodput of QUIC implementations, particularly the limitations linked to the I/O of the Linux kernel networking stack. We explore techniques to bypass the stack to improve the performance of a particular implementation: picoquic [57]. The DPDK [17] library is used to receive and transmit packets instead of the Linux Kernel UDP stack. It features Poll Mode drivers, efficient and NUMA-aware memory allocation, efficient communication mechanisms between cores through *rte_ring*, zero-copy, and the usage of HugePages to improve performance.

This thesis describes the steps that were necessary to integrate the library. The modified implementation is tested for three metrics, the maximum achievable goodput, the time it takes to complete a handshake, and the time it takes to complete a standard request to a website. Using three metrics instead of simply looking at the goodput gives us a broader view of the performance gains. We show that DPDK improves the performance of picoquic significantly. We improve the

goodput by a factor of 3 and the latency by 30% for a simulated web request. Compared to msquic a QUIC stacked optimized for performance that heavily outperforms picoquic. We show that our version provides 85% higher goodput in standard conditions and a 3x better goodput when the number of cores of both implementations is limited to one. This thesis provides clues on how msquic outperforms picoquic. Compared to picoTLS, a TCP+TLS stack, we show that when our implementation and picoTLS are limited to a single core, our version provides slightly better goodput, which is impressive considering the TCP+TLS stack is heavily optimized and outperforms all other known QUIC implementations.

As said before, QUIC has a richer feature set compared to TCP. One of those features is unreliable transport through the DATAGRAM frames [54]. The datagrams can be leveraged for proxying applications. We want minimum delay in some applications, such as VoIP (voice over IP). Thus, reliable transport with re-transmissions can increase the latency. Apple recently proposed iCloud private relays as a means to increase security and anonymity on the internet [4]. This system relies on QUIC, and HTTP3 [9]. Implementing a prototype of this relay is an excellent opportunity to study tunneling and apply DPDK and picoquic to another use case, different from a simple client/server application. We compare our system of relays to Wireguard, a state-of-the-art tunneling application. We show that our prototype outperforms Wireguard in terms of maximum goodput by a factor of 5 for packets with 100 bytes payloads and 3 for packets with 1200 bytes payloads. We also compare Wireguard to a version of the relays which uses the original version of picoquic and show that they provide similar performance. We identify the UDP stack as the limiting factor for performance for Wireguard in our setup.

To conclude, in this master thesis, we provided an implementation of the QUIC protocol with kernel bypass, which significantly outperforms other QUIC implementations and even a TLS+TCP stack in goodput performance. We analyzed the reasons behind this improvement with profiling tools. We prototyped a tunneling application using our implementation and showed that it significantly outperforms Wireguard in terms of goodput, especially when tunneling small packets.

Chapter 2

State of the art

2.1 QUIC

QUIC [24] is a recently developed transport protocol. It aims to remedy the inadequacies of TCP for the modern web. It aims to reduce handshake times, improve security, offer connection migration possibilities, more flexibility, and better integration with the HTTP protocol. The HyperText Transfer Protocol (HTTP) is the basis of the Internet. It is used to request all the objects on a web page. Older versions of this protocol run on top of other TCP. Through the years, HTTP has had to adapt to the evolving needs of the Internet. The third version of HTTP: HTTP/3, is in its final standardization phase. It aims to increase security, flexibility, and performance [9]. It replaces the TCP protocol with QUIC.

2.1.1 History of QUIC

With the ever-increasing number of internet users, the performance in terms of latency and speed of transport protocol has become an increasing concern. For these reasons, Google started to develop QUIC (Quick UDP Internet Connections) in 2012 [39]. It is designed on top of UDP to avoid issues with middleboxes on the internet since UDP is already an established protocol. Nowadays, QUIC is used by many internet applications such as Chrome, Youtube, or Instagram. In 2015 an internet draft was received by the IETF for the standardization of the QUIC protocol. A working group was formed in 2016. In May 2021, QUIC was standardized in RFC 9000 [24].

2.1.2 Handshake

Traditionally, two three-way handshakes are needed to establish a secured connection: the TCP handshake to establish the connection and a TLS handshake

to exchange cryptographic information and encrypt the communication. QUIC combines the two handshakes into one, which provides reduced latency. TLS1.3 + TCP adds one RTT to every handshake compared to QUIC. During the first connection from a client to a server, TCP+TLS1.3 needs two handshakes to establish a secured connection and start sending requests, whereas QUIC only needs one. When a previous connection to the server has already been established, TLS1.3 + TCP and QUIC can use cryptographic material exchanged in the last connection to speed up the process. QUIC can start sending requests straight away, whereas TCP + TLS1.3 still need to open the TCP connection and thus need one extra RTT.

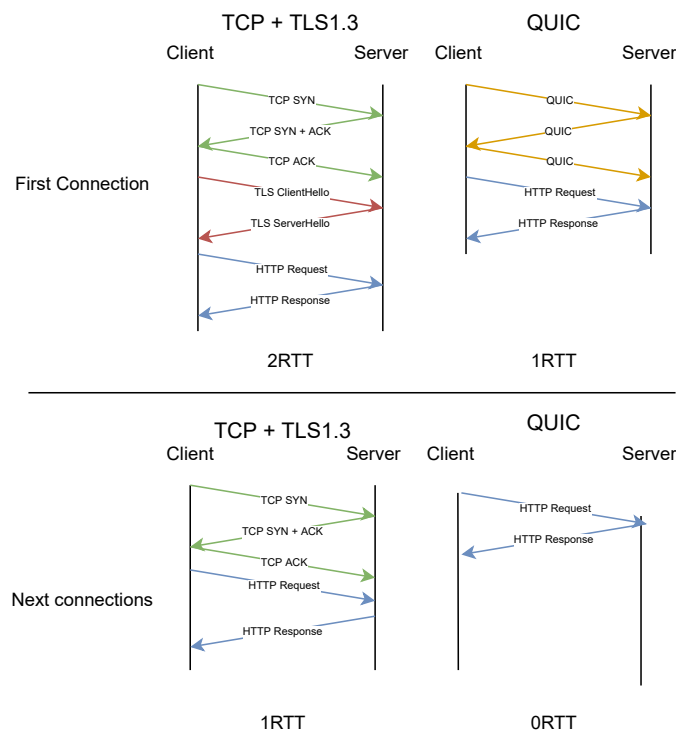


Figure 2.1: Handshake comparison between TCP + TLS1.3 and QUIC

2.1.3 Encryption

Encryption is essential on the modern internet to combat cyber attacks like phishing. Nowadays about 80-90% [19] of HTTP traffic is encrypted on the internet. QUIC packets are encrypted and authenticated, contrary to TCP, which relies on TLS for these tasks. A packet comprises a two-part header (one part is encrypted, and the other is not) and an encrypted payload. Header protection is applied after packet protection to mask specific header fields.

Figure 2.2 shows the protected fields [64] in a QUIC packet :

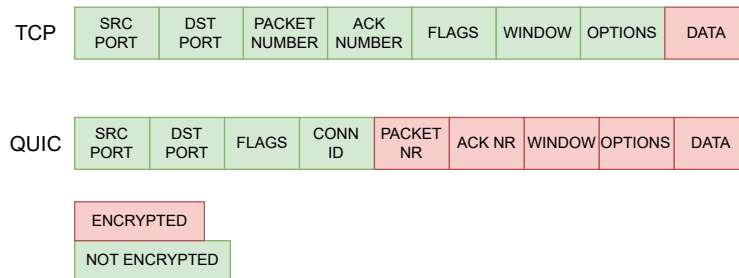


Figure 2.2: QUIC packet encryption

When using TCP + TLS only the data is protected, the middle-boxes have access to all the other fields such as the congestion windows. By reducing the value of this field, a middlebox can limit the transmission rate of a connection. This gives the network operators more tools to manage their networks.

2.1.4 Multiplexing

QUIC adds the notion of a stream at the transport layer. It is a lightweight, ordered byte-stream abstraction provided to an application. Streams can be either bidirectional or unidirectional and can be interleaved. QUIC packets are composed of a sequence of one or more frames. A frame is always contained inside a single packet and comprises a header and a payload. Streams are transported inside frames.

HTTP/2 introduced the notion of a stream. It enabled the transmission of multiple objects simultaneously in the same TCP connexion. Unfortunately, head-of-line blocking is still an issue when HTTP is used on top of TCP. TCP does not have a notion of stream, which means that if a packet is lost, all the HTTP streams of the connection are impacted, not only the ones from the lost packet.

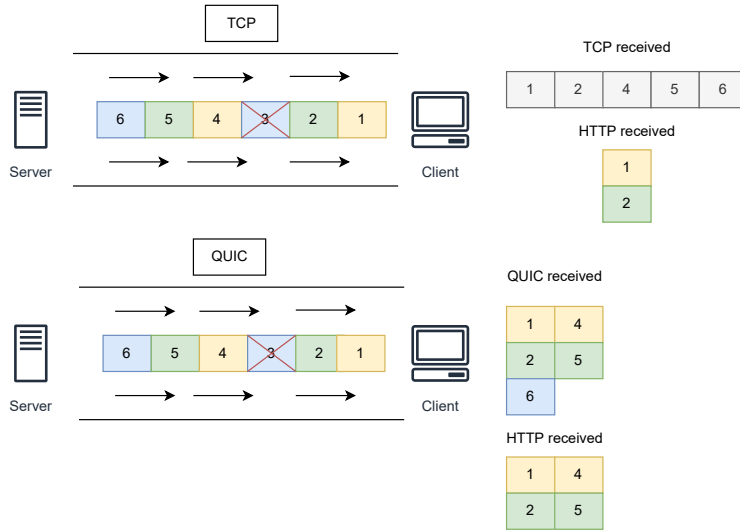


Figure 2.3: Head-of-line blocking

Figure 2.3 illustrates a server that multiplexes three HTTP streams inside the same connection (yellow stream, green stream, and blue stream). When packet number 3 is lost, since TCP only delivers bytes in order, the application will only receive packets 1 and 2. Even though TCP received packets 4, 5, and 6, they will not be available to the application, even if packet 3 carried only the blue stream, and the application could process packets five from the green stream and six from the yellow stream. With QUIC, it is no longer a problem since the notion of a stream is at the transport layer. The application can process packets 4 and 5.

2.1.5 Congestion control and Loss

Since QUIC moved to userspace, it facilitates the development and deployment of the congestion controller. During this thesis, we will mainly use the BBR [10] congestion controller. Instead of using packet losses to detect congestion like Cubic [22], BBR estimates the bandwidth and the RTT to compute the adequate delivery rate. To detect packet losses, QUIC uses the two metrics [23].

Packet Threshold

The first metric that QUIC uses is the `kPacketThreshold`. This threshold corresponds to the maximum number of packets that can be received with packet number $P > P_0$, where P_0 is the following packet number expected, without the packet P_0 being considered lost. This value should be of at least 3 to match TCP best practices. In networks with high reordering, increasing this value can lead to better performance since it can reduce spurious re-transmissions.

Time threshold

The `kTimeThreshold` metric corresponds to the amount of time that an ACK for a latter packet `P` can be received. While the ACK for an earlier packet `P0` is not still received before packet `P0` is considered lost.

2.1.6 Monotonically increasing packet numbers

All transmissions in QUIC are sent with a packet-level header that includes the encryption level and a sequence number. The encryption level is used to determine the packet number space. Contrary to TCP, packet numbers are sent in monotonically increasing order within a given number space, even for retransmitted packets. This solution avoids the need to differentiate between transmission and re-transmissions.

Contrary to TCP, QUIC decouples the delivery order from the transmission order. The packet number indicates the transmission order, whereas the stream offset field defines the delivery order in Stream frames. This differentiation solves the re-transmission ambiguity problem. In TCP, when a packet is retransmitted, and an ACK is received, the sender does not know whether or not the ACK is for the original packet or the retransmitted one. Since the packet numbers are monotonically increasing in QUIC, the original and the retransmitted ones will have different packet numbers.

2.1.7 Connection migration

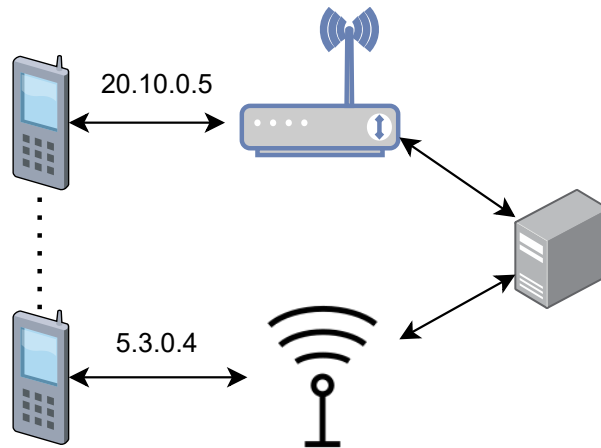


Figure 2.4: Example of connection migration

To identify a connection, QUIC uses a 64-bit connection identifier. This enables the client to change his IP address and port while maintaining the connection. Address changes can be caused, for instance, by Network address translation (NAT) or migration from Wi-fi to mobile data. The second case is a critical use case for mobile phones. Clients can move from network to network without needing to close and reopen a new connection like in TCP. This example is illustrated in Figure 2.4.

2.1.8 Flexibility

Traditional TCP is generally implemented at the kernel level. Modifying the protocol requires an update of the OS, which can take several months in some situations. QUIC, on the other hand, is implemented in user space, which makes the protocol's development and deployment easier. QUIC can be shipped directly with the application.

2.1.9 Performance drawback

Since QUIC is designed to be in user space, it has inherent performance drawbacks compared to TCP, which is generally designed in kernel space, but some implementation in user space [46] exists. The same code has the same performance, whether it runs in kernel space or user space. Still, user space applications are generally slower because they have to interact with the kernel to manipulate resources such as files through system calls which causes context switches.

Furthermore, Contrary to UDP, the TCP kernel stack is heavily optimized and performs well. The performance of UDP directly impacts the performance of QUIC.

2.2 Tunneling

The growing need for security requires the deployment of tunneling applications, and QUIC is an excellent transport layer protocol for developing tunneling applications.

2.2.1 Reliable transfer

QUIC provides multiplexing. Each user of the tunnel can have a dedicated STREAM, which makes congestion control easier than TCP based solution where the tunnel has to manage the congestion with one TCP connection per client.

2.2.2 QUIC's unreliable datagram extension

Applications that transmit real-time data, such as voice calls, don't need to transfer data reliably but instead want low latency. Historically, those applications used UDP in the transport layer and DTLS [59] to secure the connection. The datagram extension provides another solution to securely transmit unreliable data with the advantage of using the same cryptographic and authentication methods as traditional reliable streams. The QUIC datagram extension has the following characteristics :

- DATAGRAM frames contain a single message.
- DATAGRAM frames contrary to stream FRAMES don't have an identifier. It's impossible to have multiplexing with DATAGRAM frames on the same connection at the transport. Instead, the application has to manage it. One solution is to use a varint to store the multiplexing identifier.
- DATAGRAM frames are subject to congestion control. It makes it easier to manage the network loads than with classical UDP packets.
- DATAGRAM frames are acknowledged but never retransmitted when lost, which gives the sender feedback for the state of the network but doesn't impact latency.

2.2.3 The MASQUE protocol

MASQUE is an ongoing work to provide practical proxying features to QUIC developed on top of HTTP/3.

2.2.4 Difficulties

Since QUIC is built on top of UDP, using it to tunnel packets has some drawbacks. A DATAGRAM frame should be entirely contained inside a QUIC packet, limiting the size of packets that can be tunneled inside a QUIC connection. In some cases, such as when a client initiates a QUIC connection, it can send a packet of 1200 bytes which means that a tunnel with a smaller MTU cannot be used in this case.

2.2.5 Tunneling a UDP packet

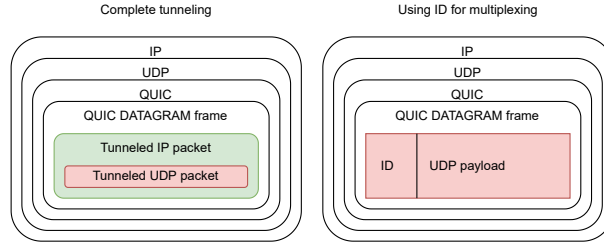


Figure 2.5: Anatomy of a tunneled UDP packet

Figure 2.5 shows two ways to tunnel UDP packets inside QUIC DATAGRAM frames. On the left diagram, the IP packet is tunneled inside a QUIC DATAGRAM. This approach is more straightforward and is the one we chose in this thesis. On the right, we have another way of tunneling the UDP packets [37]. We use an ID at the beginning of the DATAGRAM frame to differentiate between the different clients. It allows tunneling of the UDP payload alone and avoids tunneling the UDP/IP headers. The second approach is more complex because the application must reconstruct those headers when the packet exits the tunnel.

2.2.6 iCloud Private Relay

Apple introduced a new Internet privacy service called iCloud Private Relay [4]. The technology is built on the idea that no single party should have complete information on the user’s IP and the website he wants to access as shown in figure 2.6. To achieve this goal, Apple uses a system with two relays. One of the relays only knows the IP of the client, while the other only knows the SNI of the website that the server wants to use. Furthermore, each relay is operated by a different entity.

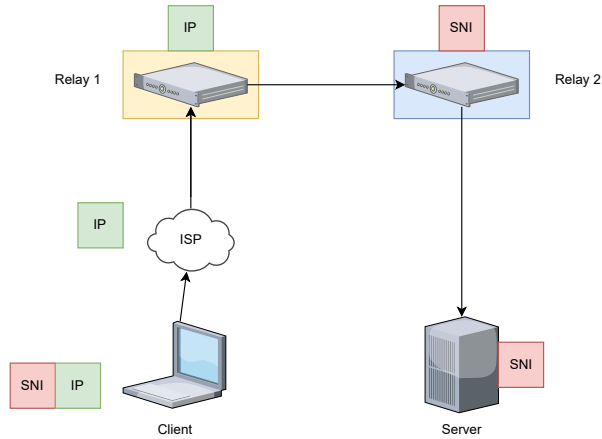


Figure 2.6: Apple iCloud relays

2.2.7 Wireguard

Wireguard [15] is a state-of-the-art VPN solution implemented at the kernel level. It uses a lightweight packet format with a sequence number and a sliding window receiver-side to manage packet delays, and reorderings [16]. It relies on ChaCha20-Poly1305 for symmetric encryption and outperforms OpenVPN [49] and IPsec [30] [32] in both latency and throughput. It aims to replace both of them. Wireguard relies on Diffie-Hellman keys to establish peerings and uses Cryptokey Routing. Cryptokey Routing works by associating public keys with a list of tunnel IP addresses allowed inside the tunnel. The Wireguard interfaces have a private key and a list of peers. Each peer has a public key. Those keys are used by peers to authenticate each other.

2.3 The need for kernel bypass

In the past decade, NIC's performance increased faster than CPU performance [69], 100Gbps NICs are becoming a commodity. The ever-increasing need for low-latency and high throughput in modern data centers and high fan-out workloads pushed kernel-bypass networking (KBN) to become the norm.

In the modern web, a request from a client to a server can trigger hundreds or thousands of requests towards microservices [50]. The latency of the request is dictated by the response time of the slower microservices. Minimizing the tail-latency thus becomes essential.

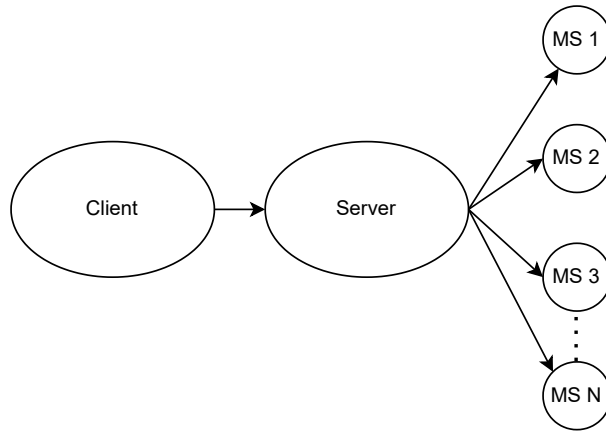


Figure 2.7: High fan-out request

2.3.1 Limitation of the Linux kernel

This sections details the performance limitation of the kernel networking stack.

Receiving a packet

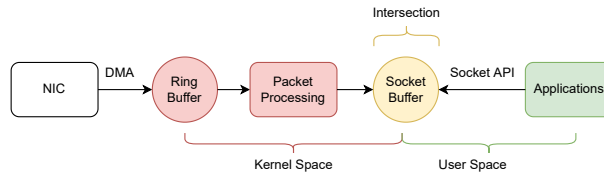


Figure 2.8: receiving a packet

The path that a packet takes through the kernel has a lot of steps. This is a significant source of inefficiency. We can divide the path of a packet inside the network stack into six steps.

1. When the NIC receives a packet, it uses direct memory access (DMA) to transfer the packet into main memory without the intervention of the CPU. The packet is stored inside a ring buffer.
2. Once the packet is stored in main memory, the NIC sends an interrupt to the CPU, which can now process the packet.
3. The kernel wraps the packet inside a `sk_buff` structure [67] to store the metadata required for processing. This structure is typical for every queue

and buffer in the network stack. It contains all the necessary information to process the packet. However, it is a large structure, and most of its members are unused most of the time, which can be a cause of inefficiency. [12].

4. Depending on the protocol of the packet, different functions will be called. If it is an IP packet, then `ip_rcv` is called. This function performs a series of basic checks. It looks at the address and drops packets not addressed to the host. This can happen if the NIC is in promiscuous mode and accepts every packet. It then verifies header values such as the length and the checksum. The packet is then given to the protocol stack for further processing.
5. When the protocol stack processing is complete, the packet is stored inside a buffer which is accessible through the POSIX API.
6. Finally, the application uses the socket API to receive the packet and can use it.

We can find five distinct sources of inefficiency in the Linux kernel network stack.

- The system calls: applications use sockets to receive packets. This generate system calls and thus switches between user space and kernel space. This context switch can cause performance issues linked to memory, such as cache pollution or TLB flushes which can happen on context switch when KPTI [47] is enabled to prevent the meltdown exploit. KPTI works by separating the user space and the kernel space pages [41].
- Data copies: as we saw in figure 4.2, a packet is copied from kernel space to user space through its path. This copy has a performance cost.
- Packet processing: the processing of a packet causes an interrupt and memory allocation of a heavy data structure: the `sk_buff`. When dealing with high throughputs, this can be a bottleneck.
- File descriptor: In a POSIX-based system, the applications use a shared file descriptor. When a new socket is created, the kernel must hold a concurrent lock to allocate a socket file descriptor. This can be a bottleneck when a server needs to handle millions of connections.
- Packet processing: the Linux network stack is inefficient. It uses big memory allocation and de-allocation as well as NUMA-unaware memory accesses. As we saw before, the usage of large structures such as `sk_buff` can also be a source of inefficiency.

2.3.2 Kernel-bypass techniques

There are two families of solutions to bypass the Linux networking stack [70]. Software-based solutions such as DPDK, where the application bypasses the Linux kernel, and the packet processing is done in user space and hardware-based solutions such as RDMA [28]. Where data plane tasks are offloaded to specialized NICs with computing cores, the hardware solution offers better performance, but less control on scheduling and networking sharing policies [70].

Some researchers propose to use data plane operating systems developed for high I/O performance, such as ZygOS [56] which is a data plane operating system designed to offer low latency.

2.4 The DPDK library

DPDK is a software framework that helps to develop efficient implementations of network functions by giving access to the NIC at the user space level. To improve performance, each DPDK process is pinned to a CPU core. Instead of using an interrupt based system for packet processing, DPDK uses polling to check that a new packet has arrived. This solution removes the need for a kernel and its limitation.

The library provides tools such as shared efficient memory pools that can be shared between CPU cores. For performance reasons, DPDK uses Hugepages to access continuous chunks of memory. Using Hugepages reduces the number of required memory pages and thus reduces the cache misses in the TLB. Furthermore sequential memory accesses are more efficient because accessing a particular address caches the nearby addresses too.

2.4.1 Environment Abstraction Layer

The Environment Abstraction Layer (EAL) offers an abstraction for low-level resources such as hardware and memory. It performs physical memory allocation with the *mmap* function on hugepages.

2.4.2 Memory

For its memory management, DPDK uses a shared memory pool. It preallocates pools of objects such as queues and buffers, which means that creating and deleting buffers one by one is not very costly. The Queues used are lockless. They enable different processes to access the queues concurrently and efficiently.

2.4.3 Mbuf

The Mbuf library enables the allocation and deallocation of packet buffers called mbufs. A *mbuf* contains both the data and the metadata of a packet. Thus only one call to allocate or free is necessary for the entire memory representation of the packet. This structure is the equivalent of the *sk_buff* in the Linux kernel that we described earlier.

The metadata in the mbuf can be the VLAN, the RSS hash, or its input port (the NIC from which it comes).

2.4.4 User space processing

DPDK enables user space I/O and packet processing. This avoids the overheads generated by context switching, data copying between the kernel space and the user space, the interrupt handler, and the Linux scheduler.

2.4.5 Packet processing model

Run to completion



Figure 2.9: Run to completion model

DPDK supports a run-to-completion model, meaning the packets are received and processed on the same logical core. The model doesn't offer a scheduler; packets are received through a polling mechanism.

Pipeline model

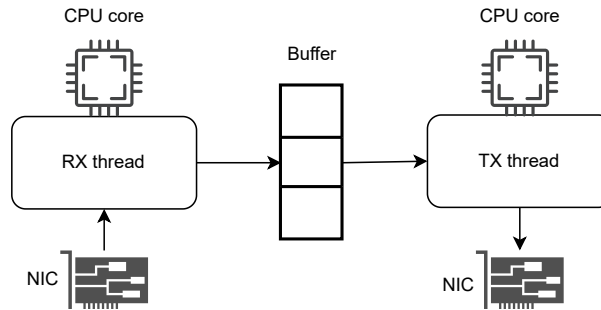


Figure 2.10: Pipeline model

DPDK also offers a pipeline model, where packets are received on a particular core but processed on another. Packets are transmitted between cores using rings. This approach can, in some situations, offer better performance where computations can be performed in stages.

2.5 Scaling a server

Web servers use multiple cores to meet the high demand and provide good latency. Scheduling requests across multiple cores is challenging, given the servers' growing number of cores and bandwidth. A good scheduling policy should meet the following constraints :

- load should be balanced between the cores,
- flow to core affinity should be respected,
- should require a low amount of per-flow state transfers.

2.5.1 Receive Side Scaling

RSS is a technology that enables the load-balancing of packets between multiple CPU cores. When using RSS, a packet follows this sequence: the packet header is hashed. This hash gives an index inside the indirection table. Each entry in this table then maps to a queue. Finally, CPU cores read from one or more queues.

A popular hash function used in a wide range of NICs to calculate the RSS hash is the Toeplitz Hash [36]. The DPDK library uses this hash function [65]. RSS is implemented in hardware and thus scales well with the number of cores.

2.5.2 Limitation of RSS

RSS provides a good load-balancing when the flows are well distributed among the queues and are comparable in workload. Those two conditions are not necessarily met even with a high number of flows. Without them, unbalanced loads between CPU cores arise.

Solutions like RSS++ [7] exist to rebalance the load between the CPU cores by looking at CPU utilization. Other solutions like Sprayer [61] calculate the hash on the checksum instead of the 5-tuple. Since the checksums have higher entropy than the five tuples, the load is better distributed across all the CPU cores. Unfortunately, this solution breaks the flow affinity between the cores and the flows: a single flow can be distributed between multiple queues and thus possibly different cores. This can be acceptable in some situations, but it is not suitable for stream processing because it can cause packet reordering and break core-to-flow affinity. One flow can be served by multiple cores, which means that any stateful processing can lead to contentions between CPU cores.

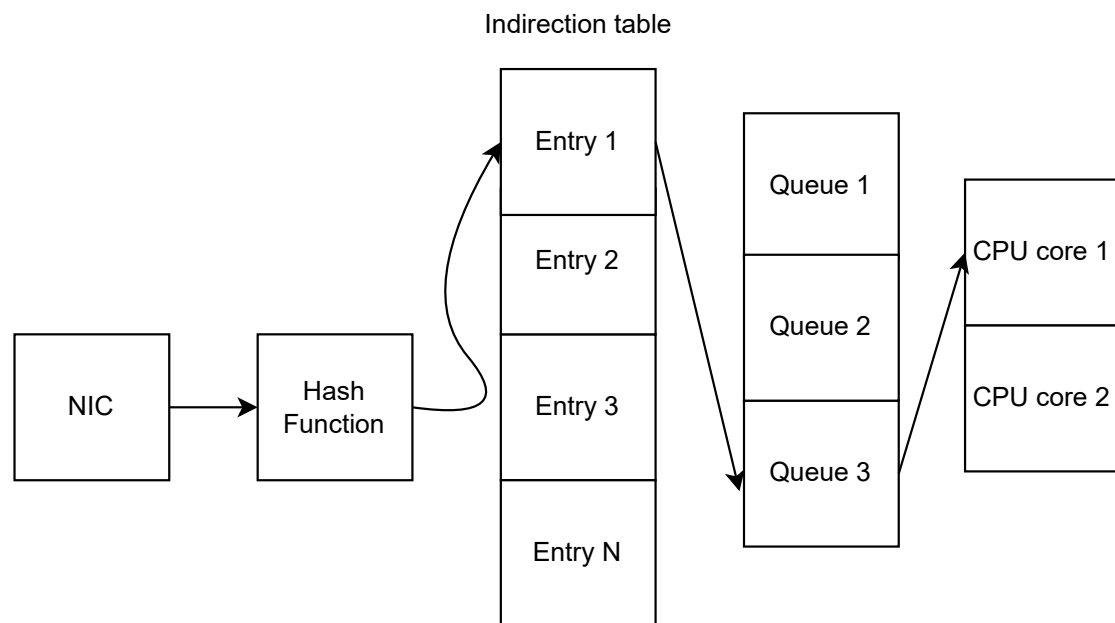


Figure 2.11: Path of a flow when using RSS

Chapter 3

picoquic

picoquic [57] is a minimalist open-source implementation of the QUIC protocol which supports QUIC v1 [24]. It is fully functional, but the performance for this implementation has not been optimized yet.

3.1 Main components

In this section we will highlight the main components of the picoquic library.

3.1.1 Developping an application

The packet loop

The packet loop manages the reception and transmission of the packets. It is implemented in *sockloop.c*. This loop receives packets and feeds them to the applications. Once no more packets can be received, the loop requests data to the application and sends it. The developer usually does not have to modify this loop to build new applications. In our case, to incorporate DPDK, we had to modify this loop heavily.

The application callback

When the main loop requests data to the application or feeds it, it does it through a function call to a callback. A simple server that stores files and responds to clients' requests by sending them the files could have a callback responsible for: opening a new stream when the server receives a request, opening the file when needed, marking the stream as ready, and then reading from the file when the packet loop requests data. Finally, it will close the file descriptors when the requests

are completed. Developing a picoquic application is done through this loop; the application's logic is managed here.

The application loop

This is the second loop that a developer has to design. This loop is optional. If the developer provides such a loop, it is called when the application is ready to receive or send packets, after receiving and after sending a packet. This loop is used, for instance, when a client application chooses to terminate when a request is completed.

The structures

picoquic uses the *st_picoquic_quic_t* structure to manage the state of a connection. It stores cryptographic information, information about the streams, and others. Another interesting structure is *picoquic_stream_data_node_t*. Every data segment received is stored inside such a structure. Those nodes are stored inside a splay tree, a binary tree with the added property that recently accessed nodes are fast to access.

3.2 The case for DPDK

Other frameworks enabling kernel-bypass for high-speed packet I/O applications, such as netmap [60] exist. We choose DPDK because of its richer library, which offers features above I/O management. It provides, among other things, NUMA-aware structures and tools for inter-core communication. Furthermore, nowadays, DPDK is widely used in the scientific literature when fast network I/O is required.

3.3 Using DPDK

Modifying an existing program to include DPDK is not always a straightforward process. It is not as simple as changing POSIX socket function calls by the DPDK ones. The Linux kernel networking stack provides a pleasant abstraction for the developers. The developer can initialize a socket with the desired protocol and addresses to send and receive data. The networking stack constructs the packets for the user. With DPDK, it is not the case.

3.3.1 Constructing and inspecting packets

When using DPDK, we are directly dealing with Ethernet frames when sending or receiving packets. Meaning we have to construct the entire packet before sending it and we have to analyze every received packet to make sure it is destined to our DPDK application.

Constructing a packet

When we want to send a QUIC packet to a specific address, we need to construct the whole packet manually. We must fill the MAC source and destination fields and set the EtherType field to "IP" at the Ethernet layer. To know which MAC destination to use for which IP, the kernel relies on the ARP tables. In our implementation, when a packet is received, the IP source/MAC source pair is stored inside an array. It is not the most efficient solution, but the performance is good enough when dealing with a small number of connections. A hash table would be more appropriate if we want to scale to a large number of connections. We have to specify the source and destination addresses at the IP layer and set the Protocol field to "UDP". We must also calculate the IP header checksum and set the length field. Finally, we need to set the source port and destination, the datagram length, and the checksum at the UDP level.

Inspecting a packet

Since we are receiving Ethernet frames, we receive every frame destined to the mac address of our NIC, which means that we may receive packets from unexpected protocols or a wrong IP address. We must manually inspect each packet to ensure it is a suitable protocol. We start with the Ethernet frame and verify that the EtherType field is either IPv4 or IPv6. Then, we have to look inside the IP packet to verify that the Protocol field is set to UDP. Finally, we can look inside the UDP packet and retrieve the payload.

3.4 How to integrate DPDK

This section describes the modifications made to the standard version (picoquic) to bypass the Linux kernel with the DPDK library (picoquic-dpdk). The library enables us to take control of a NIC in the user space. We have to configure it before using it.

3.4.1 Rx and Tx descriptors

When configuring a NIC with DPDK, we have to configure the number of *rx_descriptors* and *tx_descriptors*. They are the mechanism used to communicate with the NIC. They contain metadata about the packets. When the NIC receives packets, it uses DMA to write them in memory without interrupting the CPU. *rx_descriptors* contain pointers to those packets, whereas *tx_descriptors* contain pointers to packets constructed by the application and waiting to be sent. The NIC relies on those descriptors to know where the packets are located in memory. We have set the number of *rx_descriptors* and *tx_descriptors* to 1024, which is the default DPDK value.

3.4.2 Queues and memory pool

The descriptors are stored inside *rx_queues* and *tx_queues*. Those queues are NUMA aware. When configuring them, we have to specify the CPU core that is going to use them. Depending on the usage, we can configure the queues in different ways. For a simple client, we set one *rx_queue* and one *tx_queue*. We can specify multiple *rx_queues* and use RSS to balance traffic between those queues for a server.

To allocated *mbufs* when sending or receiving packets DPDK uses NUMA aware memory pools : *rte_mempools*. We have set them to the default DPDK value for the client. In the case of the server, we realized that sometimes our program would run out of memory when using multiple cores. Thus we adapted the size of the memory pool. We use the default size of the memory pool multiplied by the number of cores used by the server.

To send packets, we use a *rte_eth_dev_tx_buffer*. It stores packets before they are flushed to the NIC in batches.

3.4.3 Main loop

We modified the logical loop. While receiving packets, calls to *rte_eth_rx_burst* replace calls to *recvmsg*. This function tries to acquire a burst of packets. If no packets are available, it immediately returns whether it managed to receive packets or not. For the transmission, calls to *sendmsg* are replaced by calls to *rte_eth_tx_buffer* and *rte_eth_tx_buffer_flush* when necessary. *rte_eth_tx_buffer* buffers a single packet for transmission and flushes the buffer when it reaches a configured threshold. *rte_eth_tx_buffer_flush* flushes the buffer.

Algorithm 1 Main loop logic

```
1: procedure LOOP
2:   while true do
3:     while true do
4:       packets  $\leftarrow$  getPackets()
5:       if packets == null then
6:         break
7:       for packet in packets do
8:         time  $\leftarrow$  getTime()
9:         processPacket(packet, time)
10:    while true do
11:      packet  $\leftarrow$  requestPacket()
12:      if packet == null then
13:        flushBuffer()
14:        break
15:      bufferPacket(packet)
```

Even though we receive packets in batches, we call the *getTime()* function each time we feed one of the packets to picoquic. This step is necessary to make the congestion control algorithm work properly. Without it, the congestion control algorithm perceives a very low delay and tries to increase the congestion windows too quickly, which creates lots of packet losses. Regarding batching, we try to buffer as much as possible until picoquic stops providing new packets because of the congestion controller. At which point, we flush the buffer.

3.5 General Segmentation Offload and batching

picoquic support General Segmentation Offload (GSO) [62]. This technology lets the application send large segments that will be fragmented at a lower level. This reduces the per-packet processing cost. In picoquic-dpdk, batching is used to lower the cost of per-packet processing. Receiving a batch of packets and then processing them is faster than doing it separately for each packet. The size of the batches has an impact on the performance. A high batching number may increase the throughput but will increase the delay. We chose 32 as the size of the batches. This value is drawn from the experiment in section 5.4.

3.6 Limitations

DPDK greatly increases performance but has also several drawbacks. Our modified implementation has some limitations compared to the original one:

- It uses a whole NIC to run only one application.
- When bypassing the Linux kernel, we lose all of its features: routing, security, address management, management of other protocols such as ICMP or ARP, and all the other kernel network features.
- DPDK requires specific hardware and drivers, which can be an issue.

Chapter 4

Measurements

In this chapter, we will measure and evaluate the performances of picoquic and picoquic-dpdk, profile their execution, compare them against an another QUIC implementation and a TLS+TCP stack. Finally we will integrate RSS in the server and measure the performance benefits.

4.1 Defining performance

The performance of a transport layer protocol can be evaluated through many angles. We decided to select several metrics for our evaluation :

- goodput achieved during a large download,
- time to complete a handshake, this can be indirectly measured by the time it takes to complete a small request of a few bytes,
- time to complete the download of a web page.

4.2 Experimental setup

The machines used are equipped with an Intel(R) Xeon(R) Silver 4314 CPU @ 2.40GHz and 32 GB of RAM running the Linux kernel version 5.13.0-23-generic. The NICs used are Mellanox Technologies MT27800 Family ConnectX-5 100Gbps.

During our measurements, when more NICs were required, we used the Single Root I/O virtualization (SR-IOV) [5] technology available on Mellanox cards that enable a physical NIC to present itself as multiple virtual devices. Those virtual NICs are linked to an internal switch responsible for routing frames to the virtual NICs. The overhead of the technology is low because it is implemented in hardware. For the large download scenario, to avoid any I/O bottlenecks, the server fills the

packets with the *memset* function and reads nothing from the disk. When the client receives data, it's not saved on disk. Instead, the data is discarded to again avoid possible bottlenecks. The client and the server use HTTP to communicate.

picoquic offers multiple congestion algorithms. Unless specified otherwise, BRR is used as the congestion algorithm since it is the one that provides the best performance for both implementations. For encryption, AES128 is used because it provides the best performance, OpenSSL version 1.1.1f is used. Finally, to avoid packet losses on picoquic, we had to increase the size of the UDP receive buffers to 100MB. The Linux kernel uses those buffers to store UDP packets before they are processed temporarily. If those buffers are full and new packets arrive, packets are dropped, and thus losses occur. Figure 4.1 shows a summary of the setup.

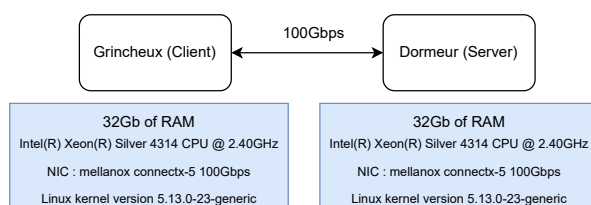


Figure 4.1: Experimental setup

4.3 Evaluating batching

In this section, we will motivate our choice for the batching size.

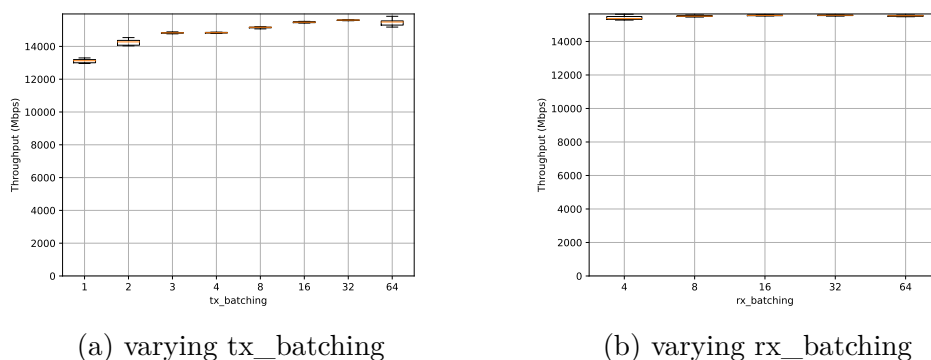


Figure 4.2: Goodput obtained with various rx_batching and tx_batching sizes

In figure 4.2, the y axis shows the achieved goodput, and the x-axis shows the different batching sizes used for transmission (*tx_batching*) or reception (*rx_batching*). For experiment (a), *rx_batching* is set to 64, and we vary the *tx_batching*. We

observe a jump from batches size of 1 to batches of size 2: some batching already makes a difference. The goodput increases until we reach a batching size of 16. In experiment (b), we fix the *tx_batching* to 64 and vary the *rx_batching*. Unfortunately, for limitations linked to the drivers, we can only set the *rx_batching* to multiple of 4 [40]. We thus have no points of comparison for batches of sizes 1-4. The experiment shows that the values set for *rx_batching* do not make a meaningful difference. For all further experiments, we set the value of *rx_batching* and *tx_batching* to 32 since it is a traditional value in other DPDK applications, and higher values do not make a difference, as shown in Figure 4.2.

4.4 Performance measures

In this sections we will evaluate the two protocols in the different scenarios that we presented in section 4.1.

4.4.1 Goodput comparison

The first performance metric that we present is the goodput. How much data can the server transfer to the client per second? In this scenario, a basic HTTP/3 application runs on the server. The client sends a GET request to the server to download objects of variable sizes. Both the client and the server use one core. The measurement is repeated 15 times.

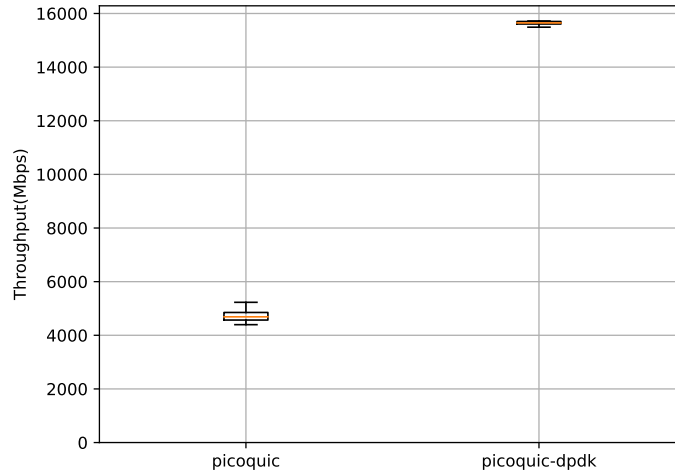


Figure 4.3: Performance comparison for a 20 GB download repeated 15 times

As expected, using DPDK increases the performance. Figure 4.3 shows that the throughput increases from 5 Gbps to 15.6 Gbps, a 3x increase. We can also observe that the variance on average is lower with picoquic-dpdk. This scenario is an excellent case for picoquic-dpdk. The transmission is long, so the congestion window has time to grow, and many packets are sent. Thus the cost of I/O is high. Since DPDK reduces the cost of I/O, we see a significant performance gain.

4.4.2 Handshakes

To measure the handshake performance, we looked at the time it takes to open a connection. We indirectly measured this metric by making small requests of 8 bytes from the client to the server, meaning that we measure the time it takes to open a connection, transmit 8 bytes and close it. We repeated the experiment 100 times to account for variability.

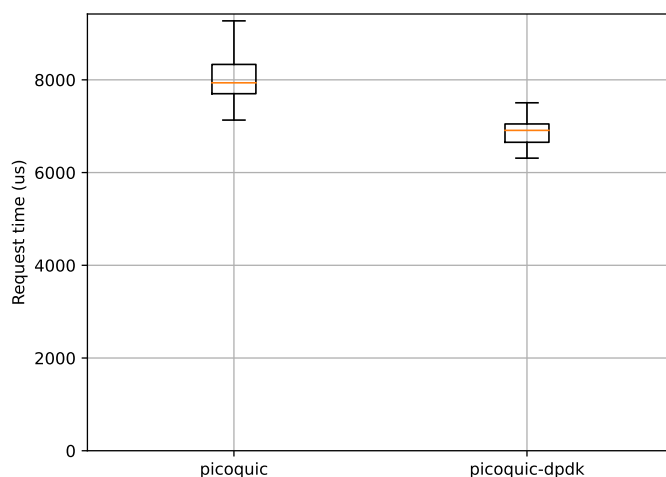


Figure 4.4: Performance comparison for a request of 8 bytes repeated 100 times

Figure 4.4 shows that the difference between picoquic and picoquic-dpdk is lower than before. The time to complete the request is lower for picoquic-dpdk. Furthermore, there seems to be less variance for picoquic-dpdk. The lower gains can be explained by the fact that fewer packets are sent and received, so fewer cycles are spent on those operations. We wanted to use perf to verify our hypothesis to see the differences. Unfortunately, there is a lot of noise for such small execution times, which makes understanding the bottlenecks difficult.

4.4.3 Web requests

In our third scenario, we will mimic a realistic web query from a single client to a single server. To determine the average size of a web page and the average number of HTTP requests to load a website, we used the HTTP Archive [52]. When selecting the period from 07/01/2022 to 07/01/22 (format (MM/DD/YY)) we see that the median number of HTTP requests on a web page is 75, and the median size of a web page is 2314.4 KB. Thus to mimic a web page, we will make 75 simultaneous requests of 30 KB from a single client to a single server to simulate the time it takes to download a web page.

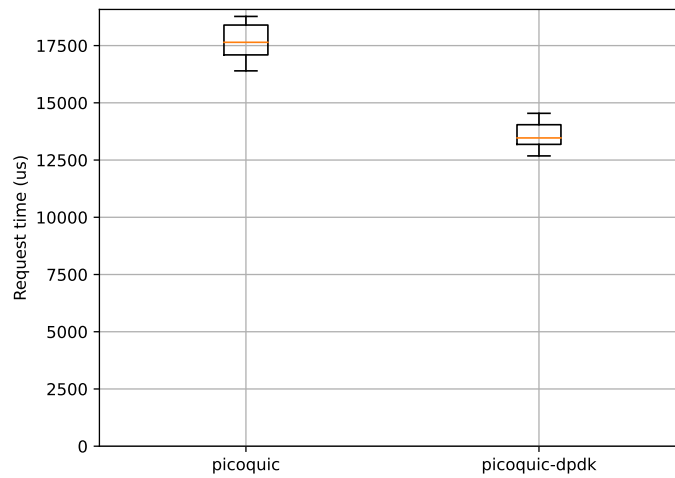


Figure 4.5: Performance comparison for a download of 2300 KB spread across 75 streams

Figure 4.5 shows performance gains of about 30% when using picoquic-dpdk compared to picoquic. This result is in-between our experiment with the goodput and the one with handshakes. We send fewer packets than in the handshake experiment but more than the goodput evaluation. This experiment tries to approximate a web request but is not perfect. During a genuine request, the server would need time to process the request and craft its response, and the client needs to process it. We expect to see gains lower than this 30% value in a real-world scenario.

4.5 Finding the bottlenecks

In this section, we will analyze what parts of the execution of the original implementation are the most costly and then see how the DPDK version improves on it. To analyze the execution, the perf [42] utility was used in conjunction with the hotspot [31] GUI. The DPDK workload is a download of 60GB. For the standard version, the workload is a download of 20GB. The workload is asymmetric to offer comparable execution time to both implementations. Each request takes around 30 seconds to execute. The workload is chosen to be big enough so that the maximum transmission rate is achieved in both cases and noise is reduced. We start by looking at picoquic and then picoquic-dpdk.

4.5.1 picoquic

In this section, we are going to profile an execution of picoquic from both the client and the server side and identify the hotspots and bottlenecks.

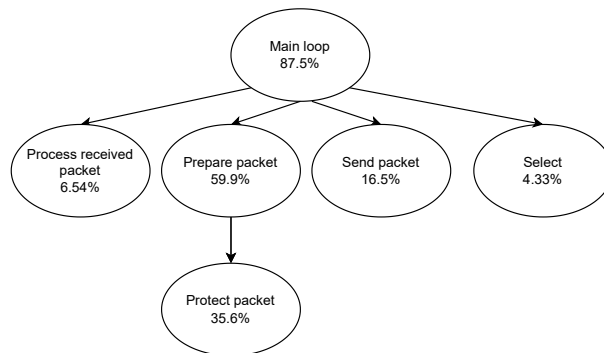


Figure 4.6: CPU cycle utilisation for the picoquic server

Table 4.1: Usage of CPU cycles for the server

task	% of CPU cycles
crypto	35.6
picoquic logic and processing	31.07
I/O	16.5
select	4.33

Figure 4.6 shows the main hotspots for the picoquic server. It is not a complete execution graph but shows the essential components. The server mostly sends packets and only receives ACKs from the client. Thus, the hotspots are all linked

to constructing and sending packets. We already observe that sending packets through the POSIX API is very costly (16.5%). We can classify the hotspots into 4 categories, the crypto, the picoquic stack processing, the I/O, and the *select* call.

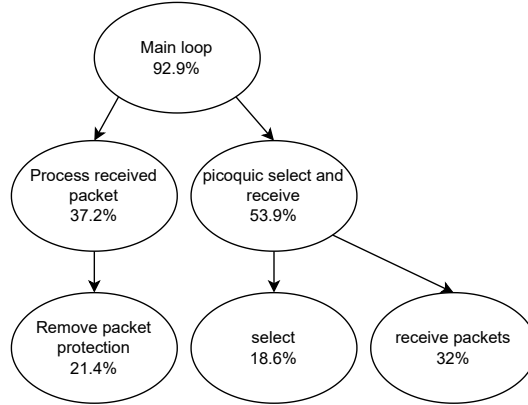


Figure 4.7: CPU cycle utilisation for the client

Table 4.2: Usage of CPU cycles for the client

task	% of CPU cycles
I/O	32
crypto	21.4
picoquic logic and processing	21
select	18.6

Figure 4.7 shows the main execution hotspots for the client. Since the client receives many packets and sends few, tasks linked to receiving and processing packets are costly. It is interesting to note that the picoquic stack only accounts for 21% of the CPU cycles. Compared to the server, the I/O for the client is very costly (32%). As we explained in section 2.3.1, receiving packets with the POSIX API is inefficient. Our profiling of the client confirms it.

4.5.2 picoquic-dpdk

In this section, we profile the execution of picoquic-dpdk from both the client and the server side and show the differences from picoquic.

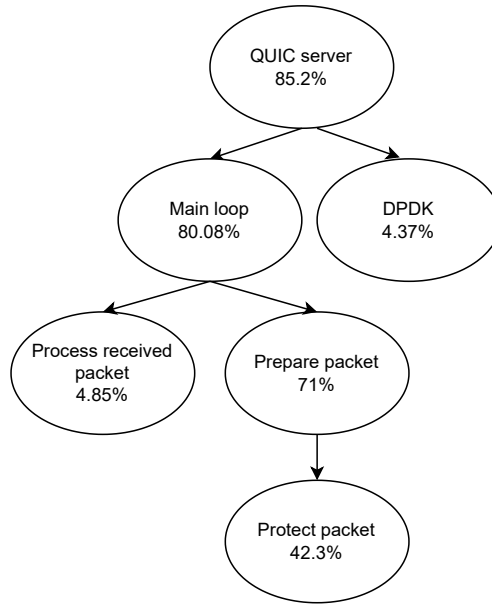


Figure 4.8: CPU cycle utilisation for the server

Table 4.3: Usage of CPU cycles for the server

task	% of CPU cycles
crypto	42.3
picoquic logic and processing	38.53
I/O + DPDK	4.85

Compared to the picoquic server, the cost of I/O is significantly reduced. The I/O + DPDK consumes 4.85% of the CPU cycles. This includes (i) allocating a packet, (ii) buffering it, and (iii) flushing the buffer when it is full. We also eliminated the cost of *select*. By reducing the cost of I/O, we observe that the picoquic stack accounts for a more significant part of the CPU cycles.

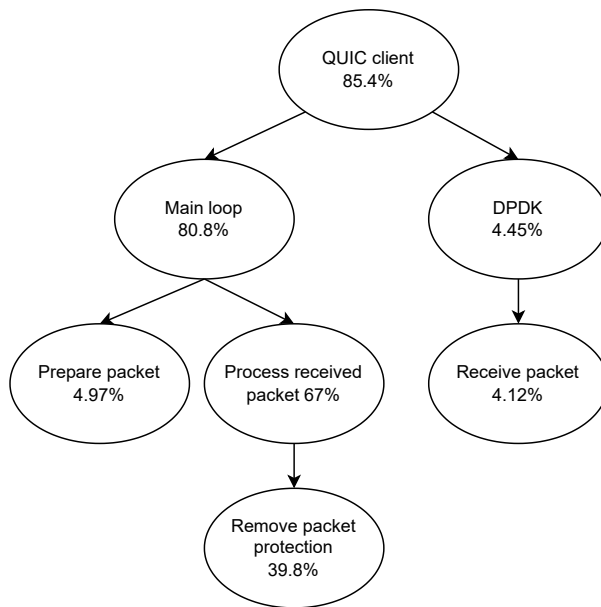


Figure 4.9: CPU cycle utilisation for the client

Table 4.4: Usage of CPU cycles for the client

task	% of CPU cycles
crypto	39.8
picoquic logic and processing	41.15
I/O + DPDK	4.45

Figure 4.9 shows the hotspots for the dpdk-picoquic client. Compared to the server, the DPDK part is much simpler. The main DPDK function call is *rte_eth_rx_burst*. It is used to receive batches of packets. Compared to its picoquic counterpart, we see that the utilization of the stack is much higher: the function responsible for constructing the packets appears, for instance. Crypto also takes a larger share of CPU utilization.

This comparison clearly shows the benefits of avoiding the Linux kernel and using DPDK. Sending packets takes 16.5% of the CPU cycles for the standard version. In contrast, for picoquic-dpdk, the path to send a packet takes about 4.85% when looking at the whole cycle shown in figure 4.10 of allocating the packet, buffering the packet, and flushing the buffer when full. On the client-side, receiving packets take almost 50% of the CPU cycles when adding up the cycles of *select* for picoquic, whereas picoquic-dpdk only uses 4.45% of the CPU cycles. The high CPU cost of *select* should be nuanced by the fact that the function waits a certain amount for a packet. It reduces the overall CPU utilization. picoquic-dpdk does

not have such a mechanism. It simply tries to receive a packet and continue the loop execution whether it managed to receive packets or not, which uses a higher percentage of the CPU.

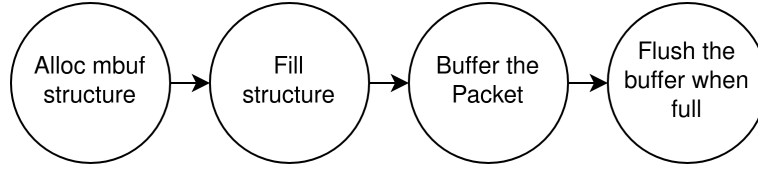
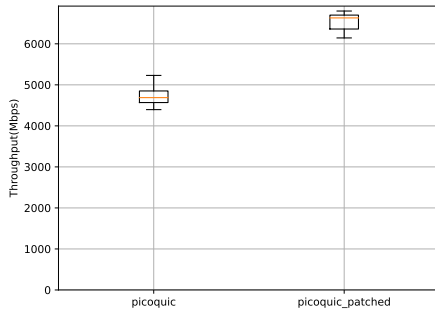


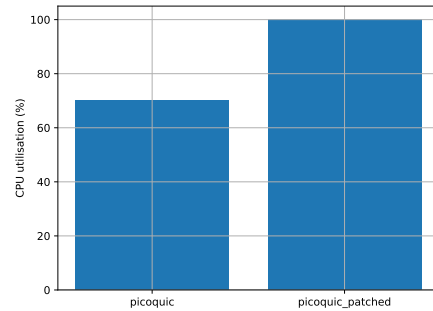
Figure 4.10: Sending a packet with DPDK

4.6 Cost of select

During our profiling, we observed that calls to *select* take around 20% of the CPU cycles for the client. *select* is a function that can listen to socket descriptors to see if new packets are available. This function has a *timeout* argument, which corresponds to the maximum amount of time that the *select* call waits for a packet before continuing. In picoquic this timeout delay is not always adequate to maximize the goodput.



(a) Goodput comparison



(b) CPU utilisation comparison

Figure 4.11: Comparison for goodput and CPU utilisation

Figure 4.11 compares picoquic and a patched version where we set the *timeout* argument to 0, meaning that the call does not block. Figure 4.11a shows performance improvement from 4800 Mbps to around 5700 Mbps on average. This increase has a cost. Figure 4.11b shows that the CPU utilization increases with *picoquic_patched*. The *picoquic* client uses around 70% of the CPU, whereas *picoquic_patched* uses 100%.

4.7 Interoperability

During the profiling, we observed that the cost of I/O is exceptionally high on the client-side. The limiting factor for the maximum achievable goodput seems to be the client. In this section, we are going to measure the goodput achieved when running a picoquic client and a picoquic-dpdk server as well as a picoquic-dpdk client and a picoquic server.

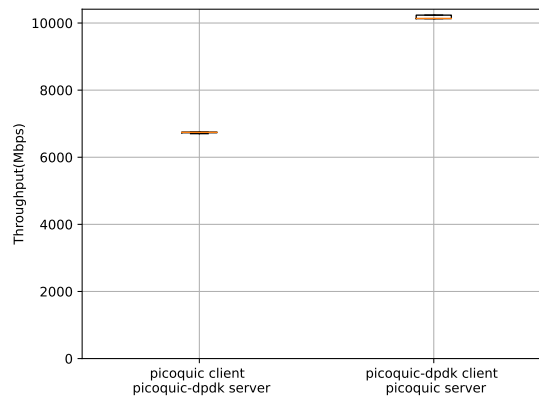


Figure 4.12: Interop performance

Figure 4.12 shows the goodput achieved during a 20 GB transfer from the server to the client. The patch explained in section 4.6 is applied to picoquic to remove the cost of *select* and focus on the cost of I/O. The left boxplot shows a scenario where the client runs picoquic, and the server runs picoquic-dpdk. On the right, we have a client running picoquic-dpdk and a server running picoquic. We observe that the picoquic-dpdk client manages to achieve a higher goodput. These results confirm what we observed during the profiling: the bottleneck is the I/O of the client. Furthermore, this section shows that our implementation is correct and is interoperable with the original picoquic. To make the two implementations communicate correctly, we had to handle ARP requests on picoquic-dpdk. The ARP protocol is used to find the destination MAC that needs to be used to reach a given IP address. When a device wants to reach an IP X, it first verifies in its ARP table if there is an entry for X. If it is not the case, it sends an ARP request and waits for a response. When running picoquic-dpdk we take control of the NIC. Thus we need to handle this protocol ourselves. Tom Barbette implemented the management of the ARP protocol.

4.8 Cost of encryption

Encryption and decryption operations are very costly. To quantify the cost, we decided to run the implementation without encryption. To remove the cost of encryption, we decided to replace the encrypt and decrypt operations with a call to the *memcpy* function.

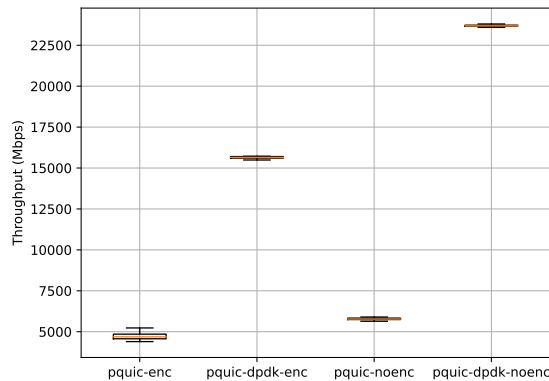


Figure 4.13: The cost of encryption

Figure 4.13 shows that in the case of picoquic-dpdk we see a lose of goodput of about 40% when enabling encryption. This result is in line with the 36% of cycles used for encryption we observed with perf when profiling. picoquic shows a similar trend in terms of performance reduction.

4.8.1 Comparing different encryption algorithms

We saw before that encryption is very costly. Up to this point, we only used one encryption algorithm, AES128, but others can be used in picoquic. We will look at them and analyze their performance. picoquic uses OpenSSL cryptographic operations and supports three different algorithms for encryption and decryption: AES-128-GCM, AES-256-GCM, and chacha20-poly1305. The AES algorithms are hardware accelerated and should provide better performance if the device has the appropriate hardware. chacha20-poly1305, on the other hand, doesn't take advantage of hardware acceleration and is thus generally slower.

AES

AES is the most widely used symmetric cipher today. It was first published in 2000. Symmetric meaning that the encryption key and the decryption key are the same.

It is a block cipher. It encrypts data by blocks of 128-bit using keys of length 128, 196, or 256 bit (AES128 uses 128-bit keys, etc.). Efficient hardware and software solutions exist. Our testing servers are equipped with Intel(R) Xeon(R) Silver 4314 CPU, which support AES-NI, an Intel technology that increases the performance of AES [1].

CHACHA

CHACHA is a newer cipher. It was proposed in 2008 and included in TLS 2016 [38]. Contrary to AES, it is a stream cipher that is generally more lightweight since they only use bit operations. They are used for IoT applications, for instance [25]. To encrypt a plaintext stream, stream ciphers xor it with a pseudorandom cipher digit stream. The pseudorandom stream is generated using a random seed that acts as the cryptographic key. Our servers don't have hardware acceleration for CHACHA crypto.

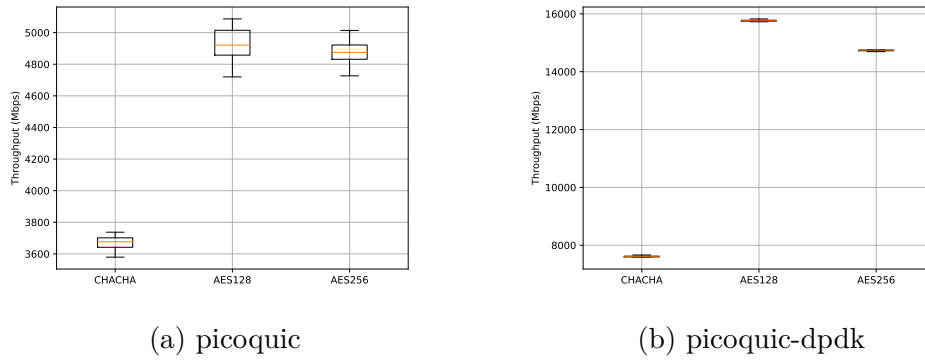


Figure 4.14: Performance with different encryption algorithms

As we expected, we see that AES performs better than CHACHA, thanks to hardware acceleration. AES128 slightly outperforms AES256 in both scenarios. It is explained by the fact that AES256 requires more operations than AES128 to encrypt a single block.

4.9 Comparison with other stacks

To get comparison points for both picoquic and picoquic-dpdk, in this section, we are going to compare them to two other QUIC implementations: msquic [44] and quiche [13], and to a TLS stack: picoTLS[21].

4.9.1 A first comparison

The cipher suite is restricted to *AES_128_GCM_SHA256* for every stack to compare the implementation fairly. To get the maximum performance out of picoTLS we had to pin the processes to the CPU core not used by the kernel by using the taskset [2] utility. To measure the performance of picoTLS we use the provided *cli* and the -B option to measure the maximum sustainable bandwidth achievable.

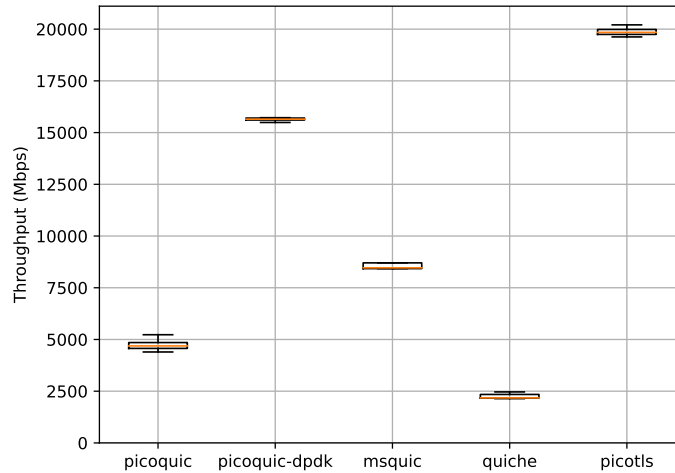


Figure 4.15: Comparison with other implementation of QUIC and TLS

Figure 4.15 shows that the TLS stack outperforms every other stack. We also can see that picoquic-dpdk outperforms msquic significantly, even-though msquic is an optimized stack that focuses on performance. We should note that msquic is better optimized on windows [45], but since our servers run Linux only, we cannot evaluate the stack’s performance on Windows. quiche comes last, but it is not optimized for performance. quiche comes last, but the configuration for this stack is different. The client and the server run inside a docker container, making the comparison less straightforward. It would be interesting to make measurements for quiche outside the containers. The quiche client and server can be found on GitHub [34]. The repository is a fork from a project of François Michel.

4.9.2 picoquic vs msquic

In this section, we profile msquic, to understand why it achieves better goodput than picoquic.

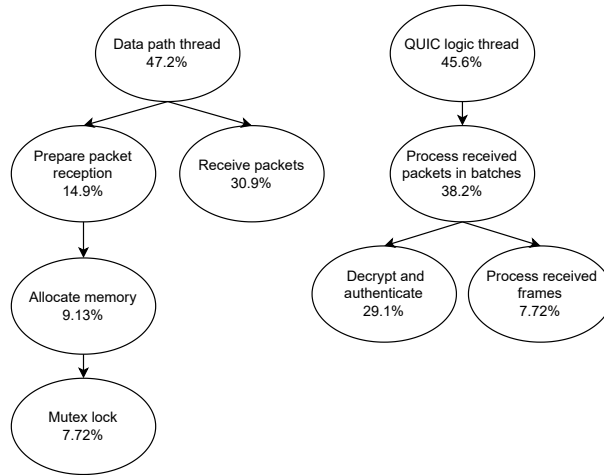


Figure 4.16: msquic execution

The two stacks differ in the number of threads used. Where picoquic uses a simple architecture where one thread is used for everything, msquic splits the tasks between different threads. Figures 4.16 shows the hotspots of a msquic client. msquic uses two main threads, the data path thread, which is responsible for handling the reception of packets and buffering them inside an internal buffer, and the QUIC logic thread, which retrieves packets from internal buffers and processes them in batches for better efficiency.

4.9.3 A fairer comparison

Our first comparison had a significant limitation. All stacks do not use the same number of CPU cores as we saw with msquic. picoquic-dpdk is at a disadvantage because it only uses one CPU core for I/O and packet processing, whereas the other stacks rely on the kernel for I/O and thus use multiple CPU cores. On our servers, when the CPU receives interrupts from the NIC to indicate that a packet has been received, they are load balanced between the different CPU cores with irqbalance [3] to improve performance.

Among the stacks that use the kernel, there are many discrepancies regarding the number of threads used. To make it fairer, we will only allocate one core for each stack for both the kernel and the application and analyze the results. We will use the following commands on both the client and the server :

- (1) `$ sudo ethtool -L ens1f0 combined 1`
- (2) `$ sudo systemctl stop irqbalance`
- (3) `$ sudo /usr/sbin/set_irq_affinity_cpulist.sh ens1f0`

Command (1) configure the NIC to have only one queue, command (2) removes the load balancing of interrupts, and finally, command (3) sets the interrupt affinity to CPU core 2. when running our future tests, we will use

```
(4) $ sudo taskset -c 2
```

to pin the processes to CPU core two, which is the same as the one used to handle the interrupts.

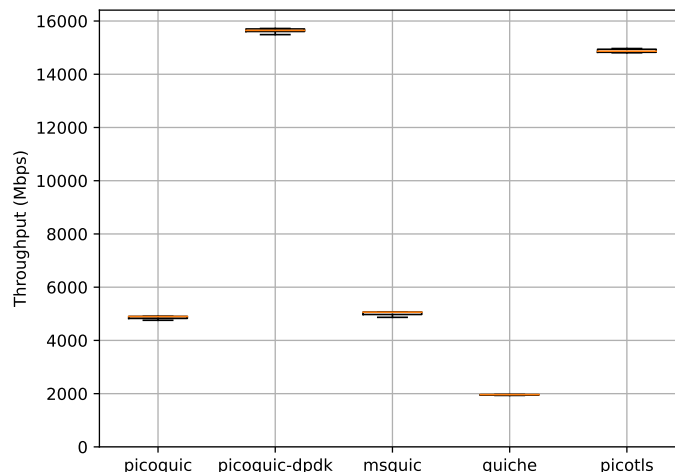


Figure 4.17: Comparison with other implementation of QUIC and TLS

Figure 4.17 gives us an interesting view. When limiting the number of cores used by the different stacks to 1 for both the kernel stack and the application, we see drastically different results from Figure 4.15. First, we see that picoquic-dpdk becomes the best-performing stack. Second, we observe that picoquic and msquic perform about the same. As we saw in section 4.9.2, msquic relies on multi-threading to improve its performance, as shown by the 8.55% of the CPU cycles spent in locks. Thus, limiting it to one core limits its performance.

4.10 Exploring multi-threading

In this section, we increase the performance of the picoquic-dpdk server, by balancing the load between different CPU cores using RSS.

4.10.1 Experimental setup

To test the performance of RSS, the following setup was chosen. The client machine is composed of separated NICs: one for each client. The server uses a single NIC and scales with the number of cores using RSS, where the Toeplitz hash function is applied on the classical 5-tuples.

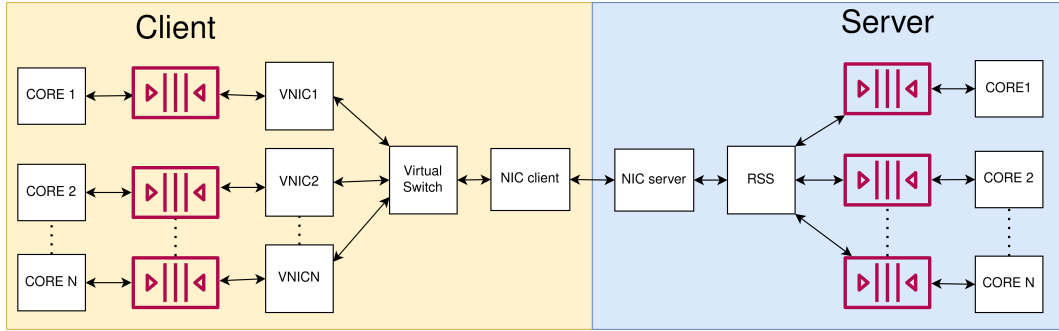


Figure 4.18: Experimental setup

4.10.2 Workload

The workload is the following: each client downloads a 20 GB file from the server and measures the goodput it receives. The graph shows the goodput that each client measures during the request. The workload is composed of 15 clients requesting a 20Gb download. 15 is chosen so that every client has a physical core since the system has 16 physical cores, and one core is needed for the master thread.

4.10.3 Analysis

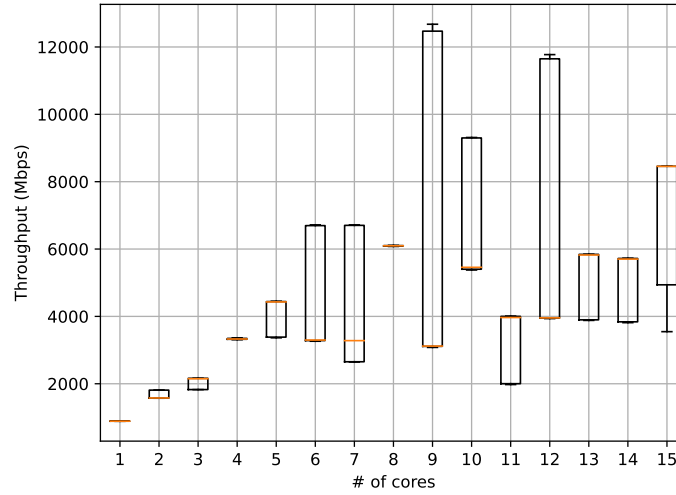
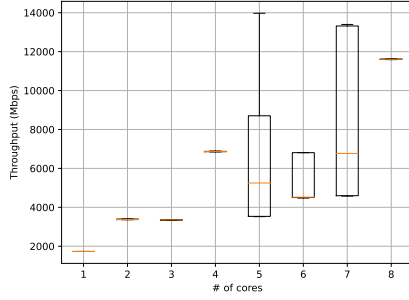
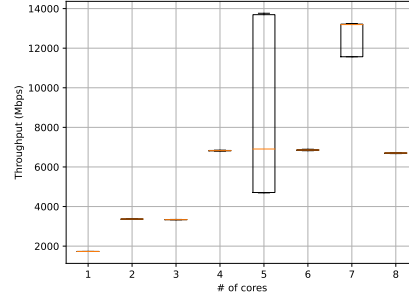


Figure 4.19: Scaling with RSS

Figure 4.19 shows that the server seems to scale relatively linearly with the number of cores until it reaches five cores. We can see dips in performance in multiple places, such as 6 or 11 cores. It can be explained by the fact that the hash function does not consistently distribute the load evenly between all the available CPU cores. The performance is very good with eight cores where the load is well balanced. This can be attributed to the maximum throughput of the NIC, which is 100Gbps. We have redone the measures with eight clients to verify this assumption. To see the possible imbalances caused by RSS, we also decided to redo the measurements, but with different IPs for the clients, we expect to see other imbalances.



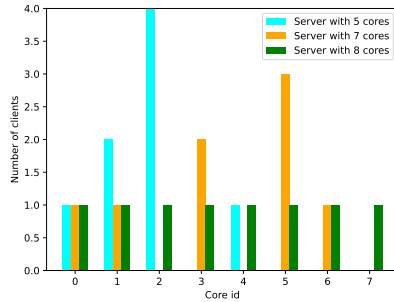
(a) 198.18.X.1 IPs



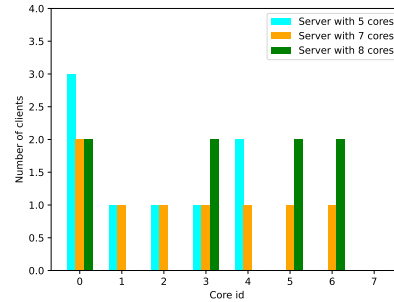
(b) 20.X.0.1 IPs

Figure 4.20: Comparing performance with two different set of IPs for the clients

To assign IPs to the clients in scenario (a), IPs of the form 198.18.X.1 are used. X corresponds to the ID of the client ranging from 1 to 8. Scenario (b) uses the 20.X.0.1 format. In scenario (a), we see a perfect balancing of the clients between the server's core with 4 and 8 server cores. In the case of 4 cores, every core handles two clients. With eight cores, every core takes a single client, and each client receives about 11 Gbps which saturates the NIC.



(a) 198.18.X.1 IPs



(b) 20.X.0.1 IPs

Figure 4.21: Comparing performance with two different set of IPs for the clients

Figure 4.21 gives us another view on the distribution of the client between the CPU cores. For clarity reasons, we choose to show the server with 5, 7 and 8 cores because it is the most interesting cases. The results confirm our analysis on figures 4.19 and 4.20. Scenario (a) shows that with 8 server cores, the client are served by one server core whereas scenario (b) cores 0,3,5 and 7 receive 2 clients each. The great variance with 5 server cores on scenario (a) is also explained by our new figure. We observe that CPU core has to handle 4 clients whereas CPU core 0,2 and 4 handle one client each. This create lots of variance in the goodput

received by the clients. Variance in scenario (a) with 7 cores is explained by the same principle. Scenario (b) shows higher average goodput because the clients are well distributed among the server cores. Each server core handle 1 client expect for core 0 which has to handle 2 clients.

4.10.4 Fairness

During our experiments, we showed that RSS distributes the load poorly and that this distribution depends on the addresses of the client. We used a relatively small number of clients (8 and 16). We expect better load balancing with an increased number of clients and a higher entropy in the addresses.

4.11 Conclusion

In this chapter, we compared picoquic and picoquic-dpdk in different performance metrics. We then profiled an execution of both to detect bottlenecks and hotspots. We further analyzed the stacks by removing the bottlenecks we observed during profiling. To get an idea of the performance of picoquic-dpdk we compared it against another QUIC implementation: msquic, and against a TLS+TCP stack. Finally, we improved the performance of the server of picoquic-dpdk by using RSS to spread the load between different CPU cores. We observed the performance improvement as well as the limitations of RSS.

Chapter 5

VPN prototype

In this chapter, we design a simple tunneling prototype, make measurements and compare it to Wireguard [15] a state-of-the-art tunneling solution.

5.1 The prototype

We designed a prototype similar to the relay architecture discussed in section 2.2.6 to evaluate the performance of picoquic-dpdk for packet tunneling. The prototype is a minimalistic version. Each relay encapsulates every packet it receives and sends it to its peer. For the tunneling, we use DATAGRAM frames. The forwarding of UDP and TCP packets is tested. The following configuration is used to test our relays.

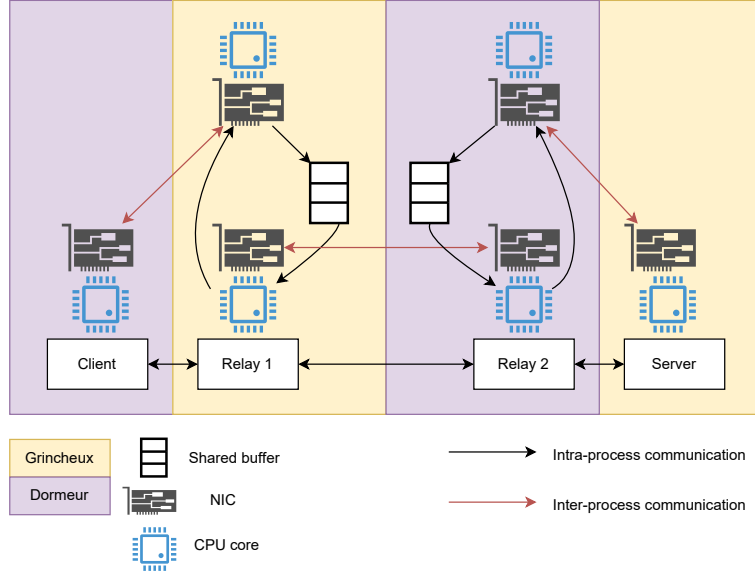


Figure 5.1: Prototype setup

Until this point, we only used the run to completion model, meaning that packets were received and processed by the same CPU core. This system is an opportunity to experiment with the pipeline model. Here is how one relay works, one CPU core listens to a NIC connected to the outside. When packets are received, it stores them inside a *rte_ring* structure, which is a concurrent ring buffer offered by DPDK. It can operate in multiple modes:

- multiple producers and multiple consumers,
- multiple producers and single consumers,
- single producer and multiple producers,
- single producer and a single consumer.

The last mode is used for our relays. A second core listens to a NIC attached to the other relay in the system. This core is charged to run *picoquic-dpdk* in either client or server node. When the application is asked to provide data to send by the callback mechanism, it reads packets from the buffer and feeds them to the application. When a DATAGRAM is received, the de-capsulated packet is sent to the outside through the outside NIC.

As shown in figure 5.1, the processes are dispatched between two physical servers. This setup is not ideal, and four servers would have been ideal since it would be possible to put each process on a separate server. Unfortunately, we did

not have access to four servers. Two processes that exchange packets are always put on different physical servers to make it as realistic as possible. Furthermore, each process has exclusive access to a set of CPU cores. Relays use two cores and a shared ring buffer to increase performance. The ring buffer is used through the peek API to maximize small packet performance: the proxy always tries to make complete QUIC packets. It fills the QUIC packet with DATAGRAM frames until it is complete. We peek inside the buffer to see if the subsequent packet is small enough to fit inside the remaining space in the packet. If it is the case, we remove the packet from the buffer and create a DATAGRAM frame. Otherwise, we keep it inside the buffer. This solution has the drawback of blocking access to the ring buffer for a long time. To peek inside the ring buffer, we have to lock the buffer peek inside it, decide if we retrieve the packet, and unlock the buffer.

5.2 Measurements

In this section, we evaluate our prototype in different scenarios. We try tunneling UDP and TCP packets through the proxy to ensure they are working correctly. We then evaluate the throughput that we can achieve through the prototype. We study the impact of the tunneled packet's size on the performance. To avoid fragmentation issues, we set the maximum size of the payloads at 1200 bytes. We also determine the tunnel's achievable throughput without sustaining losses for the UDP workload. For the TCP tunneling, we look at the maximum achievable throughput that can go through the proxy. Latency measures are irrelevant in our scenario since no connections are opened and no requests are served. We are interested in the forwarding performances. We are testing both UDP and TCP flows because they will not necessarily behave the same way since TCP uses a congestion controller that can adapt the transmission rate and pacing.

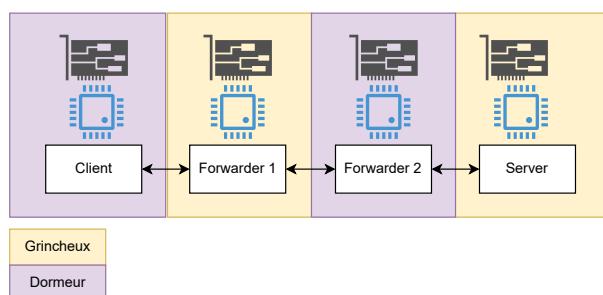


Figure 5.2: Forwarder setup

To get a benchmark of the maximum achievable throughput through relays, we designed a simple DPDK forwarder, which forwards packets from one port to

another without relying on a buffer. The forwarders receive packets, change the MAC source and destination fields and forward the packet. Those applications have virtually no overhead and provide performance equivalent to a setup where the client and the server are connected directly. Nevertheless, they are necessary because they give us a benchmark and help us spot possible bottlenecks from the clients/servers. The setup of the forwarder is shown in Figure 5.2.

UDP

Regarding UDP, we realized that iperf3 does not provide adequate performances. The transmission speed caps at around 6Gbps for 1000 bytes payloads without any losses, meaning that our proxy can go higher. We thus designed a sender and a receiver with DPDK to ensure they cannot be a bottleneck during testing. The sender sends packets with increasing rates until the receiver detects a loss or when the goodput of the receiver stops increasing, meaning that the ring_buffer is filling up.

A token bucket is used to set the rate at the desired values. To detect packet loss, the sender uses the first 4 bytes of the payload to write a counter which is incremented after each packet sent. The receiver keeps track of this counter. If the receiver receives a packet with a counter different from what it expects, it means that a loss occurred. It stops and records the achieved goodput. During our tests, we observed that when we detect a packet loss, we lose several 10's packets. Our criteria of stopping when we detect a single loss thus seem appropriate.

For UDP traffic, using the forwarders as a benchmark wasn't necessary since we manage to reach losses with every packet sizes and thus are ensured that the udpsender manages to send packets at a sufficient rate.

TCP

To make our measurements, we used the iperf3 [20] which is a simple utility used to run a client and a server to measure network performance. To ensure that iperf3 is not a bottleneck in our experiment, we tested the achievable goodput through our forwarders.

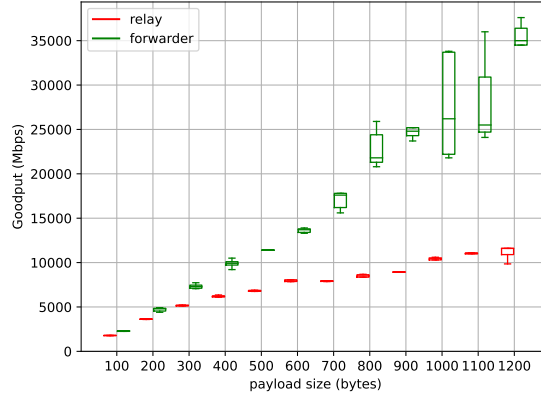


Figure 5.3: Goodput achieved by the relays and the forwarders

Our first observation is that TCP traffic can be transported through our system of relays. Secondly, we can see that compared to our simple forwarder which is the described below figure 5.2, the achieved goodput is way lower except for small payloads where both the relays and the forwarders reach a comparable goodput. This is expected since our proxy tunnels the traffic inside QUIC packets. This generates heavy operations not present with the forwarder, such as encryption. Finally, we see more variance with our forwarder, which is unimportant. The l2-forwarders are always faster except for very small packets, ensuring no bottlenecks from the client/server pair.

5.2.1 Comparing UDP and TCP workloads

In this section we compare the UDP and TCP workloads to determine differences between the two.

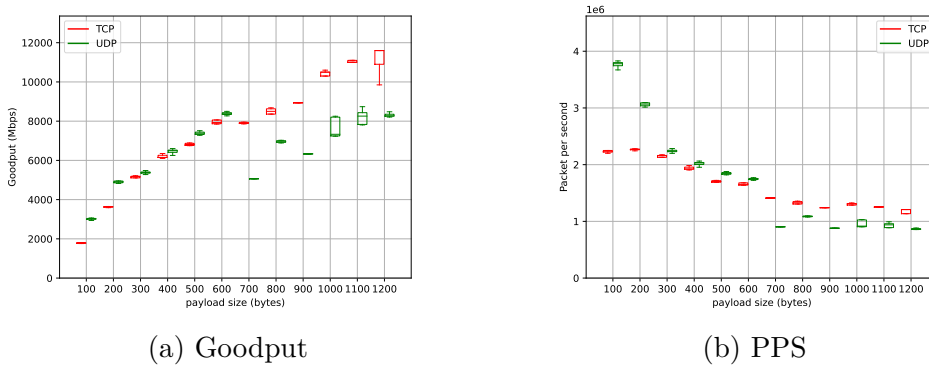


Figure 5.4: Performance when tunneling TCP packets

Figure 5.4 shows that until payloads of size 700 bytes, the relays manage to reach about the same goodput with both TCP and UDP traffic. The UDP workload seems to manage a slightly higher goodput for those values. This can, however, be because iperf3 performs poorly at those values, as we noted when we tested the relay against the forwarder.

The TCP flow reaches a higher goodput for packets with 700 bytes and above payloads. This is where the congestion controller makes a difference. It manages to reach a higher goodput without sustaining losses. iperf3 is configured to use the BBR congestion controller, which adapts the transmission rate and pacing to avoid packet losses.

5.2.2 Comparison with Wireguard and hybrid relays

This section compares Wireguard (version 1.0.20200513-1~20.04.2) with our tunneling prototype for both goodput and PPS (packets per second). The Wireguard setup is illustrated in figure 5.5, and the configuration for both interfaces is shown in listings 5.1 and 5.2. Interfaces wg0a and wg0b are the Wireguard interfaces. They are configured as peers. Namespaces are used to facilitate routing. The routing tables have been configured such that packets received on if0a from 1.0.0.1 are forwarded to 6.0.0.2, and packets received on if1b from 3.0.0.1 are forwarded to 6.0.0.1. This means traffic going from 1.0.0.1 to 3.0.0.1 is encrypted in both directions: between wg0a and wg0b.

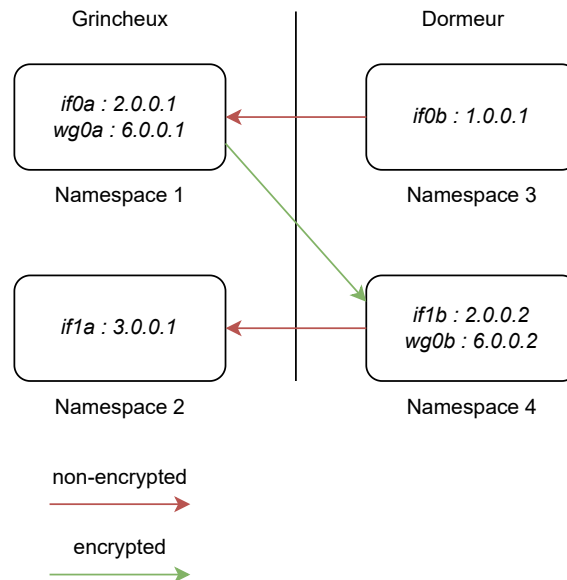


Figure 5.5: Wireguard setup

Listing 5.1: Config for *wg0a*

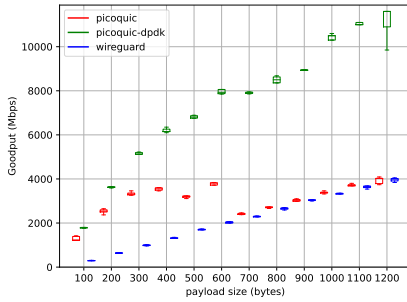
```
# local settings for Host a
[Interface]
PrivateKey = ...
Address = 6.0.0.1/32
ListenPort = 51821

# remote settings for Host b
[Peer]
PublicKey = ...
Endpoint = 2.0.0.2:51822
AllowedIPs = 0.0.0.0/0
```

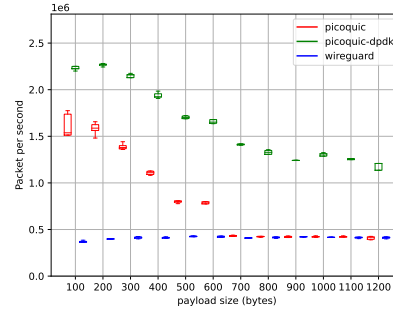
Listing 5.2: Config for *wg0b*

```
# local settings for Host b
[Interface]
PrivateKey = ...
Address = 6.0.0.2/32
ListenPort = 51822

# remote settings for Host a
[Peer]
PublicKey = ...
Endpoint = 2.0.0.1:51821
AllowedIPs = 0.0.0.0/0
```



(a) Goodput comparison



(b) PPS

Figure 5.6: Comparison for goodput and PPS

We added our relays in hybrid mode as another point of reference. In this mode, the relays use 2 NICs. The first one, as before, is used to receive and send packets from and outside the tunnel. The second one is used for the picoquic connection in non-dpdk mode.

Figure 5.6 show that our relays outperform Wireguard. The PPS view is interesting. While Wireguard has constant PPS, our relays show decreasing performance with longer payloads. Our relay always tries to send complete QUIC packets, meaning that when it receives small packets, it creates multiple DATAGRAM frames inside one QUIC packet. This reduces the number of encryption necessary and increases the achievable goodput. The comparison between relays in hybrid mode and Wireguard is interesting. When the packets become too long, and only one IP packet can fit inside the QUIC packet, we see that picoquic and Wireguard perform similarly. This is expected since both of them rely on UDP. The limiting factor for both of them is the UDP networking stack. One advantage of using QUIC for tunneling is the congestion controller. Wireguard forwards packets one by one without considering the network’s congestion. Our prototype relies on CUBIC to adapt the transmission rate to the conditions of the network.

5.2.3 Other use cases

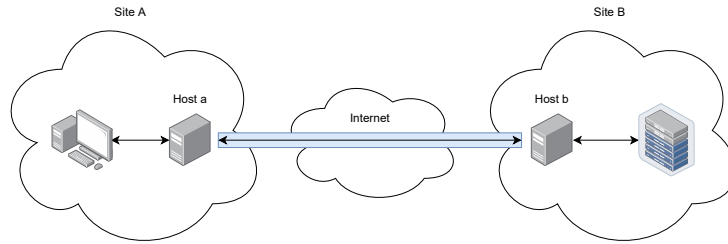


Figure 5.7: Site-to-site topology

We wanted to evaluate our prototype for the iCloud relays-like system because it is a good use case for QUIC. It is not the only use case where a tunnel can be useful. VPNs have a wide of applications. For instance, they can also be used for site-to-site topology [58], where a company wants to connect a branch office to its main office. For this kind of use case, Wireguard can be used. We showed that our prototype outperforms Wireguard for all payload sizes when looking at the achievable goodput through the tunnel.

5.3 Conclusion

In this chapter, we designed a tunneling application and evaluated it. We evaluate UDP and TCP traffic and observe that they do not behave the same way when passing through the tunnel. Next, we compared our relay with Wireguard and another version of the relay running picoquic instead of picoquic-dpdk. We observed that the picoquic-dpdk version outperforms the other two tunnels. Furthermore, for both the picoquic relay and Wireguard, the performance bottleneck is the UDP stack. Finally, we highlighted another use case for our relay: connect remote sites.

Chapter 6

Conclusion and future work

We first explored the DPDK library and modified the existing picoquic implementation. We detailed the necessary steps and choices made in the implementation. Finally, we highlighted the limitation of our approach and the improvement that can be made in the future. In the future, we would like to integrate our modifications into the picoquic github.

Profiling the execution of picoquic and picoquic-dpdk showed us that Linux’s networking stack system calls are very costly. When receiving packets, the call to *recvmsg* takes almost a third of the CPU cycles. Replacing the POSIX calls with their DPDK counterparts showed significant improvement in both goodput (3x) and latency (30%) metrics. Encryption was identified as costly as well. We explored different encryption algorithms proposed by picoTLS and compared them. Offloading encryption to the NIC would be an exciting path to explore in the future.

To improve the performance of the server-side of picoquic-dpdk, we used multi-threading backed by RSS. This approach gave us mixed results. On the one hand, the server seems to scale linearly when the load is well distributed between the different cores, but there are often many imbalances in practice. Scheduling requests across multiple cores efficiently is an active area of research. In future work, we would like to explore the possibilities that smartNICs [29] offer to improve scheduling.

Finally, we designed a tunneling application to evaluate the DATAGRAM mode of QUIC. We showed that UDP and TCP traffic could be tunneled through our application. We compared our solution to Wireguard, a state-of-the-art solution, and showed that our application provides between 3x and 5x greater goodput depending on the length of the packets transported. The prototype lacks features such as a NAT to hide the address of the client wanting to make a request. It would be interesting to continue improving it and provide a fully functional application.

All the code used in this thesis can be found on GitHub [33]. At the time of writ-

ing, the code is on branch `oldpquic` waiting to be merged with the main branch with a more recent version of `picoquic`. A readme file (`picoquic_DPDK_README.md`) is provided, indicating where to find the code used for this thesis.

Bibliography

- [1] URL: <https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf>.
- [2] URL: <https://man7.org/linux/man-pages/man1/taskset.1.html>.
- [3] URL: <https://linux.die.net/man/1/irqbalance>.
- [4] Apple. *icloud private relay overview*. URL: https://www.apple.com/privacy/docs/iCloud_Private_Relay_Overview_Dec2021.PDF.
- [5] Aviviano. *Overview of single root I/O virtualization (SR-IOV) - windows drivers*. URL: <https://docs.microsoft.com/en-us/windows-hardware/drivers/network/overview-of-single-root-i-o-virtualization--sr-iov->.
- [6] David Barach et al. “Batched packet processing for high-speed software data plane functions”. In: *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE. 2018, pp. 1–2.
- [7] Tom Barbette et al. “RSS++ load and state-aware receive side scaling”. In: *Proceedings of the 15th international conference on emerging networking experiments and technologies*. 2019, pp. 318–333.
- [8] Adam Belay et al. “IX: A Protected Dataplane Operating System for High Throughput and Low Latency”. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 49–65. ISBN: 978-1-931971-16-4. URL: <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay>.
- [9] M. Bishop. *HTTP/3*. RFC 9114. RFC Editor, 2022.
- [10] Neal Cardwell et al. “BBR: congestion-based congestion control”. In: *Communications of the ACM* 60.2 (2017), pp. 58–66.

- [11] Ivano Cerrato, Mauro Annarumma, and Fulvio Risso. “Supporting Fine-Grained Network Functions through Intel DPDK”. In: *2014 Third European Workshop on Software Defined Networks*. 2014, pp. 1–6. DOI: 10.1109/EWSDN.2014.33.
- [12] Ruining Chen and Guoao Sun. “A Survey of Kernel-Bypass Techniques in Network Stack”. In: *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*. CSAI '18. Shenzhen, China: Association for Computing Machinery, 2018, 474–477. ISBN: 9781450366069. DOI: 10.1145/3297156.3297242. URL: <https://doi.org/10.1145/3297156.3297242>.
- [13] Cloudflare. *Cloudflare/Quiche: savoury implementation of the QUIC transport protocol and HTTP3*. URL: <https://github.com/cloudflare/quiche>.
- [14] Yong Cui et al. “Innovating transport with QUIC: Design approaches and research challenges”. In: *IEEE Internet Computing* 21.2 (2017), pp. 72–76.
- [15] Jason A Donenfeld. “Wireguard: next generation kernel network tunnel.” In: *NDSS*. 2017, pp. 1–12.
- [16] Benjamin Dowling and Kenneth G Paterson. “A cryptographic analysis of the WireGuard protocol”. In: *International Conference on Applied Cryptography and Network Security*. Springer. 2018, pp. 3–21.
- [17] *DPDK documentation*. https://doc.dpdk.org/guides/prog_guide/index.html. Accessed: 2022-06-20.
- [18] Pratiksha Narasimha Nayak G et al. “A Detail Survey on QUIC and its Impact on Network Data Transmission”. In: *2022 6th International Conference on Trends in Electronics and Informatics (ICOEI)*. 2022, pp. 378–385. DOI: 10.1109/ICOEI53556.2022.9777199.
- [19] *Google transparency report*. URL: <https://transparencyreport.google.com/https/overview?hl=en>.
- [20] Vivien GUEANT. *Iperf - the ultimate speed test tool for TCP, UDP and SCTPTEST the limits of your network + internet neutrality test*. URL: <https://iperf.fr/>.
- [21] h2o. *H2O picotls: TLS 1.3 implementation in C (master supports RFC8446 as well as draft-26, -27, -28)*. URL: <https://github.com/h2o/picotls>.
- [22] Sangtae Ha, Injong Rhee, and Lisong Xu. “CUBIC: a new TCP-friendly high-speed TCP variant”. In: *ACM SIGOPS operating systems review* 42.5 (2008), pp. 64–74.
- [23] J. Iyengar and I. Swett. *QUIC Loss Detection and Congestion Control*. RFC 9002. RFC Editor, 2021.

- [24] J. Iyengar and M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. RFC Editor, 2021.
- [25] Sameeh A Jassim and Alaa K Farhan. “A survey on stream ciphers for constrained environments”. In: *2021 1st Babylon International Conference on Information Technology and Science (BICITS)*. IEEE. 2021, pp. 228–233.
- [26] Dharmanshu Johar, Minal Moharir, and Devendra Bhardwaj. “An experimental review on Intel DPDK L2 forwarding”. In: *International Journal of Applied Engineering Research* 12.18 (2017), pp. 7833–7837.
- [27] Jithin Jose et al. “SR-IOV support for virtualization on infiniband clusters: Early experience”. In: *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE. 2013, pp. 385–392.
- [28] Anuj Kalia, Michael Kaminsky, and David G Andersen. “Design guidelines for high performance {RDMA} systems”. In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 2016, pp. 437–450.
- [29] Georgios P Katsikas et al. “What you need to know about (smart) network interface cards”. In: *International Conference on Passive and Active Network Measurement*. Springer. 2021, pp. 319–336.
- [30] C. Kaufman et al. *Internet Key Exchange Protocol Version 2 (IKEv2)*. STD 79. <http://www.rfc-editor.org/rfc/rfc7296.txt>. RFC Editor, 2014. URL: <http://www.rfc-editor.org/rfc/rfc7296.txt>.
- [31] Kdab. *KDAB/Hotspot: The linux perf gui for performance analysis*. URL: <https://github.com/KDAB/hotspot>.
- [32] S. Kent. *IP Encapsulating Security Payload (ESP)*. RFC 4303. <http://www.rfc-editor.org/rfc/rfc4303.txt>. RFC Editor, 2005. URL: <http://www.rfc-editor.org/rfc/rfc4303.txt>.
- [33] Kikito07. *Kikito07/picoquic: Minimal implementation of the Quic Protocol*. URL: <https://github.com/Kikito07/picoquic>.
- [34] Kikito07. *Kikito07/Quic-implements-Dockers*. URL: <https://github.com/Kikito07/quic-implements-dockers>.
- [35] Mike Kosek, Tanya Shreedhar, and Vaibhav Bajpai. “Beyond quic v1: A first look at recent transport layer ietf standardization efforts”. In: *IEEE Communications Magazine* 59.4 (2021), pp. 24–29.
- [36] Hugo Krawczyk. “New hash functions for message authentication”. In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 1995, pp. 301–310.
- [37] Rustam Lalkaka. *Icloud Private Relay: Information for Cloudflare customers*. 2022. URL: <https://blog.cloudflare.com/icloud-private-relay/>.

- [38] A. Langley et al. *ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)*. RFC 7905. RFC Editor, 2016.
- [39] Adam Langley et al. “The QUIC Transport Protocol: Design and Internet-Scale Deployment”. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM ’17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, 183–196. ISBN: 9781450346535. DOI: 10.1145/3098822.3098842. URL: <https://doi.org/10.1145/3098822.3098842>.
- [40] *Lib/ethdev/rte_ethdev.h file reference*. URL: https://doc.dpdk.org/api/rte_ethdev_8h.html#a3e7d76a451b46348686ea97d6367f102.
- [41] Moritz Lipp et al. “Meltdown”. In: *arXiv preprint arXiv:1801.01207* (2018).
- [42] *Main page*. URL: https://perf.wiki.kernel.org/index.php/Main_Page.
- [43] François Michel, Quentin De Coninck, and Olivier Bonaventure. “QUIC-FEC: Bringing the benefits of Forward Erasure Correction to QUIC”. In: *2019 IFIP Networking Conference (IFIP Networking)*. 2019, pp. 1–9. DOI: 10.23919/IFIPNetworking.2019.8816838.
- [44] Microsoft. *Microsoft/msquic: Cross-platform, C implementation of the IETF QUIC protocol*. URL: <https://github.com/microsoft/msquic>.
- [45] *MSQUIC Performance dashboard*. URL: <https://microsoft.github.io/msquic/index.html>.
- [46] Mtcp-Stack. *MTCP-stack/MTCP: MTCP: A highly scalable user-level TCP stack for Multicore Systems*. URL: <https://github.com/mtcp-stack/mtcp>.
- [47] Lars Müller. “Kpti a mitigation method against meltdown”. In: *Advanced Microkernel Operating Systems* (2018), p. 41.
- [48] Késsia Nepomuceno et al. “QUIC and TCP: a performance evaluation”. In: *2018 IEEE Symposium on Computers and Communications (ISCC)*. IEEE. 2018, pp. 00045–00051.
- [49] *OpenVPN cryptographic layer*. 2019. URL: <https://openvpn.net/community-resources/openvpn-cryptographic-layer/>.
- [50] Amy Ousterhout et al. “Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads”. In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 361–378. ISBN: 978-1-931971-49-2. URL: <https://www.usenix.org/conference/nsdi19/presentation/ousterhout>.
- [51] B Padmavathi and S Ranjitha Kumari. “A survey on performance analysis of DES, AES and RSA algorithm along with LSB substitution”. In: *IJSR, India 2* (2013), pp. 2319–7064.

- [52] *Page weight*. 2022. URL: https://httparchive.org/reports/page-weight?start=2021_07_01&end=latest&view=list.
- [53] Lucas Pardue. *Unlocking Quic’s proxying potential with Masque*. 2022. URL: <https://blog.cloudflare.com/unlocking-quic-proxying-potential/>.
- [54] T. Pauly, E. Kinnear, and D. Schinazi. *An Unreliable Datagram Extension to QUIC*. RFC 9221. RFC Editor, 2022.
- [55] Johannes Pfau et al. “A hardware perspective on the ChaCha ciphers: Scalable Chacha8/12/20 implementations ranging from 476 slices to bitrates of 175 Gbit/s”. In: *2019 32nd IEEE International System-on-Chip Conference (SOCC)*. IEEE. 2019, pp. 294–299.
- [56] George Prekas, Marios Kogias, and Edouard Bugnion. “Zygos: Achieving low tail latency for microsecond-scale networked tasks”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 325–341.
- [57] Private-Octopus. *Private-octopus/picoquic: Minimal implementation of the Quic Protocol*. URL: <https://github.com/private-octopus/picoquic>.
- [58] Muhammad Siraj Rathore et al. “Site-to-Site VPN Technologies: A Survey”. In: (2009).
- [59] E. Rescorla and N. Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347. <http://www.rfc-editor.org/rfc/rfc6347.txt>. RFC Editor, 2012. URL: <http://www.rfc-editor.org/rfc/rfc6347.txt>.
- [60] Luigi Rizzo. “netmap: a novel framework for fast packet I/O”. In: *21st USENIX Security Symposium (USENIX Security 12)*. 2012, pp. 101–112.
- [61] Hugo Sadok, Miguel Elias M. Campista, and Luís Henrique M. K. Costa. “A Case for Spraying Packets in Software Middleboxes”. In: *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*. HotNets ’18. Redmond, WA, USA: Association for Computing Machinery, 2018, 127–133. ISBN: 9781450361200. DOI: 10.1145/3286062.3286081. URL: <https://doi.org/10.1145/3286062.3286081>.
- [62] *Segmentation offloads*. URL: <https://www.kernel.org/doc/html/v5.12/networking/segmentation-offloads.html>.
- [63] Mukesh Soni and Brajendra Singh Rajput. “Security and performance evaluations of QUIC protocol”. In: *Data Science and Intelligent Applications*. Springer, 2021, pp. 457–462.
- [64] M. Thomson and S. Turner. *Using TLS to Secure QUIC*. RFC 9001. RFC Editor, 2021.

- [65] *toeplitz hash library - data plane development kit 22.07.0-rc1 documentation*. https://doc.dpdk.org/guides/prog_guide/toeplitz_hash_lib.html. Accessed: 2022-06-15.
- [66] Martino Trevisan et al. “Measuring HTTP/3: Adoption and Performance”. In: *2021 19th Mediterranean Communication and Computer Networking Conference (MedComNet)*. 2021, pp. 1–8. DOI: 10.1109/MedComNet52149.2021.9501274.
- [67] *Wiki*. URL: https://wiki.linuxfoundation.org/networking/sk_buff.
- [68] Hongjun Zhang et al. *FastUDP: a highly scalable user-level UDP framework in multi-core systems for fast packet I/O*. en. 2020. DOI: 10.1007/s11227-020-03486-6. URL: <http://dx.doi.org/10.1007/s11227-020-03486-6>.
- [69] Irene Zhang et al. “I’m not dead yet! the role of the operating system in a kernel-bypass era”. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. 2019, pp. 73–80.
- [70] Yiwen Zhang et al. “Justitia: Software {Multi-Tenancy} in Hardware {Kernel-Bypass} Networks”. In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 2022, pp. 1307–1326.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl