Improving Transport Protocols for better flexibility

Maxime Piraux

Thesis submitted in partial fulfilment of the requirements for the Degree of Doctor in Engineering Sciences and Technologies

December 2024

ICTEAM
Louvain School of Engineering
Université catholique de Louvain
Louvain-la-Neuve
Belgium

Thesis Committee:

Pr. Olivier **Bonaventure** (Advisor)	UCLouvain/ICTEAM, Belgium
Pr. Cristel **Pelsser**	UCLouvain/ICTEAM, Belgium
Pr. Quentin **De Coninck**	UMONS, Belgium
Pr. Michio **Honda**	University of Edinburgh, Scotland
Pr. Tom **Barbette**	UCLouvain/ICTEAM, Belgium
Pr. Charles **Pecheur** (Chair)	UCLouvain/ICTEAM, Belgium

Improving Transport Protocols for better flexibility

by Maxime Piraux

ICTEAM
Université catholique de Louvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
Belgium

À mes grands-parents Jean, Joséphine, Rocco et Stella

'Am on this journey,
A very long journey,
'Am on Jah Jah journey,
A long long journey,

Although the hills are steep and the mountains are highs,
I'll get over,
All the obstacles that they put in I way,
Still I get over,

I take one day at a time,
I solve one problem at a time,
I take one day at a time,
I solve one problem at a time,
. . .
Twinkle Brothers & Young Warrior - Long Journey,
as played by Jah Shaka

Acknowledgements

I would like first to thank my advisor, Olivier Bonaventure, and the IP Networking Lab that welcomed me for the past six years. I believe this thesis, accompanied by the other works I carried on at the lab, reflects the fruitful work environment that Olivier and the rest of the team set up over the years. I would like to specifically acknowledge the participation of Quentin De Coninck, François Michel, Thomas Wirtgen, Louis Navarre and Nicolas Rybowski to the core of the lab. I believe the frequent meetings we had, although sometimes enduring given the lengths of some of my interventions, where we exchanged ideas and helped one another build better solutions to the problems we were solving are core to my training during my PhD. They helped me gain a much larger understanding of the Internet than I would have had carrying solely my research. To Louis and Nicolas specifically, I believe I have passed on all the knowledge I gathered and received at the lab to you in turn. I am impressed by what you achieved so far and I am looking forward for the next steps on your ways.

I thank Lai Yi Ohlsen for the numerous discussions we had on and around the Internet. They played a crucial part in the later stages of my PhD, helping me to settle a nearly existential schism setting apart my deep interest for the Internet infrastructure and the incentives that brought it to the world. I think the first section of the first chapter is a short but good reflection of the complex and empowering perspectives you helped strengthened in my understanding.

I thank the members of my jury for their time evaluating this thesis. I hope it reflects their implications and insightful comments.

Contents

Preamble

Transport protocols are key to realise the services of Internet applications. Their foundations were established during the inception of the Internet in the 1980s. Since then, as the Internet grew in size and in diversity, transport protocols evolved to accommodate new use-cases and devices. The Internet is a running infrastructure with a broad network of actors participating in its evolution and leveraging it. As such, any change to the Internet must be carefully considered to avoid disrupting existing uses. A consequence of this constraint is that a number of these design choices from the 1980s are still impacting transport protocols and Internet applications today.

During the course of this thesis, the QUIC protocol was standardised and adopted by large service providers. It represents a major evolution addressing some of the shortcomings of previous transport protocols and providing modern transport services to applications. We start this thesis during the standardisation phase of QUIC with the development of its first test tool, which we also used to conduct an in-depth study of the behaviour of its first implementations. Beside its improvements, QUIC still imposes a strict layering assumption between the application and the transport layer. We then revisits this assumption by providing new means to applications for tuning and extending a QUIC implementation using eBPF bytecode. From our experience with QUIC, we then rethink how TCP and TLS are layered to enable equivalent modern transport services. The thesis also considers the coexistence of IPv4 and IPv6 and their impact on the end-to-end latency of transport connections by proposing an adaptative address family selection technique. Finally, we revisit the end-to-end principle and bring more flexibility to QUIC endpoints by enabling them to delegate their QUIC connections to other hosts, either temporarily or permanently.

These improvements to transport protocols and the manner they are used on the Internet are gathered in five contributions according to the structure that follows.

- Chapter 1 introduces the context, concepts and protocols of the Internet required to understand the contributions of our thesis. We start from the first protocols of the Internet, i.e, IP, TCP, UDP, DNS and HTTP. Then, we discuss more recent changes with the broader adoption of TLS, stressing the importance of security and privacy in Internet communications. Finally, we discuss how the Internet was extended with

IPv6 and the introduction of devices with multipath capabilities. We introduce the QUIC protocol and its MPQUIC extension, positioning them as addressing both security and path diversity in the Internet.

- Chapter 2 describes our first contribution with the first test tool for the QUIC protocol, which we used to study and improve its implementations during its standardisation phase. We explain its design and capabilities, as well as our observations from March 2018 to April 2019.

- Chapter 3 revisits the paradigm for extending Internet protocols and leverages the security of QUIC to enable applications to customise QUIC connections to their needs by injecting eBPF bytecode, named *plugins*, modifying the behaviour of an implementation. We discuss its first prototype, Pluginized QUIC (PQUIC), and demonstrate how applications can inject plugins inside PQUIC to monitor performance metrics of their QUIC connections and add a new message-mode transport services with QUIC Datagrams. Orthogonal plugins can be combined, such that injecting a multipath plugin and a QUIC Datagram plugin can be used to build a multipath QUIC VPN.

- Chapter 4 improves an effort to provide modern transport services with TCP and TLS, revisiting their strict layering assumption. We identify 7 limitations from the initial version of the protocol and describe our new version of the protocol, TCPLS v2, addressing these limitations. We integrated our implementation of TCPLS v2 into web applications and evaluated in two experiments.

- Chapter 5 takes a step back from the insides of the transport layer to address the path diversity arising from Internet address families and their impact on the latency of transport protocols. We formulate the lowest-latency address family selection problem as an online learning problem balancing exploration and exploitation. We validate our design with simulations using the RIPE Atlas dataset. Then, we demonstrate how DNS resolvers can aid hosts to select the lowest latency address family with a prototype. This prototype is evaluated in real-world experiments.

- Chapter 6 revisits the end-to-end principle to enable important energy savings on mobile devices and servers using the QUIC protocol. We introduce the concept of a *delegate*, i.e., a network node that can temporarily and securely take charge of transport connections on behalf of a host. We design the *QUIC delegation*, a mechanism that can be used to securely defer the state of a QUIC connection. We implement a prototype that we evaluate on a smartphone delegating its QUIC connections

to its access gateway. Our results indicate that delegating medium and long-lived QUIC connections can improve energy-efficiency with a factor from 2 to 12.

- Chapter 7 concludes this thesis by providing possible future directions for each of our contributions.

Bibliographic notes

Conference Publications

- Q. De Coninck, F. Michel, M. Piraux, F. Rochet, T. Given-Wilson, A. Legay, O. Pereira, and O. Bonaventure. "Pluginizing QUIC". in: *Proceedings of the 2019 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM*. 2019, pp. 19–24.
- F. Rochet, E. Assogba, M. Piraux, K. Edeline, B. Donnet, and O. Bonaventure. "TCPLS: Modern transport services with TCP and TLS". in: *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*. 2021, pp. 45–59.

Workshop Publications

- M. Piraux, Q. De Coninck, and O. Bonaventure. "Observing the Evolution of QUIC Implementations". In: *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. ACM. 2018, pp. 8–14.

Journal Publications

- M. Piraux and O. Bonaventure. "Adaptive Address Family Selection for Latency-Sensitive Applications on Dual-stack Hosts". In: *ACM SIGCOMM Computer Communication Review* (2024).

Artefacts

- Our QUIC test suite (Chapter 2) is publicly available [Pir18b], along with its associated scripts to process and analyse the produced data. In addition, we release in the public domain the data we collected using our public instance from March 2018 to September 2023 [Pir24a]. This constitutes more than 1800 daily test runs, totalling more than 23 Go of traces data.
- The PQUIC implementation (Chapter 3) and its plugins are publicly available at `https://pquic.org`.

- Our implementation of our new version of the TCPLS protocol (Chapter 4) is publicly available under an open-source licence [Pir24b], along with its integration into `wrk` [Pir22b] and `h2o` [Pir22a].
- Our modified DNS resolver (Chapter 5) integrating our solution, along with all the scripts needed to reproduce our experiments and figures are available along with the corresponding article [PB24a].
- Our modifications enabling the `quiche` QUIC implementation to perform QUIC connection delegation (Chapter 6) along with the experiments scripts will be made public upon publication of the an article summarising our results.

Miscellaneous Contributions

- M. Piraux, L. Navarre, N. Rybowski, O. Bonaventure, and B. Donnet. "Revealing the evolution of a cloud provider through its network weather map". In: *Proceedings of the 22nd ACM Internet Measurement Conference*. 2022, pp. 298–304.
- M. Piraux, T. Barbette, N. Rybowski, L. Navarre, T. Alfroy, C. Pelsser, F. Michel, and O. Bonaventure. "The Multiple Roles That IPv6 Addresses Can Play in Today's Internet". In: *SIGCOMM Comput. Commun. Rev.* 52.3 (Sept. 2022), pp. 10–18.
- R. Marx, M. Piraux, P. Quax, and W. Lamotte. "Debugging QUIC and HTTP/3 with qlog and qvis". In: *Proceedings of the applied networking research workshop*. 2020, pp. 58–66.
- C. Crochet, T. Rousseaux, M. Piraux, J.-F. Sambon, and A. Legay. "Verifying QUIC implementations using Ivy". In: *Proceedings of the 2021 Workshop on Evolution, Performance and Interoperability of QUIC*. 2021, pp. 35–41.
- N. Tyunyayev, M. Piraux, O. Bonaventure, and T. Barbette. "A high-speed QUIC implementation". In: *Proceedings of the 3rd International CoNEXT Student Workshop*. 2022, pp. 20–22.
- O. Bonaventure, Q. De Coninck, F. Duchêne, A. Gego, M. Jadin, F. Michel, M. Piraux, C. Poncin, and O. Tilmans. "Open educational resources for computer networking". In: *ACM SIGCOMM Computer Communication Review* 50.3 (2020), pp. 38–45.

Background 1

This chapter explains the concepts, techniques and protocols that are used in this thesis. We first introduce the Internet from a global perspective in Section 1.1 and 1.2. Then we detail each protocol and their mechanisms in the rest of this chapter.

1.1 The Internet

The Internet, in its common understanding, could be the biggest technological advance over the last 40 years that comes first to the mind of citizens of the Global North. The term has progressively aggregated a large set of technologies related to its functioning and use. As an example, the Web has quickly become an alias for the Internet, while the two comprises different sets of technologies and the former depends on the latter. Programs such as Web browsers interacts with the Internet to access documents embedding technologies such as HTML, JavaScript and CSS to present websites to their users. In addition, the steep increase in the number of devices that can leverage the Internet is also core to this perceived progress. In the middle of the 1980s, the Internet connected a thousand of computers that were permanently installed, with most of them residing in universities and research facilities within the United States. Today, the magnitude of connected *hosts* is in the order of tens of billions.

Next to their increasing number, their patterns of use of the Internet have diversified as well. The last decade saw the rising adoption of *smartphones*, which introduced mobile connectivity to a large fraction of users. This decade introduced even more devices into the life and home of their users, with *wearable* and home devices that have different patterns of Internet usage. Some devices comprise several Internet access technologies, such as Ethernet, Wi-Fi, cellular networks or even satellite communications, that they can use intermittently, continuously or simultaneously and to exchange various kinds of data. Specialised network technologies, such as Bluetooth, LoRa and the upcoming Matter & Thread, are catered for low-power devices and appliances often referred to as *Internet of Things* devices.

The complexity of the Internet as a system has quickly prompted its first contributors to organise it as a set of *protocols*. On the contrary of the road network, where the traffic regulations come as a single set of rules, the Inter-

net comprises several systems that segment the rules required to access it and group them for specific purposes. Each protocol defines a number of assumptions, internal behaviours and services it provides. Protocols can be *layered* on one another, such that the services of one protocol are used as a basis for another to expand upon. This organisation helps network researchers and engineers to reason on the functioning of the Internet, diagnose problems, improve and extend existing protocols, or even replace some protocols with others.

A part of the story of Internet is told from the perspective of researchers and engineers that built its technological foundations and who were implicitly supposed to evolve under unrestricted institutional freedom. This perspective can appear even more relevant as some of the technical choices they took are still in place in the current Internet infrastructure. To first introduce the Internet as a technological infrastructure, we adopt a more structuralist approach and highlight in this section the underlying forces that coexist with the development of the Internet. As recent efforts of researchers are considering questions of *sustainability* for the Internet, exploring at the same time definitions for its scope and means of achieving it, we argue this perspective is key to enable long-lasting and impactful changes.

The Internet infrastructure and the market it fostered over the years consist themselves of a network of actors. They range from state actors to online communities and private companies of various scales. Some are small service providers, others are large companies with revenues comparable to the gross domestic product of countries ranked in the top 40. These Internet actors are tied together by the power balance of the market economy that they take part in.

In the case of state actors, while one can note that many countries are taking part in the development of the Internet to progress their strategic agendas, all of them bound by a common productivist system. Even China, which is often cited as a rival in press media certainly because of the scale of its economic interests, exercises a form of *authoritarian state capitalism* [PRT21]. Various degrees of this taxonomy can be used for all state actors part of the Internet development. State actors may disagree on the degree of authority they should be able to exercise on the Internet or on the degree to which companies that operates on their market can profit from it but not on the nature of its economic incentives. In fact, the start of the rapid rise in the number of connected hosts during the 1980s [KK19] is contemporary to the fall of an alternative authoritarian productivist system as found within the Soviet Union. As such, the technologies built for the Internet were designed around capitalistic incentives. We argue that this perspective brings a richer understanding of the environment that built the Internet.

Online communities are part of the Internet since the 1980s with the de-

sign of the first form of Internet forums. They took many forms since then due to their *rhizomatic* nature [Høs17]. These groups are mostly users of the Internet, but some communities also develop new uses using new protocols and applications. For instance, the open-source community has been developing *decentralised* web applications as alternatives to the oligopoly of privately owned applications [Mön24]. Others, such as the cryptocurrencies communities have built networks for digital currencies as alternatives to state-owned currencies. Both communities have very different motivations, the former is building communal software avoiding the pitfalls of the incentives of private economic actors [GZ23] while the latter stems from right-libertarianism [Gol16]. However, all of these online communities are tributary of the underlying infrastructure that they are building on. As such, its extent limits the shifts in power dynamic that they can effectively introduce.

Together, by producing the Internet infrastructure with its sets of rules, policies and protocols, these Internet actors form a new type of governance. One venue for this Internet governance is the Internet Engineering Task Force (IETF). It tackles the technical aspects of the Internet through the standardisation of protocols and welcomes all kinds of these actors. It is also at the same time an online community, where ideas are discussed on public mailings lists and during meetings three times a year. The IETF produces standard documents, i.e., specifications of protocols written in plain-English, that are supposed to reflect the consensus of these actors as expressed within the IETF [IET24]. Internet actors then develop *implementations*, i.e., write code to form software, based on these specifications to access the Internet. These two activities often take place in parallel and form an iterative process in which the experience of each feeds the next step of the other.

The Internet also corresponds to an important physical reality besides the abstract and large virtual spaces it evokes. To realise their services, platforms leverage large datacenters scattered around the globe. These have an important footprint, both in terms of consumed energy and territorial impact [DL19]. In addition to being connected to the local Internet service providers, these datacenters are also connected across continents using submarine cables [Tel24]. These concentrate significant amount of traffic in few locations, weaving important strategic points with large geopolitical implications [BL21].

The study and description of power balance and dynamics within the Internet infrastructure encompass several fields of research. As this thesis rather focuses on technical aspects of this infrastructure, we end this section following this brief introduction. In this the rest of this chapter, we focus on the technical description of the relevant parts of the Internet to help the understanding of the improvements we develop within this thesis.

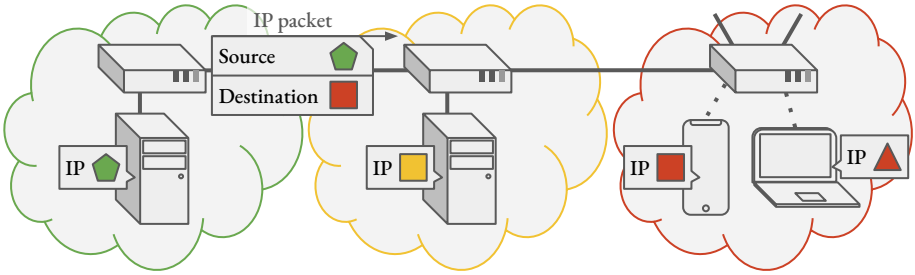


Figure 1.1: A simple Internet which connects three IP networks. IP addresses are configured such that their colour indicates a given IP network and their shape designates a device within this network. An IP packet is sent from the computer in the green network to the smartphone in the red network.

1.2 The Internet Protocol and IP networks

The Internet Protocol is the core protocol that gave the Internet its common name. It is almost always referred to using the *IP* initials. Naturally, this is the first protocol we discuss in this chapter. When the first local networks were established, they connected several computers through a set of cables, often forming a *bus* or *ring* topology. Using this network, each computer was able to reach any of the other connected ones, provided that it was the only one transmitting on the cable at a given time. In this scenario, when a computer is sending data, it can simply indicate the recipient using a hardware identifier, such as the serial number of its network interface.

As these networks grew in size and numbers in companies, factories, administrations and research labs during the 1970s, the need for a mean to connect them and enable the exchange of data between them was becoming apparent. The method used to identify recipients in local networks was insufficient to address the case of multiple networks, as a hardware identifier gives no information on to which network a computer belongs. The Internet Protocol addresses this problem and enables the forming of a *network of networks*. To do so, it defines IP addresses which have a broader scope than hardware identifiers. These can be dynamically assigned by network administrators to the network interfaces of the connected computers and have a degree of hierarchy that can be freely defined. When addressing a recipient using an IP address, this hierarchical property helps identifying the destination network where the data should be forwarded.

As an example, a simple network of three networks is illustrated on Figure 1.1. The values of IP addresses are represented as a combination of a shape and a colour. In this example, IP addresses are configured such that their colour indicates a given IP network, while their shape designates a device within this network.

The Internet Protocol comes with an important paradigm regarding the way data is exchanged over IP networks. It offers a *datagram* service based on IP *packets*. Each IP packet carries a limited amount of data from a source to a destination. A practical limit for the size of these packets on the Internet today is 1500 bytes. A dedicated space at the start of the packet is reserved to encode this information. IP networks offer no guarantee on the delivery of an IP packet. In fact, it may be lost, modified (*corrupted*), received once or multiple times. In addition, the sequence in which the IP packets are received may not correspond to their sending sequence.

To connect these networks together, IP *routers* are used to forward IP packets to the correct destination. IP routers are often dedicated equipment connecting several networks. They inspect each IP packet they receive and decide to which network it must be forwarded. When the destination network cannot be reached directly, i.e., because there is no direct connection to one of its routers, the router may choose to forward to packet to another network that can better reach the packet destination. There are many manners to establish and manage these *routing policies*, but we keep their description out of this introduction.

In our example, the computer in the green network sends an IP packet that must be conveyed to the smartphone in the red network. When the router in the green network receives the IP packet, it checks its destination. Its default route is to forward it to the yellow network. When the yellow network receives the forwarded IP packet, it checks its destination and decides to forward the packet further east as it knows that it is directly connected to the red network. When the router in the red network receives the IP packet, by checking its destination, it knows that this packet has to stay within its network given its colour and use the destination shape to decide to which device it should forward the packet.

1.3 Transport protocols

The service provided to applications by IP networks is minimal. It is often referred to as a *datagram* or *message* service. It can convey a small payload of data over IP networks in a *best-effort* manner. This is a restrictive interface for applications. Some, such as *emails*, one of the first Internet application, require more underlying services to operate correctly. In our example, one may reasonably expect that the emails that they send get effectively delivered, no matter what happens to the IP packets that are sent. In this case, an email application could include the required logic to overcome this gap, e.g. detect the missing packets and retransmit them. However, as more Internet applications are created, overlapping needs such as the need for a *reliable transfer* appeared. It becomes then more efficient to address this class of problem us-

ing a dedicated set of protocols on the behalf of the application. These are called *transport* protocols, as they offer data transportation services over IP networks to applications. They are said to be *layered* over IP as they send and receive IP packets. Each introduce its own format for the payload of the IP packets it produces. Similarly to IP, they often reserve a dedicated space to encode relevant information for their functioning.

There are several services that can be brought by transport protocols. Protocols typically implement only a combination of these services. We list the most common ones and group some of them. The first set of services are considered as basic and correspond to the needs of the first Internet applications.

- *Application multiplexing*: IP addresses identify network interfaces of connected computers. However, on a given computer, several applications are likely to run at the same time. Transport protocols often offer *ports*, which form a numerical space separated from IP addresses, to identify applications

- *Error detection*: As IP packets can become *corrupted*, e.g., they are altered by a media failure, this service ensures that a certain class of errors induced on a packet can be detected.

- *Reliable transfer*: This service ensures that a given data passed by the application is eventually received correctly.

- *Connection mode*: This service enables the establishment of transport connections, such that state is maintained to identify each transport event as part of a connection. Each connection is often established from two sets of addresses and ports. The host that initiates the connection is said to be the *client*, while the host receiving the connection request is called the *server*. When considering one of the two roles, the other role can be referred to as the *peer*, such that the peer of the client is the server. This service is often used to encode and maintain the transmission order of data.

- *Stream mode*: This service offers a bidirectional bytestream to the application. It ensures that the order in which bytes are delivered to the receiving application corresponds to the one of the sending application. Both hosts can be sender and receiver at the same time as the stream is bidirectional.

- *Message mode*: This service is identical to the one provided by IP packets, but as a transport protocol can implement several services, it is often combined with others.

The second correspond to features that were introduced more recently to support modern Internet applications.

- *Stream multiplexing*: This service extends the Stream mode by providing several concurrent bytestreams. It is often used by applications that require to transmit several objects at the same time.

- *Connection migration*: This service decouples the transport connection from the IP layer and enables hosts to change their source address. This is useful when one network interface becomes unavailable but others remain usable, e.g., when the Wi-Fi signal is lost but cellular data is available.

- *Multipath*: This service extends the number of addresses and ports that can be used to exchange packets of the transport protocol. Using these addresses and ports, several network paths can be established as part of the transport connection and used simultaneously. There are several algorithms to establish network paths and use them for exchanging application data. These are considered later as we discuss a transport protocol extension that implements the Multipath service in Section 1.11.

- *Metadata confidentiality*: In order to provide the services listed in this section, transport protocols typically encode information inside the packets they exchange. For instance, a transport protocol providing the Stream service is likely to encode the offset within the bytestream of each exchanged segment. When this information is encoded in *plaintext*, i.e., without protection, it can be read by any network equipment forwarding these packets. The Internet community has deemed that this potential monitoring is a class of attack [RFC7258]. Metadata confidentiality ensures to application that the metadata used by the transport protocol cannot be read by hosts besides the client and server.

1.4 The Transmission Control Protocol

One of the first transport protocol introduced closely after the Internet Protocol is the Transmission Control Protocol. Again, this protocol is more commonly known under its *TCP* initials. It provides application multiplexing, error detection, connection mode, stream mode and reliable transfer to applications. To achieve these services, TCP introduces a dedicated TCP header which can span up to 60 bytes. This header is illustrated on Figure 1.2 and contains a number of fields that we introduce as we describe how it achieves these services. In the rest of this thesis, we use the word TCP *segment* rather than TCP packet.

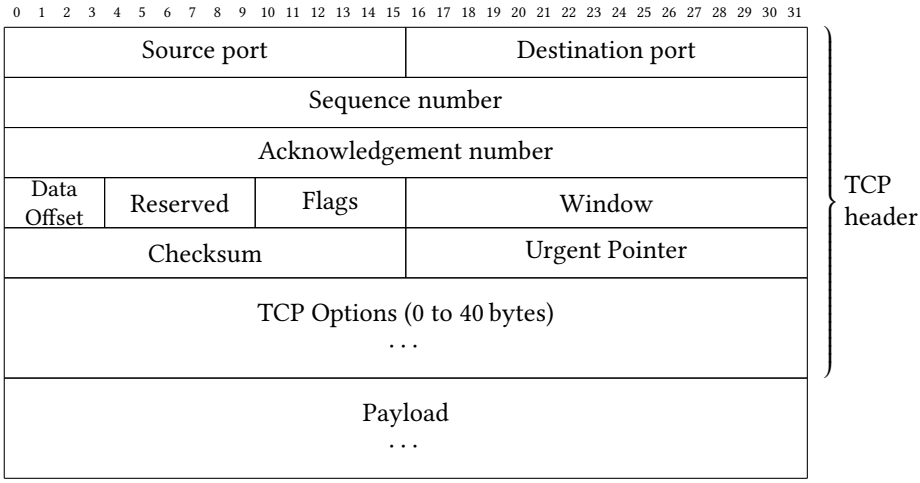


Figure 1.2: The TCP segment format.

The TCP header contains source and destination *port* numbers to distinguish the applications using TCP running on a given host. In practice, the port number is often used to identify a particular server application and client ports tend to have less meaning. For instance, a server that provides email services is expected to accept TCP segments on ports 587, 993 and 995.

To detect a malformed TCP segment due to corruption, a *checksum* field is used. It contains the *ones' complement* of the one's complement sum of the TCP header and payload. It also includes source and destination IP address in the sum. This prevents modifications performed to the TCP segment without recomputing this checksum. However, as a TCP segment is exchanged in plaintext, any equipment that forwards them can possibly alter them and recompute the checksum.

TCP is a connection-oriented protocol and uses a *three-way handshake*, signalled using the *Flags* field, to establish a TCP connection. After the three-way handshake, applications can exchange data. From the client perspective, one round-trip time (RTT) is needed to establish a TCP connection before application data can be sent. A set of source and destination IP addresses and ports uniquely identifies a TCP connection, often referred to as the *4-tuple* of a connection, and all segments received by a host having these fields in common are assumed to be part of the same TCP connection.

To provide its bidirectional bytestream to applications, TCP identifies the offset of each TCP segment within a given direction of this stream using the *Sequence number*. A receiver can then reorder the received segments to form the exchanged data and deliver them in order to the application.

In addition, TCP ensures that this bytestream is transported reliably, i.e., it can indicate to the application when all queued data has been received by the

receiver. This is done using the *Acknowledgement number*, which indicates the next expected sequence number to complete the bytestream in sequence. This field is key for TCP to detect and recover from the loss of IP packets. These mechanisms are detailed in the next subsection.

1.4.1 TCP Selective Acknowledgements

The TCP header can be extended with TCP *Options* that can span up to 40 bytes. This enables extensions to improve and extend the services of TCP by adding new fields within this reserved space of the TCP header. One very common TCP Options is TCP Selective Acknowledgements (SACKs) [RFC2018] which enables a receiver to acknowledge discontinuous blocks of received data. This helps the sender to identify gaps within the received data. However, given the limited size of TCP Options for which other Options might compete, only up to three blocks are typically encoded within SACKs.

1.4.2 Loss recovery

Using the Acknowledgement number and SACKs blocks, a TCP sender can identify the parts of data that are missing to the receiver to complete the transfer. However, the sender should avoid retransmitting data part of packets that are simply delayed or reordered. Discerning whether a packet is lost instead of delayed is not an easy task as a period of ambiguity can be experienced by the sender. Figure 1.3 compares the case of a packet loss to the case of a packet delay and the impact of retransmitting when a SACK block indicates a gap. TCP segments carrying 10 bytes of data are indicated as *Data*, while segments with acknowledgements indicate the Acknowledgement number and the different SACK blocks with *ACK*. In both cases, the sender retransmits as soon as a gap is notified by the receiver. On the left, the retransmission is beneficial as it fills the gap. On the right, it is useless and may waste bandwidth when other data could be sent instead.

Our example highlights another limitation of TCP, which is named the *retransmission ambiguity*. From the sender perspective, when receiving "ACK(30, [])", it cannot distinguish whether the acknowledgement corresponds to the original segment or the retransmitted one, as they bear the same Sequence number. This number only identifies the offset of the segment within the TCP bytestream. As a consequence, a TCP sender may be unable to correctly identify the packet loss pattern and estimate the round-trip time until more acknowledgements clear up the ambiguity. To encode the transmission order of packets, the TCP *Timestamps* Option [RFC7323] can be used. It adds a monotonically increasing timestamp to each packet and can echo to the other peer the last received timestamp in the opposite direction.

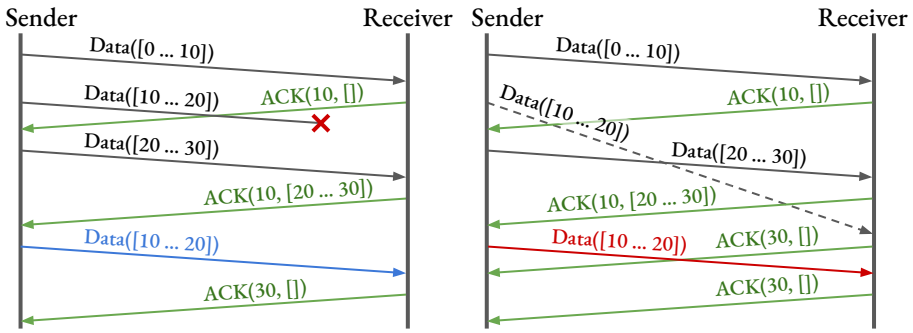


Figure 1.3: Comparison of two kinds of impairments in a TCP transfer. On the left, the second segment is lost. On the right, the second segment is delayed and reordered. Retransmitting is only beneficial in the left example.

Determining when a packet is lost and thus when a retransmission is needed is an important aspect of the performance of TCP. In modern TCP implementations, in addition to the use of SACKs and Timestamps, a TCP sender uses *Retransmission Timeout*, *Recent Acknowledgments* and *Tail Loss Probes*.

Retransmission Timeout (RTO) sets a timer for each sent segment based on the measured round-trip time from TCP acknowledgements. This timer is slightly greater than the round-trip time as it accounts for its variability. However, an absolute minimum value is enforced. The latest specification dating from more than ten years ago recommends a minimum value of 1 second [RFC6298]. Modern implementations, such as in the latest versions of Windows and Linux use 300 and 200 ms [MsRTO; Lin24]. In the latter case, these values are often far greater than the average round-trip time of European users towards popular services. As a result, the RTO is considered as a last-resort mechanism, when a serious blockage occurred and caused an important loss of TCP segments.

However, in cases where the *tail* of a series of TCP segments is lost, i.e., consecutive segments for which no more segments follow, the lack of acknowledgement could lead a TCP sender to wait for the RTO. The two other mechanisms were designed to cope with these cases. Recent Acknowledgements (RACK) proposes a heuristic for interpreting gaps as reported by SACKs which is to consider them as lost when they are continuously reported for more than a quarter of the connection round-trip time. Tail Loss Probes schedules two retransmissions at each round-trip time interval to elicit acknowledgements that can trigger RACK.

1.4.3 Congestion control

While the mechanisms we discussed are able to achieve a reliable transfer when a single TCP connection is traversing a network of networks, they are not sufficient to guarantee the proper functioning of the Internet at scale. First, when sending application data using TCP, a sender must be careful not to overwhelm a receiver with a sending rate that exceeds its capacity. TCP enables a host to advertise its *receive window*, which indicates the amount of data that they can buffer before passing it to the application. A sender can then avoid sending data over the Internet when a receiver cannot buffer it. In addition, the sender is limited by its *sending window*, that keeps track of the application data inflight in TCP segments and that may need to be retransmitted in case of losses.

The second issue relates to the overloading of Internet routers and was first encountered in an event named as the first *Internet congestion collapse* in October 1986. It was caused by several factors. First, when an IP router receives an influx of IP packets that exceeds the capacity of the network link through which they should be forwarded, the IP router discards the packets exceeding the available rate. It may choose to buffer some of them to accommodate for short surges of traffic. However, this is of limited use in the case of persistent congestion as during the Internet congestion collapse. Second, as no mechanism was governing the rate at which TCP should send segments besides the receive window, as long as the demand from applications coincided with the available network capacity, transfers could take place. As the demand from applications increased and exceeded the capacity of some links of the Internet, a cycle of packet loss and abundant retransmissions greatly reduced the effective bandwidths of some links, sometimes nearing three order of magnitudes.

To overcome this problem, several researchers proposed two principles: *congestion control* and *congestion avoidance* [Jac88; JRC98]. The first enables to recover from situations such as the Internet collapse while the other mandates senders to carefully consider the rate at which they are sending to avoid creating these situations. The basis of congestion control is the estimation of a *congestion window*, which corresponds to the given amount of data that can be *inflight* at a time, i.e., that are sent but for which no acknowledgement was received yet. These inflight packets are being forwarded from the sender to the receiver as well as their acknowledgements following a reverse path. An ideal congestion controller perfectly estimates the *bandwidth-delay product* from a sender to a receiver, such that the amount of inflight data corresponds to the available capacity.

These mechanisms are part of *congestion control* algorithms, and since then many controllers have been proposed for TCP to optimize for various

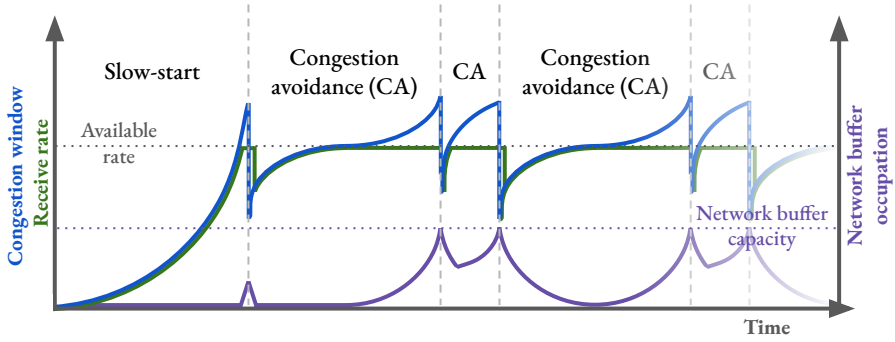


Figure 1.4: CUBIC induces some occupation of the network buffers as it only reacts to packet losses to reconsider its congestion window.

usage patterns and network conditions [BP95; Mas+01; BCV+07; Don+15; Don+18; Car+16; AB18]. We focus on two popular controllers, both representing a different approach to congestion control, and that were estimated to be used by nearly half of the popular websites [Mis+19].

1.4.3.1 CUBIC

The first is named CUBIC and is a *packet-loss* based controller. It makes the fundamental assumption that medium-related losses, e.g., interference and packet corruption, are very limited. Instead, packet loss is assumed to be largely caused by congested IP routers discarding them, and thus indicative of network congestion. Figure 1.4 illustrates the different phases of a CUBIC flow over time. The sending throughput is depicted in blue while the receive rate, as computed from the acknowledgement sent by the receiver, is depicted in green. In addition, the occupation of network buffers is illustrated in purple. The vertical dashed lines highlight the different phases of the algorithm.

The first phase of a CUBIC flow is called the *Slow-start*, in which it starts from a minimal congestion window that increases exponentially. As the network bandwidth that could be available spans more than 10 order of magnitudes, this enables both to carefully avoid overloading the network early in the phase while finding quickly a congestion window that fits the available rate. CUBIC exits the slow-start whenever a packet loss is reported or that an important delay fluctuation has been measured. This version of slow-start is known as *HyStart++* [RFC9406]. It is the only time that CUBIC considers round-trip time variations as congestion signals. This modification gives a better first estimate of the congestion window and reduces the occupation of network buffers when exiting the slow-start.

The second phase that follows the slow-start is the *Congestion Avoidance (CA)* phase. Its behaviour during this phase gave its name to the algorithm as

CUBIC uses a cubic function to compute the congestion window over time. It enables to reach both its inflection point quickly, and then probe carefully larger congestion windows. In addition, the inflection point is always chosen to be equal or lower to a previous congestion window value. In our example, the first CA phase ends when probing larger congestion windows eventually filled all the network buffers with extra packets exceeding the available rate. At this point, packet losses occur as routers are saturated. This triggers a new CA phase starting with a 30 %-lower congestion window and an inflection point close to the last congestion window before the packet losses happened. As the network buffers drain and fill up again, more packet losses are experienced before reaching the inflection point, such that the next CA phase lowers its inflection point.

When looking at the green line which represents the receive rate of data, i.e. the effective rate that the receiver obtains, we can observe that CUBIC offers a relatively high rate but at the cost of an oscillation in the occupation of the network buffers. When packet losses unrelated to congestion are experienced, e.g. due to wireless links, CUBIC deteriorates the throughput quickly. As more heterogeneous Internet access technologies got deployed over the years, the initial assumption of CUBIC regarding the nature of packet losses became weaker.

1.4.3.2 BBR

The second controller is BBR and is a *rate* based controller starting from a different assumption. When CUBIC observes congestion-induced losses, it is likely to have filled the buffers of IP routers on its path. Prior to this signal, an increase in delay as a result of the packet queuing through these buffers can be observed. BBR tries to avoid filling these buffers by measuring the fluctuation of the round-trip time. When data is sent at a steady rate and the delay increases, BBR reduces the sending rate as it considers that the available bandwidth has been exceeded. In this case, packet losses are assumed to be more likely caused by the medium, and BBR allows for ample retransmissions fitting the estimated sending rate as long as the loss rate is inferior to 2 %. BBR uses packet *pacing* to control its sending rate, which spaces the packets over time with a given interval. This reduces the pressure on network buffers as packet arriving at routers may not need to be queued first and wait for a forwarding opportunity.

BBR is developed by Google and since 2016 three versions are known. The first is part of the Linux kernel since version 4.9, but BBRv2 and BBRv3 are still off-tree patches. This subsection describes the latest version. BBRv1 is estimated to be used on 40 % of the Internet downstream traffic volume [Mis+19]. Researchers have studied its coexistence with CUBIC and found that BBR can

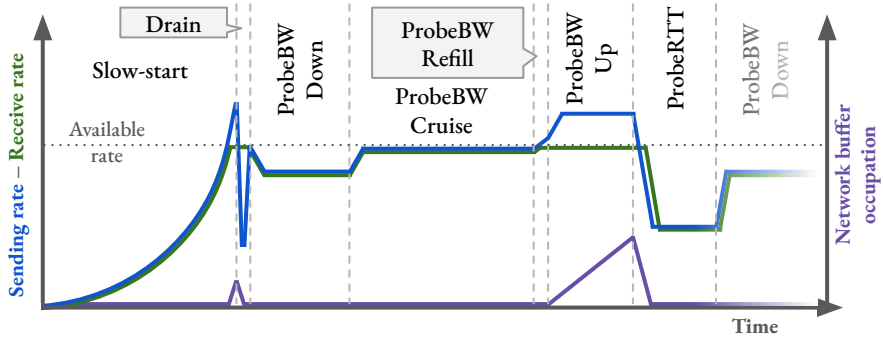


Figure 1.5: BBR tracks the receive rate to compute its sending throughput with several sending phases while minimising the occupation of network buffers.

be unfair to other CUBIC flows [War+19]. Some applied game theory to posit the existence of a Nash equilibrium between the number of CUBIC and BBR flows, such that the incentives to choose BBR over CUBIC as a congestion controller are decreasing with the number of BBR flows forwarded through a path [MTL22].

BBR comprises more phases than CUBIC but shares the same HyStart++ slow-start. After the slow-start, it enters a short and one-time *Drain* phase after the slow-start to release the network buffers occupation that may have been created. Then, it alternates between the *ProbeBW* and *ProbeRTT* phases. The former is composed of four steps and computes the adequate sending rate while the latter observes the minimum round-trip time such that the sending rate can be converted to a congestion window. BBR spends most of its time in the ProbeBW phase, only occasionally transitioning to the ProbeRTT.

The ProbeBW phase repeats four steps: Down, Cruise, Refill et Up. During the *Down* step, BBR reduces the sending rate to 90 % of estimated available bandwidth to reduce the amount of data inflight and potentially occupying network buffers. When done, it switches to the *Cruise* step during which it sends at 100 %. BBR stays in this state until roughly 62 round-trips or 2 to 3 seconds passed. This value was chosen as a tradeoff between bandwidth exploration and the potential packet losses it can cause and induce on other flows as well. After the Cruise step, BBR transitions to a short *Refill* step that aims at fully utilising the available bandwidth for one round-trip time before probing higher bandwidths in the next step. The last step before cycling again through the others is the *Up* step, during which BBR sends at 125 % of the estimated available bandwidth. If a receive rate increase is observed, then the estimated available bandwidth is increased as well. In our example, as BBR reached the optimal bandwidth, the receive rate does not increase and the additional packets introduced fill some of the network buffers.

When the application is constantly sending such that the bandwidth is fairly utilised at all time, BBR transitions every 5 seconds to the *ProbeRTT* phase in which the sending rate is reduced to 50 % such that the minimum round-trip time value can be updated with samples taken with a low bandwidth utilisation. When the application has silence periods, i.e., when it does not queue application data for sending, BBR is able to refresh its minimum round-trip time observations and does not trigger the ProbeRTT phase. This makes BBR sacrifice 2 % of the available bandwidth in the worst case only.

1.4.3.3 Fairness

When estimating the congestion window, a desirable property for congestion controllers is to *share* the available bandwidth with other congestion-controlled connections, often named *flows* in this context. This principle is called *fairness* and is key to the design of congestion control algorithms. It establishes that each flow should obtain a fair share of the available congestion window [Jac88]. As a consequence, fairness is established in terms of throughput and is often evaluated with models or experiments with flows that send indefinite amounts of data for a given duration. Such flows only represent a particular and minor case of usage of the Internet. As such, researchers have criticised a rate-based definition of fairness and proposed alternatives definitions such as flow completion time [DM06; Bri07; ZK23]. However, flow-rate fairness remains the status quo.

1.4.4 TCP Head-of-Line blocking

In addition to the bidirectional bytestream that TCP offers to applications, it also guarantees that application data is delivered in order. This can simplify the design of applications as they do not have to deal with reordering the received data. However, a strict constraint stems from this choice as well. When a part of the bytestream is not received, the data delivery to the application has to stop as soon as this part is reached. In circumstances where recovering from the loss becomes complicated, e.g., as retransmissions get lost as well, this prolongs the pause and can potentially halt the application for some time.

This problem is known as the TCP *Head-of-Line (HoL) blocking*. Note that this problem can be exacerbated by the congestion controller which might constrain the sending rate as a result of the loss, multiplying the HoL blocking impact.

1.4.5 Extending TCP

Two limitations arise from the design of the TCP header. First, the space for TCP Options is limited, and with the numerous TCP Options that exist to-

day [RFC2018; RFC7323; RFC8684; RFC5925], TCP applications are limited in the set of Options they can use. Some Options require being present in the first TCP segment of a connection which further limits how many of them can be used together. For instance, common Options such as the `MSS`, `Window Scale`, `SACK Permitted` and `Timestamp` Options consume 20 bytes together, i.e., half of the available Option space. More complex extensions such as `AO` [RFC5925] and `MPTCP` [RFC8684] consume each 16 additional bytes.

Second, as with the rest of the TCP header, the Options are not confidential nor authenticated. This means that IP routers and other network equipment, such as *firewalls*, can restrict the use of these Options. Even though the Internet Protocol mandates IP routers with the role of forwarding valid IP packets, regardless of what transport protocols are used, a number of network equipment manufacturers designed equipment failing this requirement in several manners. These equipments are named *middleboxes* when they are interacting with TCP segments instead of simply forwarding them. Middleboxes often "support" a limited set of Options, i.e., they do not interfere with them. This set often stems from the set of Options that the designers of a middlebox were aware of or that existed during its design phase. Researchers have found middleboxes to drop TCP segments containing Options outside their set, to modify TCP segments and alter the values of Options, to temporarily block IP addresses sending TCP segments with certain Options and to blindly echo TCP Options. There exist several research works documenting the practical constraints of extending TCP in the recent years when facing these behaviours [Hon+11; Rai+12].

While several solutions have been proposed to increase the number of TCP Options [TE24; Ram12], all are facing significant implementation and deployment issues. As a result, TCP is considered almost not extendable any more today.

1.5 The User Datagram Protocol

The *User Datagram Protocol (UDP)* is a transport protocol designed in 1980 to offer a different set of transport services than the one of TCP to applications [RFC768]. UDP is a *connection-less*, *unreliable* protocol that offers a *message* service. It thus does not establish explicit transport connections and does not guarantee the effective delivery of the messages it sent. It also offers application multiplexing with ports number similar to TCP. Error detection is optional and can be enabled with a checksum similar to the one used by TCP.

Figure 1.6 illustrates the format of a UDP packet. The *Payload* contains a single message that is sent by a UDP sender towards a UDP receiver. As a UDP packet fits inside an IP packet, the maximum length for an application message is constrained by the maximum length of IP packets that can be

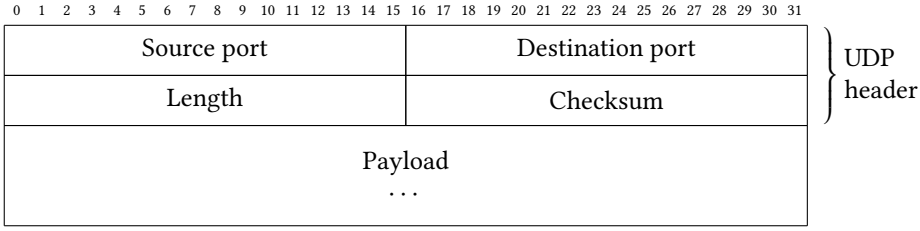


Figure 1.6: The UDP packet format

conveyed between a sender and receiver. The *Length* field indicates the total length of the UDP header and payload.

UDP does not include mechanisms for congestion control and applications that use UDP should ensure that their use of UDP is fair and does not induce congestion in the network. While UDP does not establish connections, UDP flows are discerned by their set of source and destination IP addresses and ports.

1.6 The Domain Name System

When an application uses a transport protocol such as TCP or UDP, it indicates the computer it is willing to reach using its IP address and a known port to identify the application that should receive its packet on this computer. As IP addresses identify network interfaces of computers, it is a non-trivial task to identify which IP addresses belong to computers offering the service required by an application. This problem is common to many applications and lead to the development of a dedicated protocol called the *Domain Name System (DNS)* to solve it.

The DNS establishes a *hierarchical* and *distributed* naming system for computers connected to the Internet. The names it provides could be compared with the addresses of the postal service, where a postal address has several levels, e.g., continent, country, city, etc, and in which the possible values for each level are defined by different entities. However, it also offers a *resolution* service, akin to a phone book, as it can map a name to an IP address.

This mapping process is illustrated on Figure 1.7, in which the domain name `ncs.uclouvain.be` is resolved to an IP address. This process involves several parties as the domain is composed of three parts. In our example, the name belongs to the `uclouvain.be` domain name, which in turns belongs to the `.be` *top-level* domain name (TLD).

Interacting with the DNS takes the form of queries. A query is composed of a domain name and a record type, i.e., a type of information that is requested for the given domain name.

First, the client is requesting a resolver to resolve the domain name to an

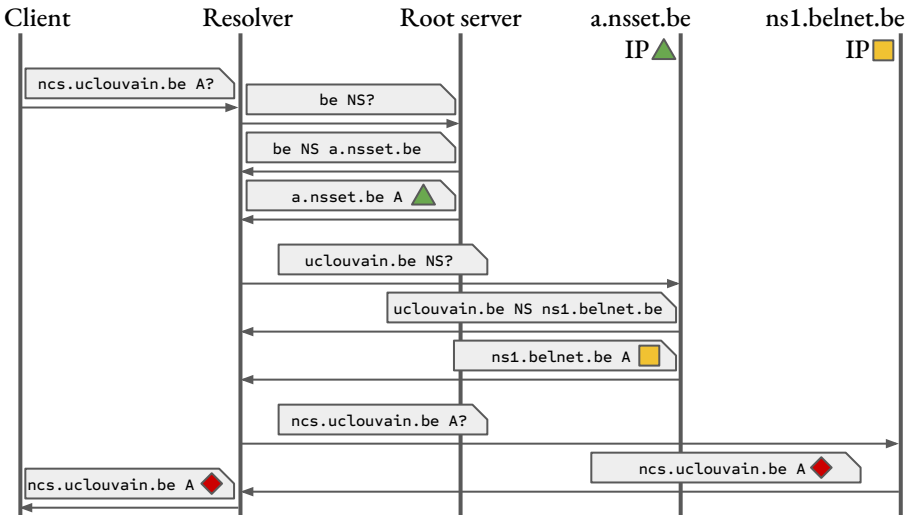


Figure 1.7: An example of recursive resolution of the `ncs.uclouvain.be` domain name

IP address using the A record type. The resolver is in charge of querying the DNS in order to obtain this information. It can be local to a client, e.g., a background software on that machine, or running on the access gateway in the client network. The resolver often has a cache such that frequent queries can be served directly from the cache during the validity period of their answers.

When receiving the client query, the resolver breaks down the requested domain name into the different hierarchical levels. To resolve the requested domain name, the resolver starts by querying a *root* server for the authoritative name server, using the NS record type, of the `.be` domain name. A root server handles all the TLD names. It receives the domain name and an IP for this authoritative name server. Then, it queries the authoritative name server for the `uclouvain.be` domain name. Finally, it can query this final authoritative server to obtain an IP for the `ncs.uclouvain.be`. The client then receives the resolved IP address.

There exists more record types that can be queried from the DNS, and for a given record query, several answers can be provided, e.g., multiple IP addresses for a given domain name.

DNS queries and responses were first conveyed over UDP, as they are small and time-sensitive. When an application uses the DNS to resolve a domain name, it is often a first blocking interaction with the Internet before establishing transport connections. Using UDP, a query can be sent at any time. DNS client and resolvers embed timers for detecting lost queries and can retransmit them when needed. As these timers are often in the order of seconds and the exchanged data size is low, the DNS exempts itself from ex-

ercising congestion control. Several other transports were proposed over the years for the DNS to improve its reliability and provide better security, e.g., confidentiality, when querying the DNS, such as DNS over TLS [RFC7858], DNS over HTTPS [RFC8484] and DNS over QUIC [RFC9250].

1.7 HTTP

The Internet architecture is often described as having a *narrow waist* due to the ubiquity of the Internet Protocol. IP can be layered above several *link-layer* protocols, e.g., Ethernet, Wi-Fi and 5G, and is used by several transport protocols. In the recent years, a *neck* has been taking shape in the Internet architecture with the increasing use of the Hypertext Transfer Protocol (HTTP). HTTP is an *application* protocol that is layered on top of transport protocols. While the HTTP protocol is mainly known for its presence within web browsers as the foundation for the Web, a very important number of applications are using HTTP in different contexts outside of web browsers. For instance, mobile applications that embed connected features are very likely to use HTTP to realise their services.

HTTP was introduced in 1990 as a protocol that uses TCP to transfer hypertext document that are then processed and rendered by web browsers. It is a *request-response* protocol, in which the client issues a request to access a given document and the server responds by sending the document and its associated metadata. HTTP requests are composed of three elements: *method*, *headers* and *body*. Responses also have the two last elements in addition to a *status code*. The method, status code and headers are encoded using the ASCII encoding. The encoding of the body can take many forms and is described within the headers.

Figure 1.8 illustrates a request and response towards the *example.com* website using the first version of HTTP. A preliminary step is to obtain an IP address for a server that hosts this website using the DNS as described in Section 1.6. As it is layered over TCP, a connection to this server is established first. The client and server use the TCP connection to write HTTP messages using the ASCII text encoding.

First, the client sends the HTTP request. In our example, it is a `GET` request towards the root of this website, i.e., `/`. The method is followed by headers. In our case, the `Host` header is used to indicate the desired website for this request. A `GET` request has no body, so the end of the headers marks the end of the request.

The server responds with a status code of 200 indicating that the request was successful and that the document is part of the response body. Then, four headers describe metadata on the document. Its file type and encoding is described using the `Content-Type` header while `Expires` and `Last-`

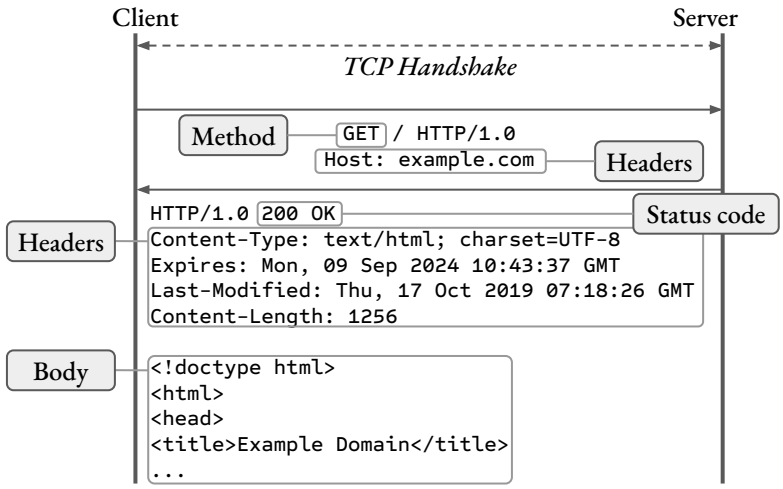


Figure 1.8: Example of an HTTP/1.0 request and response.

`Modified` can be used by web browsers to put the document into a cache during its validity period.

1.7.1 Improving HTTP

Over the years, HTTP has been improved in several ways. HTTP/1.1 introduced a small set of changes that enabled to reuse a TCP connection for several requests by simply writing them on the TCP bytestream one after the other. This spares a client from establishing TCP connections for subsequent requests but also improved congestion control as short-lived TCP connection spend an important part of their lifetime in the slow-start, which typically has a low usage of the available bandwidth.

HTTP/2 introduced more important changes for better performance [RFC9113]. First, it proposes a binary encoding for the headers, which can greatly reduce its size for faster transfers. Second, it enables to *multiplex* requests such that they can progress in parallel over a TCP connection. To achieve these features, HTTP/2 introduces *frames* as a basic protocol unit. In previous versions of HTTP, the TCP bytestream consisted of a series of requests from the client and responses from the server, while in HTTP/2 a series of frames is exchanged in each direction.

Using frames, a client can send several requests to be processed in parallel. The server chunks each response in a series of frames and alternate them over the TCP bytestream. The server is free to choose which response should be delivered first and how they should interleave. To discern which request each frame refers to, HTTP/2 introduces the concept of *streams*. Each stream orders a set frames into a single request and response. In addition to the

frames of a stream, called DATA frames, HTTP/2 introduces a number of *control* frames to manage the connection. These include the SETTINGS frame to exchange configuration parameters for the connection and the RST_STREAM frame to request the immediate termination of a given stream.

Given that TCP is vulnerable to Head-of-Line blocking as explained in Section 1.4.4, the multiplexing achieved by HTTP/2 can also be impacted by this issue. For instance, the loss of a TCP segment containing data from a given HTTP request blocks the delivery of subsequent requests on the TCP bytestream until the loss is recovered. As such, HTTP/2 streams enable interleaving of HTTP responses such that they can progress at a chosen rate, but it does not enable true concurrency between two requests and responses as they are serialised on a single bidirectional bytestream.

1.8 Securing Internet communications with TLS

When IP and TCP were designed, a need for reliable communications existed but it did not imply *security*. Together, these protocols can guarantee that the application data has been delivered, with bounds on the class of errors that can be detected on an IP packet or TCP segment. These bounds were designed to cope with medium failures such as electro-magnetic interferences but not with an *attacker*, i.e., an ill-intentioned host that might read and send IP packets. As such, they do not ensure *confidentiality*, which prevents third-party from reading the exchanged data, *integrity*, which prevents third-party from modifying the data exchanged, and *authentication*, which prevents third-party from sending or receiving application data by impersonating a host. In the rest of this thesis, we refer to these three properties at once using the terms *protection* and *protected*.

The Transport Layer Security protocol was designed to secure the bytestream offered by TCP and provide these security properties to applications. Its most renowned use is within the HTTPS variant of HTTP such that the HTTP messages are exchanged securely over a TCP connection. This is a very important part of many use-cases of the Internet, e.g., online banking or payments, which would otherwise exchange their data in plaintext and be vulnerable to attackers. To provide confidentiality, integrity and authenticity, TLS uses encryption. In addition, TLS can verify the identity of both peers using certificates. A certificate contains a *public* key that can be used to encrypt data to be decrypted by a secret *private* key held by the certificate owner. Certificate are established in *chains*. By trusting a restricted number of *root* certification authorities, a host can verify that a certificate is issued using a legitimate chain of certificates linked to one of these roots. It does so by verifying the *signatures* that authenticate the issuer of each certificate within the chain.

TLS 1.3 comprises two main phases, the handshake and the traffic phase.

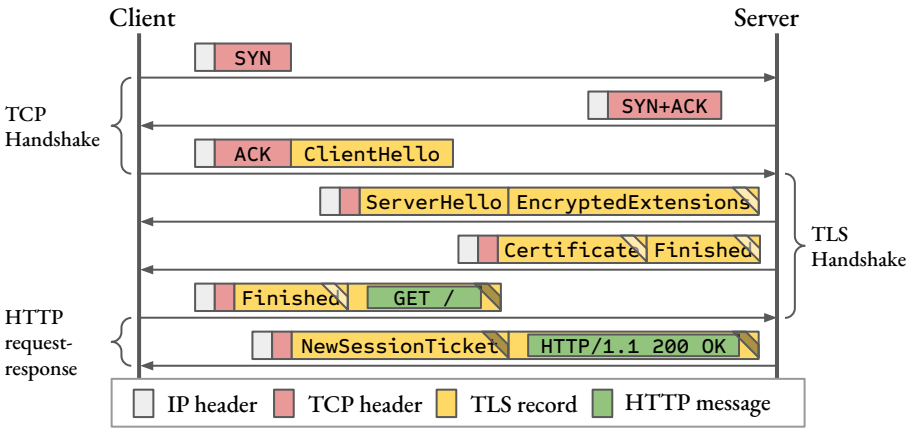


Figure 1.9: TLS is layered atop the TCP bytestream and can secure the exchange of HTTP messages.

During the handshake phase, the client and the server negotiate the cryptographic algorithms that are used to protect the application data and the method to establish a shared secret privately that will be used with these algorithms. When it completes, the TLS session is established and enters the traffic phase where application data can be protected.

The basic protocol unit of TLS are TLS records. These are *Type-Length-Value* units that are used to exchange control information to establish the cryptographic algorithms that are used to segment and to protect the application data. TLS records have a maximum size of 16 384 bytes and thus can span several TCP segments to be conveyed in their entirety. Figure 1.9 illustrates how a client uses TLS 1.3 [RFC8446] to secure the bytestream offered by TCP and exchange HTTP/1.1 messages inside protected TLS records. The Figure breaks down the stack of protocols used, indicating important values and abstracting them when needed.

First, the client initiates the TCP handshake with a SYN TCP segment for which the server agrees with a SYN+ACK segment. This completes the handshake from the client perspective which then sends a ClientHello TLS record to announce the cryptographic algorithms it supports. The client can announce extensions to the TLS protocol inside a dedicated space inside this TLS message. On the contrary to TCP, this space is much less constrained in size. However, this record is written in plaintext on the TCP bytestream.

The server advances the TLS handshake by sending four TLS records. ServerHello announces the server parameters and selects a set of cryptographic algorithms from the ones announced by the client. It also takes places in plaintext. At this point, both client and server can securely derivate keying material to secure the rest of the TLS handshake based on the ex-

changed parameters. This material is used to protect the three remaining TLS records sent by the server as indicated by the clear white strip, such that it can announce TLS extensions (EncryptedExtensions) and present its certificate securely (Certificate). To ensure that the ClientHello and ServerHello were not modified when exchanged over the TCP connection, the server computes a *hash* of the exchanged TLS records and sends it in the Finished record. The client recomputes the expected value and compare it to the received value. When done, the client also sends a Finished record to confirm that it effectively received all records sent by the server. After a host has received its peer Finished message and sent its own, it can use the final secure keying material, as indicated by the clear black strip, to protect AppData records carrying application data. In our example, these contain HTTP/1.1 messages such that a request and a response are exchanged. But TLS can be used with many other application protocols [RFC2595; RFC8689; RFC7858; RFC4513].

1.8.1 Session resumption, Pre-Shared Key and 0-RTT data

TLS 1.3 also enables a client to resume the keying material of a previous TLS session instead of performing a full handshake. This is less costly in terms of computations but requires that a session with the TLS server was established previously and that a *Pre-Shared Key (PSK)* was exchanged. This key can also be used to secure AppData records such that they can be sent along with the connection resumption request inside the ClientHello record. These AppData records, referred to as 0-RTT data or *early data*, bear different security properties than records secured after a full handshake. First, a compromised PSK gives access to the plaintext of these records, which are then said to be not *forward secret*. Second, these records can be replayed between connections as they only depend on the PSK and are not tied to a particular connection. For instance, an attacker could store their ciphertext and may reorder, duplicate and re-inject them. Despite these degraded properties, they remain confidential, integrity-protected and authenticated, and are useful when the application messages they contain are *idempotent*, i.e., they can be applied multiple times and retain the same effect as a single application, such that an attacker replaying these messages would not compromise a given system.

1.8.2 Authenticated Encryption with Associated Data

The confidentiality, integrity and authentication of TLS records are guaranteed with the use of *Authenticated Encryption with Associated Data* (AEAD) algorithms [BT16; LP24; RFC5116]. These algorithms can perform authenticated encryption and decryption. The encryption operation requires four inputs, (*i*) a secret key, (*ii*) a *nonce*, i.e., a unique value for each encryption

operation[1], (*iii*) the plaintext to encrypt and authenticate and (*iv*) the associated data to authenticate. It produces in result a *ciphertext* at least as long as the plaintext, such that *authentication data* may be appended in the ciphertext. Each TLS record is protected using an encryption operation. When received, it is processed with a decryption operation.

When used in TLS, the key and the Initial Vector (IV), i.e., the basis value for the nonce to which a counter may be added to form a unique value, are derived from the shared secret established by the TLS handshake.

The associated data is useful to protect non-confidential information, such as fields that must remain in plaintext and that are required to handle plaintexts or ciphertexts. When protecting a TLS record, its plaintext header, which indicates its length, is part of the associated data.

The decryption operation requires four inputs, which are the key, the nonce, associated data and ciphertext as defined in the encryption operation. In result, it produces the plaintext when the inputs are *authentic*, i.e., they correspond to the encryption operation without modification. AEAD algorithms can detect when these inputs are not legitimate with a very high probability.

In a TLS session, nonces are formed from a fixed value derived from the shared secret and include a counter for every TLS record sent. As TCP delivers data in order, a receiver can simply count the TLS records it decrypted to form the nonce of the next TLS record to decrypt.

The uniqueness of the nonce is a critical assumption of AEAD algorithms. When it is not guaranteed such that two different plaintexts are encrypted using the same key and nonce, the resulting ciphertexts are very weak and can reveal their plaintext. The nonce uniqueness also guarantees that the produced ciphertexts cannot be *replayed*, e.g. an encrypted TLS record cannot be sent twice by an attacker.

1.8.3 Key Update

AEAD algorithms have limits on the amount of ciphertext they can produce and process with a given key before the security properties they ensure can fail with known probabilities. Depending on the particular algorithm, these can be as low as $2^{21.5}$ packets with a probability of 2^{-57} [RFC8446]. As TLS uses a 64-bit record sequence, these limits can be reached before the end of a TLS session. To mitigate this problem, TLS 1.3 includes a *KeyUpdate* message to announce the use of a new key derived from the shared secrets established in the handshake. This process can be repeated when needed.

[1]Nonces are often formed with a fixed part in their higher bits and a counter in their lower bits.

1.8.4 TLS Exporter

In addition to establishing a shared secret to protect the TCP bytestream, TLS can be used by applications to create derivate secrets for other uses. Applications can then use these secrets with other cryptographic constructs in other parts of their functioning. To this end, TLS exposes an *Exporter* that leverages the `HKDF-Expand-Label` function to create derivate secrets given a label and context value. A client and server that invoke this function with equal parameters produce the same secret. As such, the Exporter can also be used by a host participating in the TLS session to ensure through its application protocol that its peer has access to the TLS session.

1.9 Multiple paths in IP networks

We have briefly discussed the Internet Protocol in Section 1.2 and detail in this section more advanced concepts, problems and solutions related to IP networks.

1.9.1 Versions of the Internet Protocol

Two versions of the Internet Protocol co-exist on the Internet today. The first, IP version 4 is the first widely used Internet Protocol. It encodes IP addresses using 32 bits, such that 4.3 billions of IP addresses can be defined. As the Internet grew in size, the exhaustion of this address space became apparent and led to the design of the version 6 of the Internet Protocol. While IP version 6 introduces a number of changes in its semantics and the way it can be provisioned inside a network, its biggest difference is the encoding of IP addresses using 128 bits. This pushes the number of available addresses to 3.4×10^{38}. In the rest of this thesis, we use the term *address family* to designate either the set of IPv4 or IPv6 addresses, such that IPv4 (resp. IPv6) can be said to define an address family.

IPv4 addresses are represented as an octet tuple in decimal form, such as `130.104.1.1`, where each octet is separated by a dot. Given the larger address space of IPv6, its representation is also more complex. IPv6 addresses are represented as a 2-octet tuple separated by a colon, such as `2001:6a8:3080::`. The 2-octet parts that equal zero can be left out and represented by a single colon immediately succeeding the previous colon.

Although IPv6 was designed as a successor and replacement to IPv4, both versions of the protocol are used on the Internet today as totally abandoning IPv4 remains a challenge. Most of the popular web services are supporting IPv6 today, but there remains a long tail of networks and services which do not support IPv6 [Soc24]. User devices, such as smartphones, laptop and computers generally include support for IPv6, often for more than 10 years. There

remain a set of legacy devices which only support IPv4 and for which manufacturers abandoned updating their software. For these devices, technologies have been designed such that they can still access an IPv6-only Internet with IP-translation services and devices [RFC2473].

Following the design of IPv6, the DNS was updated with a new record type such that domain names can be resolved to IPv6 addresses as well. IPv4 addresses uses the A record type and a new AAAA record type was introduced for IPv6 addresses. Applications were modified to send queries for both types of addresses when they support IPv6 such that they can fall back to IPv4 when the domain does not support IPv6. However, they had to include the required logic for sending parallel queries and design heuristics on how to wait and use their answers.

1.9.2 Default Address Selection

Shortly after IPv6 was introduced, as it became apparent that devices entered a transition period where they would support both IPv4 and IPv6, they faced the choice of selecting the source address to establish transport connections. In addition, as mentioned when describing the DNS in Section 1.6, when resolving a domain name, multiple addresses can be obtained, even for the same IP version. This broadens the problem to choosing a pair of source and destination addresses.

The IETF addressed this problem with the *Default Address Selection* algorithm [RFC6724]. It defines a set of rules to establish the preferred source address for each destination address obtained from the DNS. The preferred source address is decided based on a number of criteria. For instance, preferring same-scope addresses, e.g., a local source address for a destination within the local network and preferring privacy-preserving source addresses.

These sets of rules work well when the destination is in one of the home networks of the host. In this case, the source address corresponding to the network interface connecting the host to this network is selected. This choice is considered appropriate as it is likely to form the shortest path to the destination.

However, in the general Internet, the last rule, acting as a tie-breaker, which prefers the longest matching prefix, is often used. When the source and destination addresses are part of entirely different prefixes, the outcome of the algorithm has no guarantee on the quality of the resulting network path. A host can also compare source addresses based on the history of round-trip times from connections established using this address, but this is purely optional and documentation on its implementation within operating systems is lacking.

Then, the host sorts the set of destination addresses considering their

corresponding preferred source address. This list is used to establish the sequence of destination and source addresses that will be tried by the host. The list is established with similar criteria as used for the source selection.

When an address from the sorted list is tried, the application has to choose the time it is ready to spend waiting for a connection to be established before considering another address from the list. An application could also choose to race the n best addresses from the list to then use the connection that is established first, but the Default Address Selection algorithm does not provide any recommendation in this regard.

1.9.3 Happy Eyeballs

When an application uses the Default Address Selection algorithm as defined in Section 1.9.2 and initiates a transport connection, in the case where the connection silently fails, i.e., no packets are received to indicate a failure, the host running the application has to set a timer to eventually abandon the connection establishment. When the timer expires, another address from the list established by the Default Address Selection algorithm is tried. The magnitude for connection establishment timeout is often in the order of tens of seconds, which severely degrades the user experience.

The approach taken by the IETF to solve this problem is to provide to applications an address selection mechanism favouring IPv6 that could quickly fall back in case of unexpected connection delay. The algorithm is called Happy Eyeballs and was first standardized in RFC 6555 [RFC6555], and then revised in RFC 8305 [RFC8305].

Figure 1.10 illustrates the state machine of the Happy Eyeballs Version 2 algorithm as described in RFC 8305. The algorithm introduces several choices. IPv6 is favoured when choosing the first address to establish a transport connection. When the DNS resolution of IPv6 is delayed compared to IPv4 (state 3), the algorithm recommends waiting 50 ms before trying the IPv4 address. When an empty AAAA DNS record is received, IPv4 is tried as soon as possible ($5 \longrightarrow 6$). When an address is tried (state 6), the algorithm specifies three time bounds before trying another address from another family. The next attempt delay must not be lower than 10 ms, with a recommended value of 100 ms and a maximal recommended value of 2 s.

A third version of the Happy Eyeballs algorithm has been proposed [Pau+24]. It adds support for the DNS SVCB record type. It can be used to advertise information on the services provided by a given domain name. For instance, it can advertise the application protocol supported on a given port. It can also contain hints on IPv4 and IPv6 addresses for a given domain name. These are useful to obtain these addresses in a single query as an SVCB record can also contain an alias, which would otherwise require additional DNS queries to

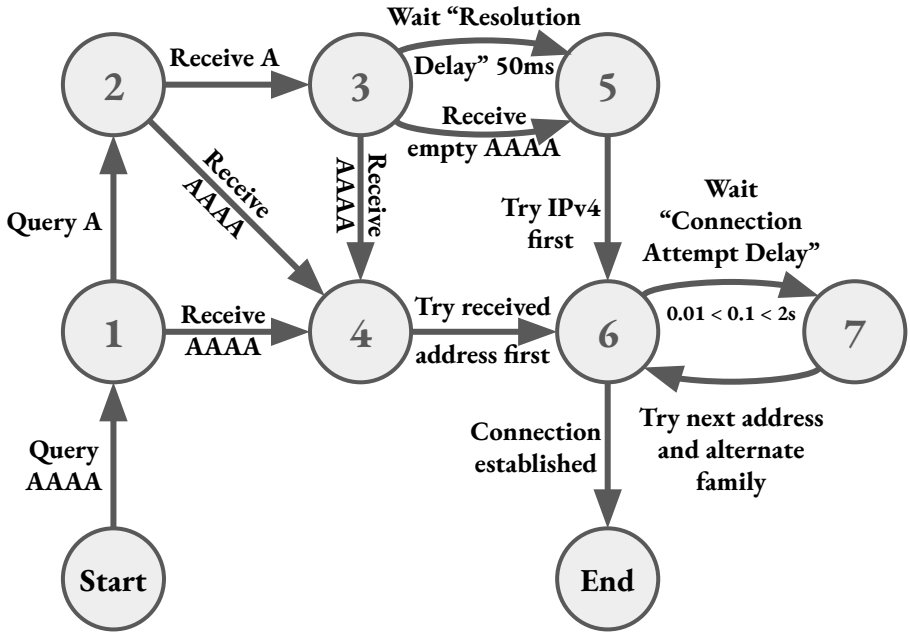


Figure 1.10: Happy Eyeballs Version 2 as defined in RFC 8305.

resolve it.

1.10 The QUIC protocol

The QUIC protocol is the most recently standardised transport protocol considered in this thesis. It is almost entirely contemporary to the works presented in this thesis. Its set of specifications [RFC8999; RFC9000; RFC9001; RFC9002] constitutes a significant effort within the IETF. HTTP is the first application that was adapted to use QUIC with the new HTTP/3 [RFC9114]. QUIC has then been adopted by a majority of the large web services provider through the deployment of HTTP/3, with 30 % of the encrypted Internet traffic estimated to be transported by QUIC [Clo24], and by user devices by being directly embedded into applications and web browsers. Researchers measured that 80 % of user devices in Europe could connect using HTTP/3 and QUIC [APN24].

QUIC introduces a number of important changes but keeps the *end-to-end* principle at its core. This paradigm decouples the network layer from the transport layer, which is the responsibility of hosts connecting to the Internet. IP, TCP and UDP were designed according to this principle. QUIC greatly strengthens the end-to-end principle by extensively using encryption to protect most of its packets. This contrasts with TCP and TLS that only protect

the application data within the TCP bytestream. This is key to increase the privacy and reduce the possible interference from middleboxes against QUIC. As a result, the QUIC protocol can be extended more effectively than TCP.

In terms of transport services, QUIC provides equivalent services to TCP by including application multiplexing, error detection, connection mode, stream mode and reliable transfers. In addition, QUIC includes a message mode, stream multiplexing, connection migration and metadata confidentiality. QUIC also integrates TLS within its handshake such that application data is always encrypted.

On the contrary to TCP and UDP, QUIC is not directly layered above IP and reuses UDP. This has several advantages. First, the middlebox interference that was discussed for TCP in Section 1.4.5 is also impacting IP such that introducing a new transport protocol above IP risks its packets being altered or filtered. UDP is also accessible to applications which enables QUIC to be developed as a library directly embedded within applications. This simplifies both its development and deployment. Other transport protocols such as TCP and UDP are often implemented inside operating systems, such that improving and updating them is more complex and time-consuming. UDP also contributes to the application multiplexing and error detection services of QUIC. They are complemented with other mechanisms that are described in the rest of this section.

1.10.1 Connection IDs

QUIC decouples its connections from the set of IP addresses and UDP ports (4-tuple) that are used to exchange QUIC packets and identifies them using *Connection IDs (CIDs)*. These are octet strings that can span up to 20 bytes. Each host chooses a sequence of Connection IDs that they communicate to their peer. When sending QUIC packets, a host uses a Connection ID from the sequence provided by their peer. Having a sequence rather than a unique Connection ID enables a host to change it periodically to break the linkability of its packets by an on-path observer. In addition, in the event of a change within the 4-tuple used on a QUIC connection, it can continue given this separate identifier. This is an improvement compared to TCP for which connections are terminated in such cases. It is then up to their applications to include the required logic to overcome these events.

These 4-tuple changes can happen passively for QUIC clients and actively for QUIC clients and servers. The active case is discussed in Section 1.10.7. The UDP port used by the client as seen by the server can change after periods of inactivity when the client connects to the Internet using an IP router that performs *Network Address Translation (NAT)*. Such a NAT maps IP packets using private IP addresses to public addresses and may change the source port

within the transport-layer headers of these packets. NATs are often found in IPv4 deployments today to cope with the exhaustion of IPv4 addresses. As UDP does not have a notion of connection that NAT can observe, the *binding* may be freed and renewed after a period of inactivity, causing a change in the UDP source port from packets sent by the client to the server [Hät+10].

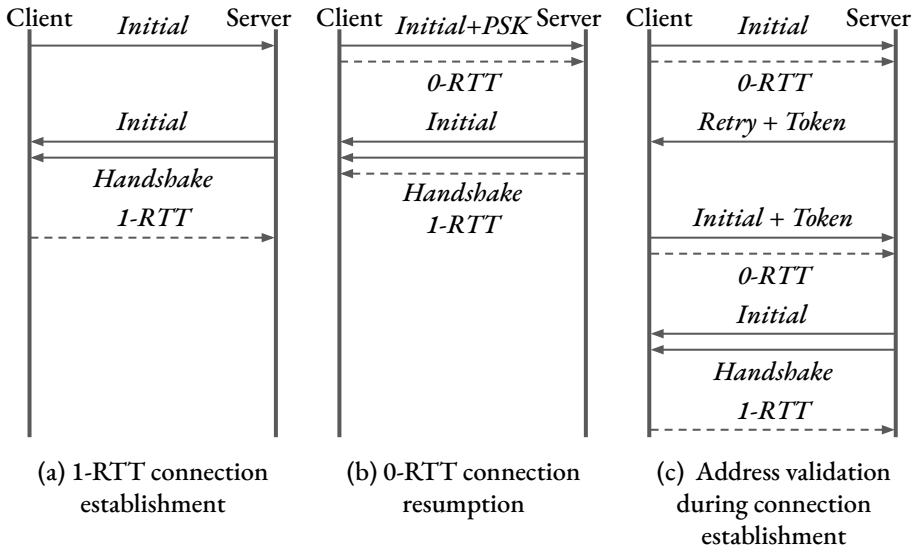
QUIC clients and servers can choose to not use Connection IDs with *Zero-length* Connection IDs to shorten the header of packets they receive. In this case, they loose the properties of the CIDs discussed in this subsection. The NEW_CONNECTION_ID and RETIRE_CONNECTION_ID frames are used to manage the set of Connection IDs provided by a host to their peer.

1.10.2 Modes of connection establishment

QUIC offers two modes of connection, that can optionally validate first that the client possess the source address found in its packets before proceeding with the rest of the QUIC handshake. Figure 1.11 illustrates these two modes (a, b) and the optional preliminary address validation (c). The dashed lines represent optional packets that may contain application data. The first mode is the *1-RTT connection establishment* (a). It can be used with all QUIC servers. It comprises three types of packets: *Initial*, *Handshake* and *1-RTT*. The *Initial* packets contain the TLS ClientHello and ServerHello messages. The ClientHello also announce QUIC *Transport Parameters* inside a dedicated TLS extension which set initial values of the QUIC connection parameters but that can also negotiate extensions to the protocol. The *Handshake* packets contain the subsequent TLS messages from the server, i.e., EncryptedExtensions, Certificate and Finished. They are protected using the intermediary TLS handshake keying material. The server's QUIC Transport Parameters are exchanged privately inside the EncryptedExtensions. After processing these packets and messages, the client can derive the final secure TLS keying material to protect and exchange 1-RTT packets containing both application and control data.

The second mode of connection establishment (b), *0-RTT connection establishment* leverages the TLS 1.3 Session resumption and Pre-Share Key mechanisms described in Section 1.8.1. As such, it requires that a previous connection to the QUIC server was made and a PSK was exchanged. Using this key, the client can *resume* the cryptographic state of the previous connection instead of performing a full cryptographic handshake. This key can also be used to secure application data inside 0-RTT packets that can be sent along with the Initial packet. These 0-RTT packets bear different security properties than 1-RTT packets. Similarly to 0-RTT data in TLS 1.3, they can be replayed which may impact the applications using QUIC.

As the processing of Initial packets by QUIC servers has a given computa-



(a) 1-RTT connection establishment

(b) 0-RTT connection resumption

(c) Address validation during connection establishment

Figure 1.11: QUIC comprises three modes of connection establishment.

tional cost, some might prefer to validate first that QUIC clients possess the IP address indicated as source in their Initial packets (c). To this end, the QUIC protocol introduces the *Retry* packet that is sent by a QUIC server to provide a *Token* to the client. When the client is legitimate, it re-sends its Initial packet and includes this Token. The Token may be constructed in a way such that the server can verify it without storing an associated state, i.e. by encoding and protecting the client IP address and UDP source port inside the Token.

1.10.3 Packet and frame format

The basic protocol unit in QUIC are QUIC *frames*. A frame is a *Type-Value* record that can convey control and application data with a format that is specific to its type. RFC 9000 defines 20 different frame types, each having a dedicated use to accomplish a part of the protocol. We introduce the ones of interest throughout the rest of this section as they are needed to realise the mechanisms we describe.

QUIC frames are conveyed inside QUIC packets. QUIC uses *long* and *short headers* packets. The former are used for the first packets during the connection establishment, i.e., for Initial, Handshake and 0-RTT packets, and the latter for the rest of the connection with 1-RTT packets. Figure 1.12 illustrates the format of 1-RTT packets. One can note that it contains a more restricted number of fields than the TCP segment format. QUIC favours using frames to encode control information. The first byte of the packet contains a number

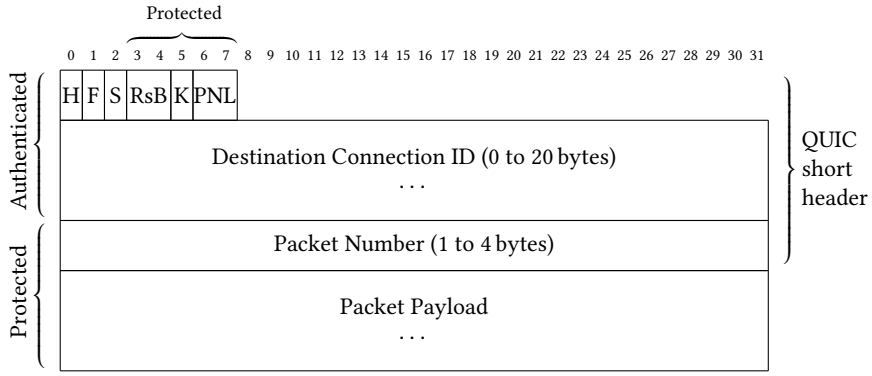


Figure 1.12: The QUIC 1-RTT packet contains minimal metadata in plaintext and authenticates and protects most of QUIC packets.

of flags. The first bit (H) is set to 0 to indicate a short-header packet. The second bit (F) is fixed to 1 for all packets defined in RFC 9000, such that other protocols can be distinguished on a single UDP flow [RFC9312]. The third bit (S) *spins*, i.e., changes in value, with every round-trip, such that passive observers may infer the round-trip time for performance monitoring purposes. The fourth and fifth bit are reserved. The sixth bit (K) encodes the current key phase, such that a QUIC host can detect TLS key updates in the same manner that it would process the TLS KeyUpdate message described in Section 1.8.3.

The last two bits encode the size of the *Packet Number* field in the header from one to four bytes. The QUIC packet number spans 64-bit and is reconstructed using the value of this field in its lower-order bits. It uniquely identifies a packet of a given type within a QUIC connection and is monotonically increasing. As such, it encodes the transmission sequence of QUIC packets over a connection. Initial, Handshake and 0/1-RTT packets have different *Packet Number Spaces (PNS)*. The third PNS is also named the *Application* PNS. These exist to ensure a cryptographic separation between the contexts in which packets can be processed.

The *Destination Connection ID* contains a Connection ID that was provided by the peer as discussed in Section 1.10.1.

The *Packet Payload* contains a sequence of frames. As they can contain control and application data, QUIC imposes fewer constraints for extensibility than TCP with its TCP Options space. In addition, a QUIC packet is protected by applying the AEAD algorithm negotiated during the TLS handshake, as discussed in Section 1.10.2, in various manners on packet. All fields of the QUIC packet are authenticated and integrity-protected, such that they cannot be altered. Only the first three bits of the packet and the Destination Connection ID are not confidential, in order to aid the receiver to identify

the corresponding connection and decrypt the packet. The packet payload is encrypted using the packet number as part of the AEAD nonce and the rest of the header as associated data. The Packet Number field and the last five bits of the first byte are confidentiality-protected using *Header Protection*. As they are part of the associated data of the encrypted packet payload, they become fully protected. While a modification to the packet number cannot be detected by the *Header Protection*, the subsequent failure at decrypting the packet payload ensures the full protection of the packet. This extensive use of protection on QUIC packets greatly alleviates the possible interference of middleboxes, such as faced by TCP and IP, such that QUIC should retain an important extensibility in the future. In addition, any QUIC packet can be decrypted regardless of the receive order of packets or packet losses.

A QUIC packet is never retransmitted, i.e., its packet number is never reused on a connection. Instead, the lost frames it contained are placed inside a new QUIC packet when they carry control or application data that should be retransmitted as is. QUIC thus never faces the ambiguity that exists for TCP regarding its sequence numbers.

1.10.4 Streams

QUIC enables applications to exchange data reliably using QUIC *streams*. These are used to realise the stream mode, stream multiplexing and reliable transfer services. They are either bidirectional or unidirectional, and deliver data in order. They can be initiated by QUIC clients and servers. Streams are identified by a 62-bit value and opened in order. The two lower-order bits of this identifier encode whether the stream is bidirectional or unidirectional and the peer that opened it.

A set of QUIC frames encode the different state transitions of QUIC streams. The first is the STREAM frame, which conveys a chunk of data of a given stream. It implicitly creates the QUIC stream and eventually marks the end of the stream. The RESET_STREAM frame requests the abortful closure of a stream. The STOP_SENDING frame requests the sending of a RESET_STREAM frame by its peer.

Given that QUIC packets can be decrypted and processed regardless of packet losses and reordering, applications can ensure a high resilience to Head-of-Line blocking by placing a single STREAM frame per QUIC packet. In this manner, the loss of a packet only blocks the QUIC stream corresponding to the frame it contained.

1.10.5 Datagrams

QUIC also offers an unreliable message mode service using QUIC *datagrams*. These are defined in an extension and enabled using a dedicated QUIC Trans-

port Parameter [RFC9221]. Each datagram is conveyed in a single DATAGRAM frame and thus cannot span several QUIC packets. Datagrams are delivered unreliably, i.e., the order in which they are received may not correspond to their transmission order and some datagrams might be missing. However, unlike UDP, they are considered for congestion control as further discussed in Section 1.10.6.

1.10.6 Loss recovery and congestion control

A QUIC receiver informs the sender of the QUIC packets it received using the ACK frame. The packet number space an ACK frame refers to is implicitly indicated by the type of packet it is placed in, e.g., an ACK frame in a 1-RTT packet acknowledges 1-RTT packets. The format of the ACK frame enables the receiver to indicate the blocks of continuous packet numbers it received. It forms selective acknowledgements akin to the TCP SACK option. However, given that a QUIC frame can span an entire packet, the ACK frame is much less restrictive and allows more precise reporting of received packets to the sender. The receiver can also encode in the ACK frame the time spent between the receipt of the largest packet number acknowledged and the sending of the frame, such that delays intentionally introduced, e.g., due to computations or packet pacing, can be subtracted from the RTT to obtain a more precise network delay.

QUIC draws from the experience of TCP and proposes a unified loss recovery mechanism. It leverages the more powerful acknowledgements of the ACK frame to detect losses based on two concurring criteria. A given inflight packet is considered lost when: (*i*) the packet is unacknowledged and subsequent packet has been acknowledged, (*ii*) the packet precedes an acknowledged packet by three or more packets *or* its loss timer has expired. The loss timer is computed as 9/8 of the RTT. This timer is slightly shorter than TCP's 5/4 RTT ratio as described in Section 1.4.2.

As the first criterion enforces that an acknowledgement is received to consider a packet as lost, and to cope with cases where no acknowledgement are received, e.g., as the loss packet constitutes the tail of the transfer, QUIC replaces the RTO and TLP mechanisms of TCP with a unified *Probe Timeout (PTO)*. For each packet, a PTO timer is computed based on the RTT and its variation, such that a reasonable margin is taken to wait for the packet acknowledgement. When the timer expires, the packet is not considered lost but a probe is sent to elicit a new acknowledgement. The sender then prepares a packet with new application data when available or retransmit previous data when not.

QUIC leverages its packet loss detection and acknowledgements to reuse the algorithms for congestion control that were defined for TCP. All packets

are considered for congestion control except packets that only contain *ACK* frames.

1.10.7 Connection Migration

QUIC addresses the mobility of hosts connecting to the Internet with the *Connection Migration* mechanism. When using TCP, a connection is tied to its 4-tuple, such that when a host cannot continue it from its source address or port, the flow is de facto interrupted. Using Connection IDs, QUIC enables connections to continue after these changes, regardless whether they are introduced by middleboxes as discussed in Section 1.10.1 or triggered by QUIC clients and servers as discussed in the rest of this section.

However, QUIC restricts connections such that they can only send *non-probing* packets, i.e., packets that carry non-probing frames such as STREAM and DATAGRAM frames, using a single 4-tuple at a time. As such, QUIC does not enable several *network paths* to be used at the same time. Before non-probing packets can be sent using a new 4-tuple, a QUIC host has to perform *Path Validation*, which confirms that its peer can receive packets on the corresponding path. The validation process consists in sending a PATH_CHALLENGE frame containing 8 bytes of random data that the peer is expected to echo in a PATH_RESPONSE frame. As Path Validation only verifies a single direction, i.e., from its sender to the receiver, it performed in both directions of a network path.

The first version of the QUIC protocol as defined in RFC 9000 only enables servers to trigger migration once after the handshake. To do so, a server includes the *Preferred Address* transport parameter in the handshake to indicate the address and port that the client should use after the handshake to continue the connection. When the handshake is complete, the client changes its destination address and port according to the transport parameter and *validates* the path. When the path is validated, the connection continues.

QUIC clients can migrate at any point after the handshake by validating new network paths and then changing the network path on which non-probing packets are sent to a validated path. This process is illustrated on Figure 1.13 with a QUIC client changing its source IP address. The path validation spans an RTT. To prevent linkability between UDP flows of a QUIC connection, Connection IDs should be changed when migrating.

After migrating to a new network path, each QUIC host must reset its congestion control state, as it pertained to the previous path and is unlikely to be relevant for the new path. In addition to the Path Validation process, the reset of the congestion control impacts the throughput available to application when migrating.

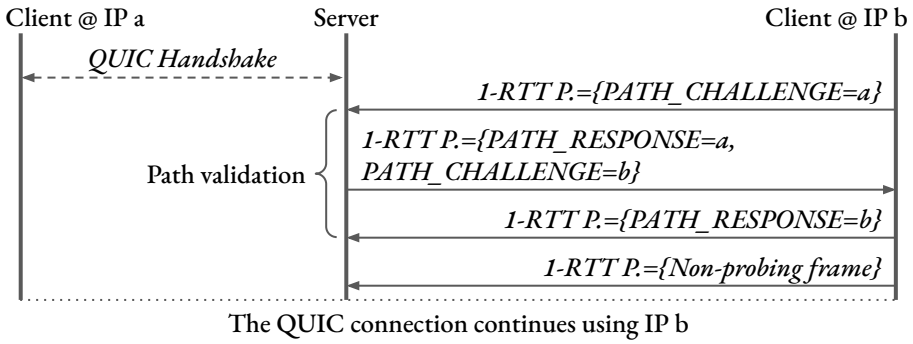


Figure 1.13: A QUIC client can change its source address and keep its QUIC connections active.

1.11 Multipath extension for QUIC

We discussed a number of limitations in which QUIC hosts can use network paths in Section 1.10.1 and 1.10.7. Researchers and engineers developed an extension to QUIC, known as *Multipath QUIC (MPQUIC)*, to enable the use of multiple network paths within a QUIC connection [DB17; De 20; Liu+24]. It extends the QUIC protocol while reusing as much as possible its existing mechanisms, such the packet format and path validation. The use of several paths brings new operating challenges that we also discuss in this subsection.

The Multipath extension introduces an explicit 32-bit *Path Identifier* to network paths used in a QUIC connection, such that the set of Connection IDs and the packet number space can be defined for each path. Separate packet number spaces have several advantages both from a design and a performance viewpoint [De 22]. The extension introduces modified versions of the ACK, NEW_CONNECTION_ID and RETIRE_CONNECTION_ID frames which include the path identifier, such that they can convey control information regarding a given path as part of a packet sent on another path. The path identifier does not appear in the packet header, as it is not modified by the extension, but is rather obtained by observing the Connection ID used. As a result, the extension cannot be used with Zero-Length Connection IDs.

1.11.1 Managing paths

The Multipath extension adds frames to manage the set of network paths used on a connection. The number of active paths can be controlled with the MAX_PATH_ID frame, paths can be abandoned with the PATH_ABANDON frame and preference can be expressed among active paths using PATH_BACKUP and PATH_AVAILABLE frames. No additional frame is required to open a new path. Instead, a QUIC host starts using a Connection ID from

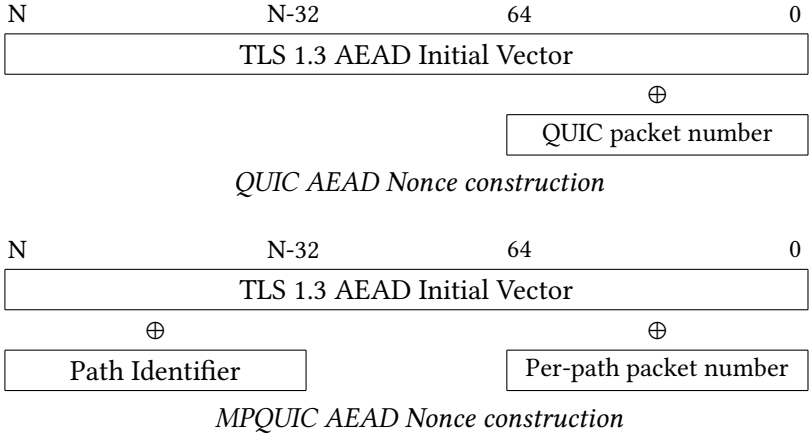


Figure 1.14: MPQUIC updates the AEAD Nonce construction of the QUIC protocol to enable protection of all paths of a connection.

the set of CIDs of an unused path identifier. When the network path used by the new path has not been validated before, Path Validation must take place before it can be fully utilised.

The policy for establishing new paths is entirely controlled by MPQUIC hosts. For instance, a client might establish a path per network interface it possesses, but it also can establish several paths using the same 4-tuple. The latter can be useful for certain extensions, such as Forward Erasure Correction [Mic23], where paths might have different uses and semantics.

1.11.2 Packet protection

The Multipath extension modifies the AEAD Nonce construct given that a MPQUIC connection can have several packet number spaces, and the packet number is involved in the forming of the AEAD Nonce used for packet protection as discussed in Section 1.10.3. Figure 1.14 compares the AEAD Nonce construction of QUIC and MPQUIC. MPQUIC incorporates the Path Identifier to the first four bytes of the Initial Vector (IV). Then the per-path packet number is incorporated to the last 8 bytes as in the QUIC construction. This guarantees that a unique nonce is formed from the IV at any time.

1.11.3 Congestion control

The QUIC protocol requires resetting the congestion controller state when migrating the connection. This can be achieved by implementations by maintaining separate states per path, such that they can resume the previous state when connection migration is inconclusive. When several paths are used,

MPQUIC enables each path to update its congestion control state based on the packet loss and acknowledgements it experiences.

When several paths share a common part, and are said to be partially *congruent*, there is a risk of unfairness to other flows as an MPQUIC connection with n paths can behave as n separate flows from a congestion control perspective. This risk can be mitigated by the use of *coupled* congestion controller, such as the Linked-Increase Algorithm (LIA) [Wis+11; RFC6356] and the Opportunistic Linked-Increase Algorithm (OLIA) [Kha+13].

1.11.4 Packet scheduler

When an MPQUIC connection uses several paths, its hosts face the problem of choosing a given path to send a given QUIC packet. This is known as *packet scheduling*. As for the management of the paths, the extension does not specify a *packet scheduler* and it is left to the MPQUIC implementation and application to decide how scheduling should be performed. A preliminary step before letting the packet scheduler decide is to rule out the paths which are blocked according to their congestion window and the packets already inflight.

There exist a number of packet schedulers for multipath transport protocols in the literature [Bon+20b]. The simplest one is the *Round-Robin* scheduler which considers each path for sending in turns. It is very simple to implement but also unlikely to provide good performance because it implicitly assumes the equality in characteristics of available paths. This is rarely the case given the heterogeneous nature of Internet access technologies. A more practical scheduler is the *Lowest-RTT First* scheduler which sorts the available path in increasing round-trip times. This is a good fit for applications that require a low delay. However, it may not always result in a low transfer *completion time* as it does not consider the available bandwidth per path nor the amount of data that is pending to be sent. The *Earliest Completion Time* scheduler does account for these metrics and chooses the path that leads to the shortest transfer time [Lim+17].

When scheduling packets containing STREAM frames of a given stream, an MPQUIC sender could be careful not to introduce Head-of-Line blocking when a low delay is required. When splitting a stream on several paths, there is a risk of introducing reordering which can block the processing of the stream. In practice, the round-trip time of the slowest path is often the limitation and scheduling a stream on several paths can increase the completion time proportionally to the RTT.

1.12 Multipath capabilities and user privacy in the Internet

The Internet protocols presented in this chapter were developed to meet the needs of its users. The first few connected computers in the 1980s were permanently installed to meet the need of researchers and governments. Over the years, more devices connected to the Internet, using different Internet access technologies and for a much more diverse set of uses. This has prompted a number of changes that we described in the protocols.

Since the start of the century, two major changes can be summarised. First, the security of Internet communications has been a central point. It is a broad effort that first addressed the security of application data and the authentication of servers with TLS and enabled many new types of transactions on the Internet. However, over the last decade, researchers put a focus on securing the metadata introduced by the functioning of protocols to mitigate pervasive monitoring [RFC7258]. At the same time, the tentative to extend transport protocols to better address new applications faced the interference of middleboxes, that blocked and altered extensions to IP and TCP. A unified response to this context can be found in the QUIC protocol, which extensively protects its control and application data.

Second, the restricted number of computers and services from the early Internet days greatly evolved since then and brought an important diversity in paths over the Internet. For instance, a domain name can be resolved into several IP addresses of different families. A mobile device may possess several access technologies, such as Wi-Fi, 5G and satellite communications. The characteristics of a path through the Internet can also evolve over time, due to congestion and *traffic engineering* [Red+20], i.e., the steering of packets within or from a network to another. In this context, it is both important for transport protocols to evolve to support the dynamicity brought by mobile devices and leverage path diversity to better serve applications. In this endeavour, the QUIC protocol also makes a step in this direction with the use of Connection IDs and its Connection Migration feature.

In the rest of this thesis, we describe five of our efforts to improve transport protocols and their uses given the context we described in this section.

Observing the Evolution of QUIC Implementations

2

In this chapter, we design and implement a test suite for QUIC servers and analyse its test results against several implementations. Our test suite comprises 32 test scenarios to evaluate the support of many features of the QUIC protocol. Through our results, we highlight the high dynamicity of QUIC implementations during their development phase.

Our work is contemporary to the pre-standardisation phase of the QUIC protocol during which a dozen of QUIC implementations [Gro18a] were actively being developed. They were likely to face similar problems as TCP implementations during the last decades [RFC2525]. The interoperability tests that were regularly organised by the QUIC IETF working group have helped to identify some ambiguities in the specifications and solve interoperability problems. We contribute to this implementation effort with a publicly available and detailed test suite for QUIC. To our knowledge, this was the first public test tool for the QUIC protocol.

We first motivate the need for a QUIC test suite in Section 2.1. We then describe the architecture of our test suite in Section 2.2. Section 2.3 analyses results collected from mid-March 2018 to the end of April 2019 using our test suite on the public servers that already implement QUIC. Section 2.4 concludes this chapter by reviewing the future prospects for our work and assessing how it can be freely improved, reused and extended.

The content of this chapter has been published in the article *Observing the Evolution of QUIC Implementations* [PDB18] and was presented at the EPIQ workshop of the CoNEXT 2018 conference.

2.1 Complexity of the QUIC protocol

The efforts of the QUIC designers and implementers is probably unique in the history of protocol design and implementation. Although many IETF protocols have been designed, very few have attracted so many implementers while the protocol was still being developed. As an illustration of the complexity of the QUIC protocol, Figure 2.1 plots the number of RFC2119 keywords (i.e., “MUST”, “MUST NOT”, “SHOULD”, “SHOULD NOT”) in the draft versions of RFC9000, the main document specifying the QUIC protocol [RFC9000]. We

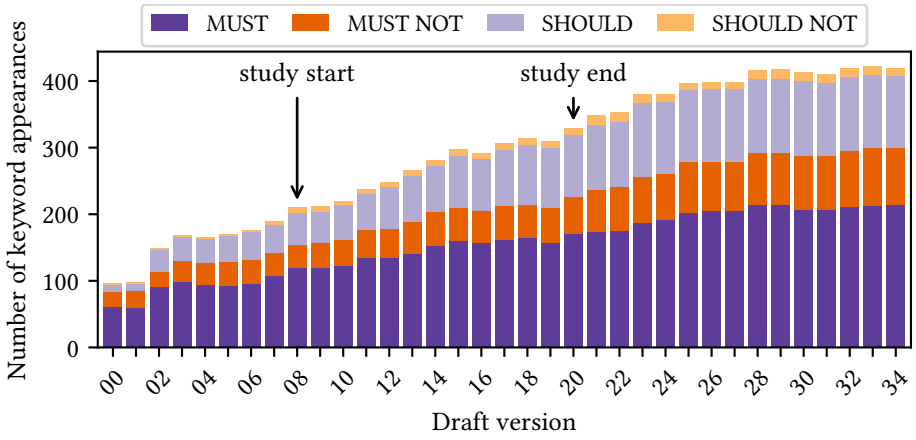


Figure 2.1: Keywords in draft versions of RFC9000.

can observe that the number of these keywords has almost constantly risen, reaching nearly four times more keywords in the final document than in its initial draft version.

Implementing network protocols is not a trivial task and several authors have proposed techniques to test and validate protocol implementations. Some of these techniques rely on formal methods to automatically derive the test suite from the implementation [Gra+03; HL91]. However, it is difficult to apply them to Internet protocols since their specifications are informal. Subsequent efforts to our work from the research community applied formal methods to QUIC [McQ+20; MZ19; Cro+21].

Researchers also have proposed different solutions to test and validate protocol implementations. Some have proposed techniques to validate complete TCP implementations [ME+04; Bis+05]. Paxson et al. proposed specific tools to validate the conformance of TCP implementations [Pax97; RFC2525]. With TBIT, Padhye and Floyd developed techniques to infer the characteristics of TCP implementations by interacting with them using specially crafted packets [PF01]. These tools have played an important role in improving TCP implementations.

2.2 The QUIC Test Suite

In this section we first describe both the approach and the architecture of our test suite. We then depict the test scenarios that constitute it.

2.2.1 Testing approach

Network protocol testing approaches can be categorised according to two dimensions, black-box versus white-box testing and passive versus active testing. The first dimension defines the perspective chosen to evaluate an implementation, i.e. an external or internal perspective. We chose the black box approach as want to test as much QUIC implementations as possible without being constrained by source code availability. Our test suite only observes the external behaviour from a QUIC implementation, i.e. the packets sent and received, to evaluate it.

The second dimension defines how the tool behaves with respect to the implementation under test (IUT). The first approach is passive testing. It has been used in earlier works with TCP [Pax97; RKS06; Jai+04], but is of limited use with QUIC given that most of the packets are encrypted.

The second approach is active testing, in which the tool used for experiments actively exchanges messages with the IUT. It requires the IUT to be available when using the tool. Several studies have applied this approach to TCP. TBIT [PF01] was one of the pioneering work in this domain. It has been extended in later works [ERA11; Yan+14]. Conducting active tests with TCP is becoming more difficult today given the deployment of various types of middleboxes that may interfere with active tests [Hon+11; MAF04; Hes+13]. By encrypting most of the control information and payload, QUIC exposes a smaller surface subject to ossification.

The objective of our test suite is to analyse how QUIC servers implementations are implementing the specifications by only exchanging packets with them. Given that the IETF specifications were still being developed during our study period, we only cover a subset of them as indicated by the arrows in Figure 2.1. We want the tool to be autonomous in two manners. First, it must be able to create QUIC packets on its own to perform the tests. Second, it must also be able to appropriately present bug reports for an implementer to locate which mechanisms caused their occurrence.

2.2.2 Architecture

Figure 2.2 illustrates the architecture of our test suite. Akin to TBIT [PF01], our test suite is constituted of several self-contained scenarios as illustrated in the top-left corner of the Figure. Each scenario is self-contained in the sense that it establishes one or several new QUIC connections to perform the test. A separate connection increases the isolation between two scenarios, in an attempt to accurately test a specific mechanism. Each test addresses a particular feature of the QUIC protocol. Combining several scenarios within a single connection is left as future work.

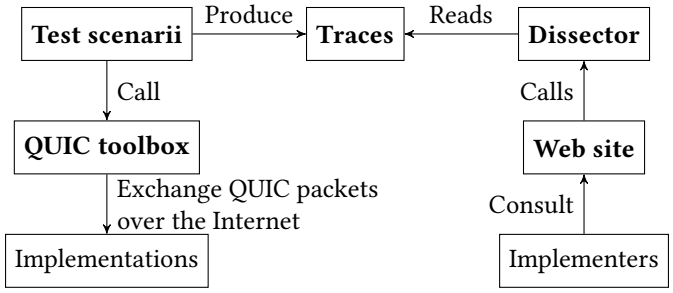


Figure 2.2: Tools forming the test suite.

We implement the scenarios on top of a high-level API, i.e. the QUIC toolbox, that allows easily manipulating QUIC packets. We implemented the toolbox using the Go language. Pieces of QUIC client behaviour are implemented as asynchronous message passing objects, called agents. We implemented 17 different agents responsible for, e.g. issuing ACK frames in response to received packets, retransmitting lost data, interacting with TLS, decrypting and parsing QUIC packets, bundling frames into packets and performing the different types of handshake (1-RTT, 0-RTT, handling of Retry and Initial tokens, ...), sending HTTP/0.9 or HTTP/3 requests, enforcing flow control, etc.

This increases modularity by defining the behaviour of a test without having to reimplement mechanisms shared by several tests. For instance, the `address_validation` scenario, which tests whether a server validates the client address before sending significant amount of data to it, does not send acknowledgements but retransmits lost data and derive session keys from the handshake to decrypt all received packets. Therefore, it disables the agent responsible for acknowledgements, but enables the agents interacting with TLS and sending retransmissions.

Our QUIC toolbox depends on *picotls* [OHc24] for using TLS 1.3 through a Go binding we wrote [Pir18a]. The interaction between TLS and the toolbox is isolated in a separate 115-line long agent. One can thus replace the TLS stack used by implementing a new agent providing equivalent functionalities.

We synchronise the different agents using specific events inside a connection, e.g., a packet has been received or sent out, a new encryption scheme is available. This paradigm allows attaching new behaviours upon reception of these events without requiring extra coordination with other agents or having to define a common event loop for each connection. Agents can also emit events as their connection progresses. One drawback is that due to the concurrent execution of all agents, synchronisation must be planned carefully using channels, as racing between agents can occur.

Each test outputs its result in a JSON trace. Each trace contains an error code that summarises its outcome and a clear-text trace of the packets ex-

changed during its execution. The error code is not purely binary, i.e. passed or failed. It can be used to discern various cases of failures to help the implementer to locate which part of the tested mechanism was deemed as erroneously implemented. For instance, the `zero_rtt` test can report whether a valid resumption token was sent by the server, whether it allowed the test suite to effectively succeed a 0-RTT handshake and whether the test suite could perform a 0-RTT HTTP request.

A trace can also contain scenario-specific data, such as the list of the protocol versions that were announced during the `version_negotiation` test and the transport parameters received during the `transport_parameters` test. Using this feature, we instrumented several test scenarios to collect several metrics. We present three of them in Section 2.3.

To track the evolution of QUIC implementations, we ran an instance of the test suite every day from March 2018 until September 2023. The different scenarios are run in a randomised order to prevent a particular sequence of tests from repeatedly impacting the data collection. A publicly accessible web application allowed visualising the test results [Pir18c]. It eased the communication of bug reports to implementers and encouraged them to consult test results. The presentation of the results emphasised on the cause of the problem so that implementers can efficiently diagnose which mechanism was erroneously implemented. The website also provided a detailed description of each test.

Our web application embeds a packet dissector we implemented. We chose to develop our own because existing packet dissectors, such as those in Wireshark [Gou18], did not consistently support QUIC during our study period. Being able to dissect the packets exchanged by the test suite greatly improves its ability to efficiently present bug reports. The dissector operates based on a specification of the protocol written in YAML and a cleartext trace of the packets exchanged. We maintain separate specifications for different QUIC versions in order to ensure backward compatibility.

Our QUIC toolbox consists of more than 6200 single lines of Go code. The web application consists of 1100 single lines of Python and HTML/JavaScript code, while the dissector is 300-line long supplemented by 1600 lines of YAML for protocol descriptions.

2.2.3 Testing the specification

We derive test scenarios from the QUIC specification. This process cannot be automated, because the specification is written in English in an informative style. We analyse the sentences containing strong indicators of importance as defined in RFC2119 [RFC2119], i.e. sentences containing the words “MUST” or “MUST NOT”. We then extract rules from these sentences that should not

be violated throughout a test. Based on these rules we design scenarios that ensures that a set of these rules are not violated, such that a combination of the scenarios cover all rules. The tests follow the evolution of the specification and are updated accordingly. We prioritise the design of tests that involve features chosen by the QUIC working group as part of the *Implementation Drafts* [TE18] to provide valuable feedback.

Our test suite contains 32 test scenarios. In the interest of space, we do not present all of them but summarise the mechanisms they verify. Eleven of them check several aspects of the QUIC handshake and long-headers packets, such as the *0-RTT*, the exchange of the *Transport Parameters*, the *Version Negotiation* and the *Key Update* mechanism. Eight of them focus on QUIC streams, e.g., bidirectional and unidirectional support, flow control and reordering in stream transitions. Two of them test the handling of acknowledgements and the support for Explicit Congestion Notification (ECN). Four tests explore connection migration and the use of Connection IDs. Finally, five scenarios tests aspects of HTTP/3. These 32 test scenarios are implemented by about 2100 lines of Go code.

As an example, in September 2018, by manually analysing the 14th version of `draft-ietf-quic-transport`, we identified 29 strong requirements covered by the test suite, i.e. "MUST" and "MUST NOT". We note that 41 of the 203 strong requirements are only applicable to QUIC client implementations and thus out of the scope of our tool.

2.3 Test Results

We focus on the analysis of our QUIC test suite from mid-March 2018 to the end of April 2019, a time frame during which its development and maintenance were the most active. We updated the list of public endpoints during this time when they were publicly announced on the communication channel dedicated to testing coordination [Gro18b].

We report our results into three phases. First, we provide a high-level view showing key metrics collected by our test suite. Then, we dig in two case studies on two specific test scenarios. Finally, we present a snapshot of the test results to show the diversity of behaviours between studied implementations.

2.3.1 Measurements

In this section, we present three metrics extracted from the data collected by the test suite, i.e. the QUIC version announced by tested endpoints, the handshake success with our test tool and the test outcome percentage. For each metric we explain how the measurements were conducted and what conclusion can be drawn from them.

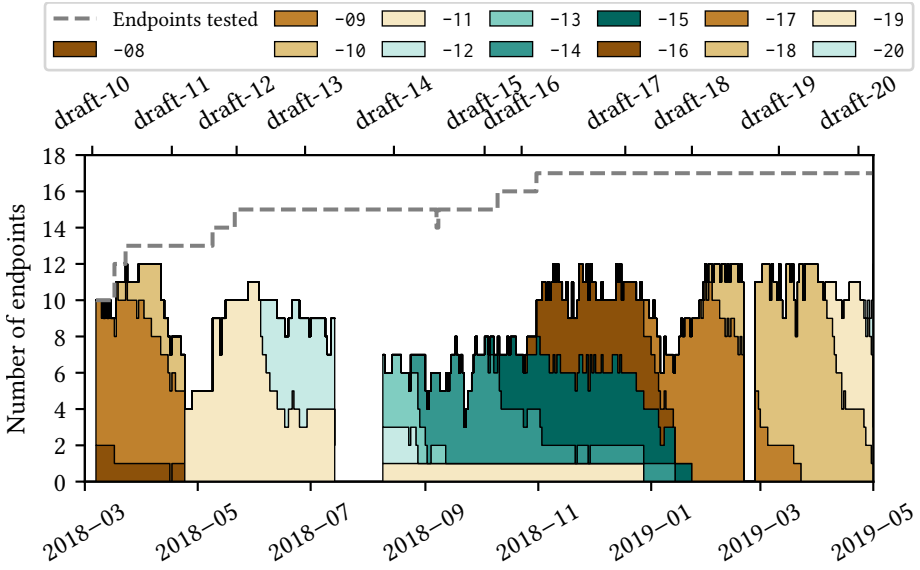


Figure 2.3: Number of endpoints announcing different draft versions.

2.3.1.1 Deployment of QUIC versions

We first report the result collected by the `version_negotiation` scenario. This test triggers the version negotiation process and records the versions announced by the tested implementations. In a sense, this is similar to the measurements carried out by Rüth *et al.* to identify the number of servers that support Google's version of QUIC (gQUIC) over the entire IPv4 addressing space [Rüt+18]. Figure 2.3 summarises our results over the 6-month period. It also indicates the number of endpoints we tested over this period. This number fluctuated with the appearance of new implementations and others that became unmaintained. The gaps in the data were caused by bugs and operational issues that prevented the tests from running during periods of lesser involvement with the tool.

We can observe that when a new version of the specification was published, most implementations chose to stop supporting the previous version in favour of the new one without maintaining backward compatibility. This is reflected in the figure by a simultaneous increase and decrease between two successive versions. This lack of backward compatibility is normal for prototype implementations. It contrasts with the findings of Rüth et al. about public gQUIC servers that gradually update the set of versions they support.

QUIC draft versions can introduce a lot of changes, including modifications to the QUIC invariants about the public header format. `draft-11` is an example. It introduced a version negotiation process that is not backward-compatible. We updated the test suite to support `draft-11` shortly after its

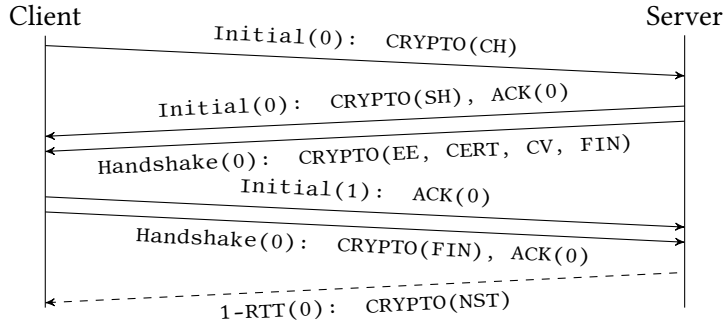


Figure 2.4: 1-RTT connection in our `handshake` test.

publication. From this point, we were unable to observe prior versions.

We can conclude from Figure 2.3 that the implementers of QUIC often need between fifteen days to a month to integrate the changes published in a new version of the specification to their implementations. As a result, tracking the behaviour of these QUIC implementations required us to regularly update the test suite. In addition, the total number of endpoints announcing versions tend to follow the IETF meetings cycle, with March, July and November repeatedly exhibiting a higher active implementation count.

2.3.1.2 Successful handshakes

Version negotiation and connection establishment being separate mechanisms, there could be a mismatch between the number of implementations that announce a particular version and the number that effectively support it. To investigate this possibility, we report the number of implementations that successfully performed a 1-RTT handshake with our test suite. It is based on the data collected by the `handshake` scenario which discerns various causes of 1-RTT handshake failure. Figure 2.4 illustrates the behaviour of our `handshake` test and indicate the types of packet exchanged, the QUIC frames and the TLS messages they contain. We can observe that the test performs a complete 1-RTT handshake and derives the corresponding session keys. The server can provide a *New Session Ticket* (NST) which will be recorded and decrypted by the test. As some tested implementations used a self-signed certificate, our test does not check the certificate validity.

Figure 2.5 reports the number of endpoints that succeed our handshake test over the 6-month period. During this period, we implemented eight draft versions of the specification. We chose to not deploy `draft-10`, because most implementers indicated that they were willing to support the next version as soon as possible [Gro18b]. `draft-10` contained very few new features when compared to `draft-11`. This was also the case for `draft-16`,

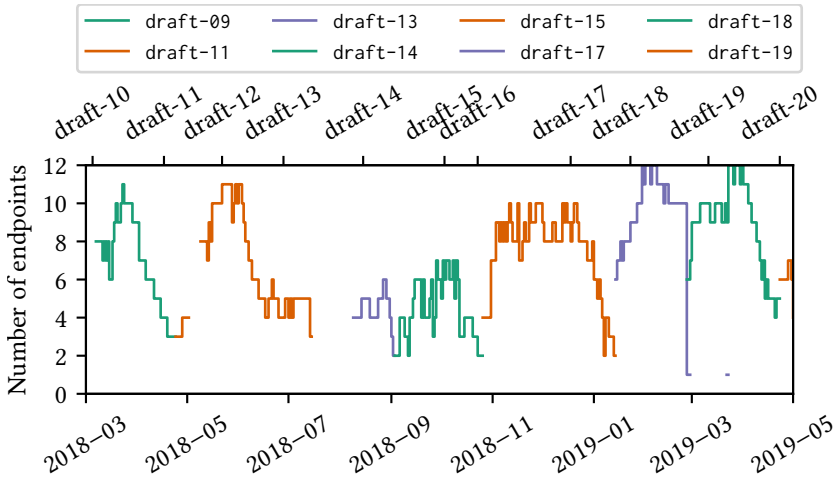


Figure 2.5: Number of endpoints succeeding our handshake test.

which contained changes more on the side of errata than protocol changes. Other unimplemented versions such as `draft-12` happened outside our active development time window. We can indeed observe in Figure 2.3 that only a maximum of four implementations simultaneously announced its support.

Overall, the resulting graph contains several slight fluctuations when compared to Figure 2.3. These fluctuations are a result of the rapid pace at which changes are deployed among all the tested implementations, our test tool included. Some of these changes have caused interoperability problems among them. This is expected as implementing the version negotiation involves simpler mechanisms than the 1-RTT handshake.

2.3.2 Test suite outcome ratios

We present the evolution of the success, failure and unsupported rates over the entire test suite during our data collection period in Figure 2.6. We computed the percentage over the tests that require the handshake to complete and only kept implementations that succeeded handshakes. We overlay the number of these implementations as well as the number of these tests on this figure. *Success* corresponds to a successful execution of a test. *Failure* reports that a test violated the specification and *Unsupported* reports the tests for which a prerequisite was missing, e.g., the endpoint crashed, no IPv6 address could be resolved, no unidirectional streams were available, ... In this case, as the test could not be conducted, no conclusion can be drawn.

We can note that most of the fluctuations occurred during March 2018 when the first tests were introduced. Based on the feedback of implementers [Gro18b], we updated several tests to improve their correctness and address false posi-

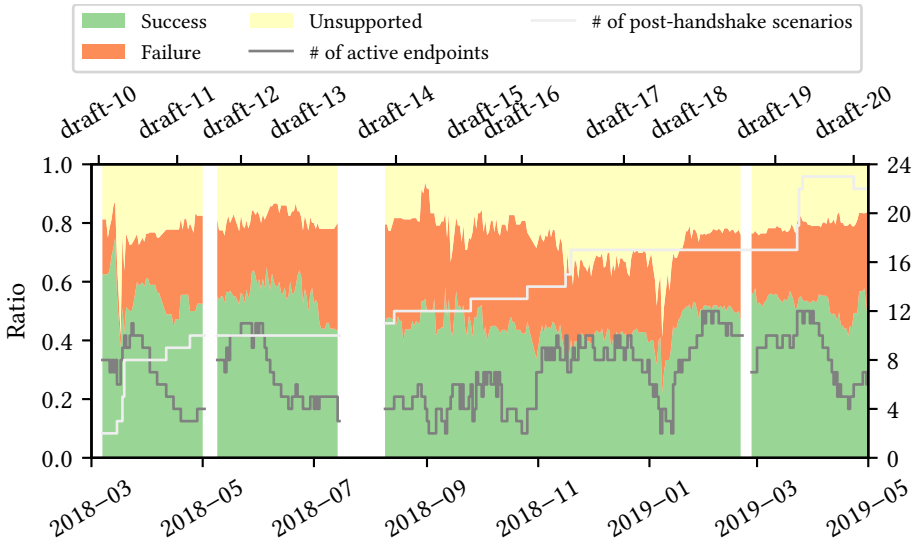


Figure 2.6: Percentage of outcomes for post-handshake tests.

tives.

The Figure illustrates that the *Success-Failure* ratio follows the number of implementations available. Periods of higher interoperability activity are the ones with the highest *Success* to *Failure* ratio. Periods of lower activity have higher *Failure* and *Unsupported* rates. Overall, while rates remain stable over our study period, as the number of test scenarios within our test tool greatly increased, the features correctly supported by QUIC implementations progressed at a steady pace. Finally, the curves show many fluctuations that are indicative of the dynamic nature of the development process of these QUIC implementations.

2.3.3 Case studies

We review in this section some test scenarios which reported bugs in several implementations. For each test, we first explain its intent, then we report the evolution of its results based on the feedback submitted to and received from the implementers. We concentrate on a 3-month period starting on the 1st of March 2018.

Flow control. Flow control is an important part of a transport protocol. It prevents a fast sender from overwhelming a slow receiver. A peer can signal flow control through two different mechanisms in QUIC. The first are the transport parameters. For instance, parameter `initial_max_stream_data_bidi_local` allows a client to limit the amount of data that a server can send on a stream initiated by the former. The second is the `MAX_STREAM_`-

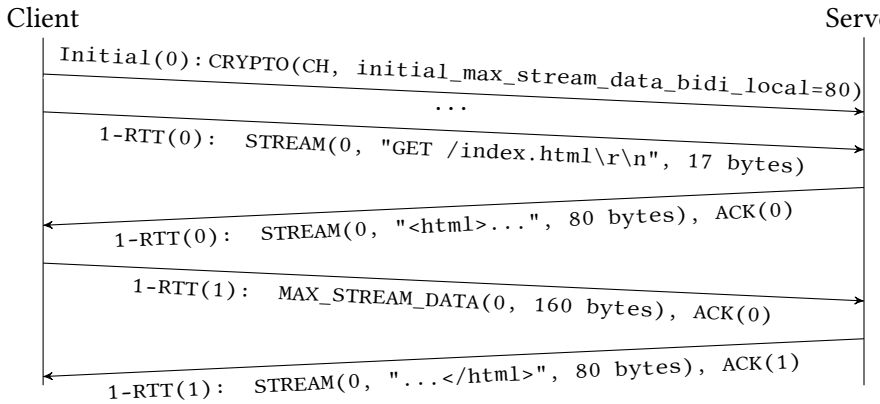


Figure 2.7: Example flow for our `flow_control` test.

DATA frame, which advertises higher limits.

The `flow_control` test, as illustrated in Figure 2.7, initiates a connection and sets the `initial_max_stream_data_bidi_local` parameter to 80 bytes. This limit has been chosen sufficiently low to be smaller than most of the web pages that are hosted by the endpoints. Once the connection is established, the test sends an HTTP request and waits for the server to send the first 80 bytes of the response. The server must not send more than 80 bytes because of the limit imposed by the transport parameter. Once these bytes are received, the test sends a MAX_STREAM_DATA frame that raises the limit to 160 bytes. The test ensures that the server resumes the sending of data after receiving the frame.

We found several implementations failing this test at different stages of the specification process. On the 10th and 17th of March 2018 two implementations entered a loop when running the `flow_control` scenario. We reported these bugs and discussed with their implementers. The first was repeatedly sending ACK frames due to an incorrect integration of flow control with other mechanisms. The second was sending empty STREAM frames, which is forbidden by the specification, because of a missing corner-case when clamping these frames according to flow control. The test results collected after the 20th of March indicated that both implementers had fixed their bug.

We found another implementation that incorrectly implemented flow control on the 23rd of March. It only divided its response into two pieces, the first being 80-bytes long. We reported the bug and were notified that a fix was implemented, which was confirmed by the test results shortly after. On the 18th of May 2018, after this implementation added support for `draft-11`, we observed a regression regarding this test. The implementation aggressively sent

STREAM_BLOCKED frames and retransmissions of the second half of data requested. We did not observe the deployment of a fix before the end of our data collection. We later learned that its developer was not active any more during this period.

Stream transitions reordering. A QUIC implementation must be able to react appropriately when packet reordering occurs. We can discern two cases which can induce packet reordering. The first is introduced by the use of different network paths, due to, e.g., load balancers. The second one is caused by a packet loss during the transmission of a series of packets. The data of the lost packet will be retransmitted and received after the rest of the series.

The `stream_opening_reordering` test simulates the first type of reordering. It initiates a connection and then sends an HTTP request in two packets. The first packet contains the graceful closure of the client-side of the request stream. The second contains the data of the stream, which contains the request. The first packet is sent with a higher packet number than the second packet. The test successfully completes once the server has responded to the request.

We report three of the cases we observed during the 3-month period. The first one lead to a one-to-one conversation with an implementer. The scenario triggered a livelock in their implementation and the latter did not produce any kind of observable external behaviour. We provided assistance to install the test suite and run it against a local and better-instrumented version of the implementation.

On the 11th of May 2018, we detected a regression for a particular implementation for which support of `draft-11` was recently added. We were not actively analysing the data on this day and thus did not report the bug. We later found that the implementer had consulted the test result and fixed the bug. We argue that this is an indication of the benefits of an autonomous test suite that runs on a daily basis and provides public results.

Finally, this test triggered a bug in the ACK frame generation of an implementation on the 22nd of May. We believe that the bug was discovered shortly after its introduction, as the results from the past days did not reveal it. The bug caused the generated ACK frame to report $2^{64} - 1$ missing packets, probably due to an overflow induced by the reordering of packets. Indeed, considering that one could determine the gap between the received packet and the last received packet by subtracting the received packet number with the last received packet number, reordering then causes an overflow. The implementation source code not being public, we could not confirm this hypothesis.

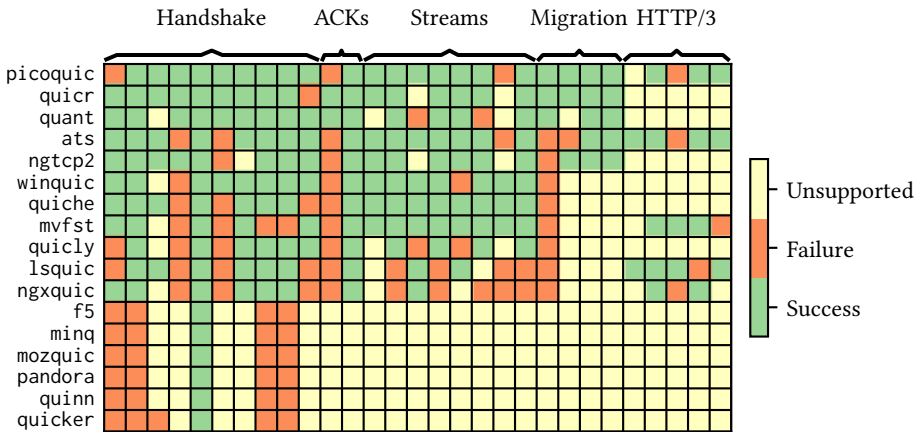


Figure 2.8: Results grid on the 1st of April 2019.

2.3.4 Results grid

We conclude this section by presenting Figure 2.8 which summarises the outcomes for the different tests and QUIC implementations. The grid is a snapshot captured on the 1st of April 2019.

Those results show the diversity of the outcomes generated by the available implementations. We can observe that six of them, located in the bottom part of the grid, only succeeded a single scenario. This test sends a single Initial packet containing PADDING frames. Discarding it completes the test. These endpoints were not actively maintained any more. For the basic parts of the QUIC protocol, i.e., the handshake, acknowledgements and streams mechanisms, are correctly implemented in various degrees. However, for more advanced features such as the Connection Migration and HTTP/3, these are implemented in a more binary fashion. Some implementers expressed interest in these features and implemented them in their entirety. Some test scenarios reveal features that are supported by only a fraction of implementations, such as ECN marks in ACK frames, key updates, and the *Spin* bit.

2.4 Discussion

In this chapter we have proposed a first active test suite for the QUIC protocol based on its specification within the IETF. We detailed its architecture and the supported test scenarios. We presented the results collected using this test suite and reported two case studies. The test suite has been used by the QUIC community. Its source code is publicly available under an open-source licence[1]. Two implementers integrated the test suite as part of their workflow,

[1]See `https://github.com/QUIC-Tracker`

independently of our public instance.

The tool being open also implies that it can be reused, extended and improved by the QUIC community. Due to the very modular design of our architecture, it can be extended in different ways. New scenarios can be implemented to cover new features of the protocol and collect new metrics. The QUIC toolbox can be reused for other purposes. It can also be extended with new features that are not currently supported, such as sending coalesced packets, or improved with a better user-facing API. The visualisation application can also be improved, e.g., by adding more feedback to the implementers based on the trace format. We note that one is not required to use this web application, and can instead consume the test results in the traces using other applications or command-line tools. For instance, we developed a set of scripts that allows generating CSVs based on these traces, which were used to produce Figure 2.3 and Figure 2.5[2].

So far, our QUIC test suite is limited to QUIC servers only. The QUIC protocol also requires the compliance of QUIC clients to the specification. Including them in our study would raise several challenges beyond the additional implementation efforts.

How to initiate connections from clients to the test tool? We chose a black-box approach as the source code of QUIC implementations is not always available. Applying this approach to client testing requires some techniques to encourage diverse clients to connect to the test tool.

How to identify the various implementations connecting to the test tool? In the server-side approach, we know to which servers the test tool connects to. If many clients can anonymously connect to the test tool, correctly identifying the tested implementations becomes critical for providing relevant feedback to the QUIC implementers.

2.4.1 Uses of our work within the state of the art

Our work has been used by the research community in two other published papers. First, in 2020, Reen *et al.* built a differential fuzzing framework to detect strategies of QUIC endpoints to elude Deep Packet Inspection systems [RR20]. In addition, they found four critical security vulnerabilities in the tested implementations. For this purpose, they reused our high-level QUIC API to produce the sequence of packets required by their tool. Second, in 2021 and for the SIGCOMM conference, Ferreira *et al.* built a framework for automated closed-box learning and analysis of models of TCP and QUIC implementations [Fer+21]. They used our QUIC test suite as their reference

[2] `https://github.com/QUIC-Tracker/web-app/tree/master/quic_tracker/postprocess`

implementation. They detail how they added hooks inside our tool for their purposes, which was facilitated by its modular nature.

2.5 Conclusion

In this chapter, we were actively involved in the standardisation and interoperability testing efforts of the QUIC protocol by proposing its first test tool. The tool can test a variety of mechanisms found in the protocol by interacting with servers on the Internet and report on their conformance to the specification. Our results highlight that the tool was useful to implementers and helped uncover several bugs in implementations. In addition, our tool was useful to the research community as well, with two published papers reusing part of our tool for other research purposes on QUIC.

Pluginizing QUIC | 3

In this chapter, we completely revisit the extensibility of transport protocols. We consider that transport protocols should provide a set of basic functions which can be tuned, combined and dynamically extended to support new use cases on a per-connection basis. We apply this approach to the QUIC protocol, which we studied in the previous chapter, detailing its complex features and their implication on the behaviour of its implementations. Such an approach could enable QUIC applications to adapt the underlying transport layer to their specific needs, e.g., using specialized retransmission algorithms or taking advantage of non-standard extensions. This would bring innovation back in the transport layer with researchers and software developers being able to easily implement, test and deploy new protocol features. For this, we leverage the extensibility and security features of QUIC. We make four main contributions in this chapter.

- We design a technique where an extension to the QUIC protocol is broken down in a *protocol plugin* which can be dynamically attached to an existing implementation. These plugins interact with this implementation through code which is dynamically inserted at specific locations called *protocol operations*.
- We propose a safe and scalable technique that enables the on-demand exchange of protocol plugins over QUIC connections. This solves the deployment problem of existing protocol extensions.
- We implement a prototype of Pluginized QUIC (PQUIC) by extending `picoquic` [Hui18], one of the most complete implementations of IETF QUIC [PDB18]. We add to `picoquic` a virtual machine that allows executing the bytecode of protocol plugins in a platform independent manner while monitoring their behaviour.
- We demonstrate the benefits of PQUIC through plugins that add new monitoring capabilities and support the Unreliable Datagram Extension [RFC9221]. We build a VPN application on top of those plugin-implemented capabilities to demonstrate the flexibility of our approach.

This work was conducted in collaboration with other researchers such that we took part in equal efforts in the design and implementation of the

core of PQUIC. We present it in this thesis to reflect our involvement and to serve the understanding of the rest this chapter. The two plugins we describe are our contributions. The rest of the plugins are described in the article *Pluginizing QUIC* published and presented at the SIGCOMM 2019 conference [De +19] and in two other theses [De 20; Mic23] and .

This chapter is organized as follows. Section 3.1 overviews the design of PQUIC and how a PQUIC implementation supports plugins. Section 3.2 presents several use cases of protocol extension with plugins.

3.1 Pluginized QUIC

Our design for **Pluginized QUIC** (**PQUIC**) builds upon the QUIC protocol and does not introduce changes to its specification [RFC9000]. From the implementation viewpoint, the main difference between a PQUIC implementation and a QUIC one is that PQUIC is easily customizable on a per-connection basis. This customization relies upon a modular, extensible design that allows adding and modifying behaviours for the target flows. A PQUIC implementation can be extended by dynamically loading one or more *protocol plugins*. A *protocol plugin* consists of platform-independent bytecode which can be executed within the PQUIC implementation.

A PQUIC implementation provides an API to protocol plugins. Most protocol implementations are designed as black-boxes that expose a small external API to applications. For example, the Linux TCP implementation exposes the socket API. A PQUIC implementation can be represented as a gray-box containing a set of functions that are exposed to protocol plugins. In PQUIC, we call these functions *protocol operations*. These are common routines being part of any implementation, and the workflow of PQUIC can be expressed as a succession of such protocol operations. As in a C API, each protocol operation has a specification and a set of conditions under which it should be called. Protocol operations in PQUIC include the parsing and the processing of frames, setting the retransmission timer, updating the RTT, removing acknowledged frames from the sending buffer, etc.

On-the-fly protocol plugin insertion. A *pluglet* consists of bytecode instructions implementing a function, e.g., computing an RTT estimate. A *manifest* contains a globally unique plugin name and indicates how to link several pluglets to a connection, i.e., to which protocol operations they should be attached to. The combination of pluglets and the manifest forms a *protocol plugin*. Once a PQUIC connection is established, PQUIC can potentially load plugins at any time.

Isolation between connections and between plugins. Each plugin is instantiated to operate on a given connection. Our framework ensures that each instance has its own memory which is only shared among pluglets of

this plugin. The plugin memory is isolated from access or sharing with any other plugin or connection. This yields strong memory safety guarantees for the plugins and the sharing of information. Interactions are still possible through the protocol operation interface or by calling the functions exposed by PQUIC. However, these are clearly defined information flows that ease reasoning about the behaviour and the safety of the plugins.

The rest of this section details the core elements of PQUIC. We first describe the environment executing pluglets. Then, we elaborate on the concept of protocol operations. We finally describe how PQUIC interfaces with pluglets.

3.1.1 Pluglet Runtime Environment (PRE)

Pluglets are the building blocks of the protocol plugins. These pieces of bytecode are independent of the PQUIC implementation itself. Therefore, an environment to execute them is needed and has to solve two major concerns. First, it has to provide an abstraction where plugins can run regardless of the underlying hardware and operating system. Second, given the untrusted nature of the plugins, the environment should keep each pluglet under control.

To address these two issues, PQUIC executes plugins inside a lightweight virtual machine (VM). Various VMs have been proposed for different purposes [Lin+14; Haa+17; Wan+15; Geo+10; BMG99; Fle17]. In this paper, our *Pluglet Runtime Environment* (PRE) relies on a user-space implementation [IO 18] of the eBPF VM [Fle17]. It is separate from the Linux in-kernel eBPF VM, as the latter comes with constraints that can severely restricts the behaviours that can be implemented. Although being present in the Linux kernel since 2014, the Linux eBPF VM can be too restrictive to implement some legitimate behaviours. While it has been used to support various services [Edg15; Gre15; Bra17], the Linux in-kernel eBPF VM includes a verifier that is very conservative, as it puts hard limits on the size and complexity of an acceptable eBPF program.

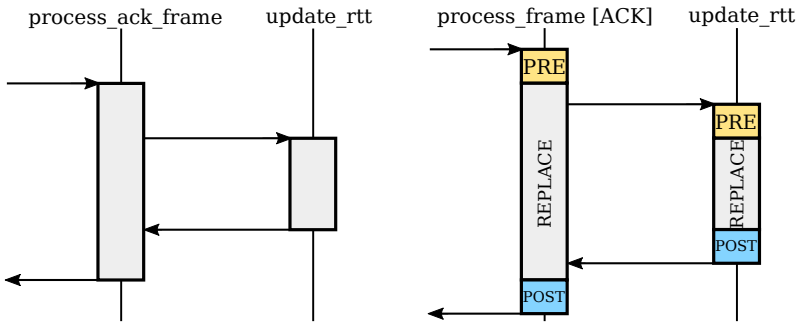
To lift these constraints, our implementation embeds its own eBPF VM and extends a relaxed version of its eBPF verifier with additional monitoring capabilities. Those are similar to works in Software-Based Fault Isolation [Wah+94; Yee+09]. First, our PRE checks simple properties of the bytecode to ensure its (apparent) validity. This includes checking that: (i) the bytecode contains an exit instruction, (ii) all instructions are valid (known opcodes and values), (iii) the bytecode does not contain trivially wrong operations (e.g., dividing by zero), (iv) all jumps are valid, and (v) the bytecode never writes to read-only registers. Furthermore, our PRE statically verifies the validity of stack accesses. A plugin is rejected if any of the above checks fails for one of its pluglets.

Second, our PRE monitors the correct operation of the pluglets by injecting specific instructions when their bytecode is compiled ahead of time (AOT compilation). These monitoring instructions check that the memory accesses are performed within the allowed bounds. To achieve this, we add a register to the VM that cannot be used by pluglets. This register is used to check that the memory accesses performed by a pluglet remain within either the plugin dedicated memory or the pluglet stack. Any violation of memory safety results in the removal of the plugin and the termination of the connection. The LLVM Clang compiler supports the compilation of C code into eBPF. This allows us to abstract the development of pluglets from eBPF bytecode and propose a convenient C API for writing pluglets.

3.1.2 Protocol Operations

In order to attach pluglets to PQUIC, we define an API revealing the insertion points and the interface between PQUIC and the pluglets. We break down the protocol execution flow into generic subroutines. These specified procedures are called *protocol operations*. Each has its human-readable identifier, inputs, outputs and specifications. A protocol operation with a parameter has a high-level goal, but its actual behaviour, and therefore its function, changes depending on the given parameter. This provides a generic entry point allowing the definition of new behaviours without changing the caller for, e.g., the serialization of new QUIC frames. Our PQUIC implementation currently includes 72 protocol operations. Four of them take a parameter. We split these operations into several categories. A first category concerns the handling of the QUIC frames. This includes their parsing, processing and writing. A second category groups all the internal processing of QUIC. It contains the logic for retransmissions, updating the RTT, deciding which stream to send next, etc. A third category involves QUIC packet management. It includes setting the *Spin Bit*, retrieving the connection IDs, etc. A fourth category contains several events in the connection workflow, whose protocol operations have empty anchor points, i.e., no default behaviour. For instance, a protocol operation exists after decoding all frames of an incoming packet or after a packet loss.

To illustrate how an implementation can be split into protocol operations, consider the example shown in Figure 3.1a. The processing of an ACK frame would likely be performed in its dedicated function. One of its subtasks is the computation of the RTT estimation, which is implemented in its own function too. PQUIC keeps the same programming flow. As shown in Figure 3.1b, PQUIC functions are wrapped by a protocol operation whose human-readable string identifier describes its goal. While the name of the protocol operation and the original function are similar, the processing of ACK frames is linked



(a) Monolithic QUIC design (b) PQUIC design

Figure 3.1: A monolithic protocol is turned into an pluginized one with protocol operations and pluglet anchors through the example of the QUIC ACK frame processing.

to a more generic `process_frame` operation taking ACK as parameter. As illustrated, a given protocol operation can call other operations. Furthermore, protocol operations are split into three *anchors*, each of which is a possible insertion point for a pluglet. Protocol operations with parameters propose a specific set of anchors for each parameter value. The first anchor, called *replace*, consists of the actual implementation of the operation. This part is usually provided by the original PQUIC function. This mode enables a pluglet to override the default behaviour. Because it may modify the connection context, at most one pluglet can *replace* a given protocol operation. If a second one tries to replace the same operation, it will be rejected and the plugin it belongs to will be rolled back. The two other anchors, *pre* and *post*, attach the plugin just before (resp. just after) the protocol operation invocation. These modes are similar to the eBPF kprobes in the Linux kernel [KPH16]. By default, those are no-ops in PQUIC. Unlike the *replace* anchor, any number of *pre* and *post* pluglets can be inserted for a given protocol operation. However, they only have read access to the connection context and the operation arguments and outputs. The only write accesses they have is to their pluglet stack and their plugin-specific memory. In the rest of the paper, unless explicitly stated, we discuss pluglet insertions in *replace* mode, and refer to pluglet inserted in *pre* and *post* as passive pluglets.

3.1.3 Attaching Protocol Plugins

Implementing protocol extensions may require a combination of several pluglets forming a plugin. The PRE provides a limited instruction set and isolates the bytecode from the host implementation. Therefore, plugins require an interface with which they can operate on their connection. Moreover, a plugin

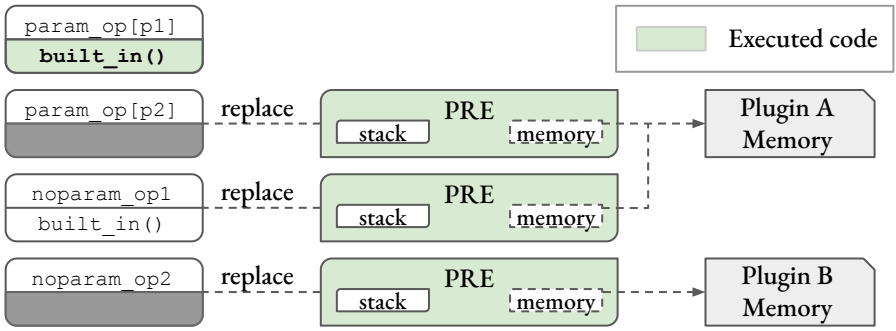


Figure 3.2: Attaching PREs in *replace* mode to protocol operations.

Functions	Usage
`get/set`	Access/modify connection fields.
`pl_malloc/pl_free`	Management of the plugin memory.
`get_opaque_data`	Retrieve a memory area shared by pluglets.
`pl_memcpy/pl_memset`	Access/modify data outside the PRE
`plugin_run_protoop`	Execute protocol operations.
`reserve_frames`	Book the sending of QUIC frames.

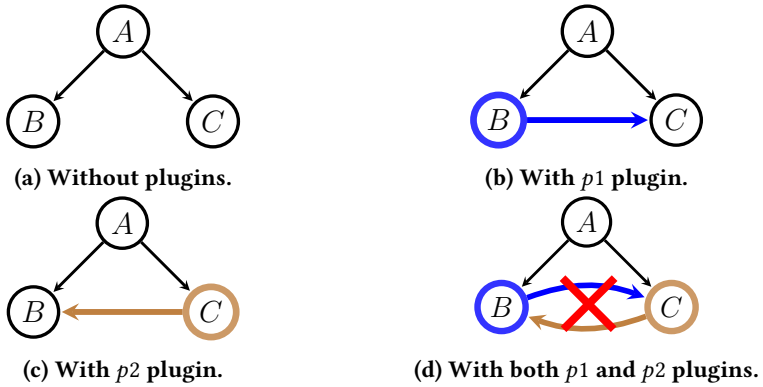
Table 3.1: PQUIC API exposed to pluglet bytecode.

might need to share some state among its pluglets.

To address these needs, PQUIC is organized as illustrated in Figure 3.2. As explained in the previous section, the behaviour of a protocol operation is either provided by a built-in function (e.g., `param_op[p1]`) or overridden by a pluglet (e.g., `noparam_op1`). Observe that plugins can also provide new protocol operations absent from the original PQUIC implementation. This can be done either by hooking a new parameter value for an existing protocol operation (e.g., like `param_op [p2]`) or by adding a new protocol operation (e.g., `noparam_op2`). PQUIC is thus extensible by design.

A PRE is created for each inserted pluglet. Each PRE contains its own registers and stack. The PRE heap memory points to an area common to all pluglets of a plugin, as illustrated in Figure 3.2. This link, ensured by the PQUIC implementation, provides a communication channel between pluglets. In addition to the isolation benefits, this architecture ensures that aggressive or ill memory management only affects the plugin itself. Thanks to our PRE, pointer dereferencing is restricted only to the pluglet stack or its plugin memory. In addition, the pluglet also needs to communicate with the host implementation to interact with its connection. As in a related work [AW18], PQUIC exposes some functions to the PRE. These functions form an API that pluglets can use (Table 3.1). We detail its six major operations below.

Exposing connection fields through getters and setters. Letting plu-



(a) Without plugins. (b) With $p1$ plugin.

(c) With $p2$ plugin. (d) With both $p1$ and $p2$ plugins.

Figure 3.3: Combining plugins requires protocol operation monitoring. (a), (b) and (c) are valid calls graphs while (d) is not since it creates a loop between B and C.

gins directly access the fields of PQUIC structures makes the injected code very dependent on the PQUIC internals. Consider the case of two hosts with different PQUIC versions. If the newest version added a new field to a structure being used by a pluglet, the offset contained in its bytecode would point to a possibly different field in the other version, leading to undefined behaviour. Therefore, this interface abstracts the implementation internals from the pluglets, making them compatible with different PQUIC versions or implementations. In addition, it allows the PQUIC host to monitor which fields are accessed by the injected code. A host could thus reject plugins based on the fields that it wishes to access. For example, a client could refuse plugins that modify the *Spin Bit*, as it is not encrypted. Similarly, depending on its local user policies, a host could accept or deny a plugin accessing the TLS state.[1]

Managing plugin memory. Pluglets might need to keep persistent data across calls. Therefore, we provide functions to allocate and free memory in the plugin dedicated area. Our framework dedicates a fixed-size memory area split into constant size blocks [Ken12]. Such approach provides algorithmic $\Theta(1)$ time memory allocation while limiting fragmentation.

Retrieving data shared by pluglets. Pluglets from the same plugin might need to access a common data structure. Pluglets can assign an identifier to a given plugin memory area enabling them to retrieve and modify it consistently.

Modifying connection memory area. Plugins might need to modify memory outside the PRE. For instance, a pluglet might need to write a new frame inside a buffer. The API controls the plugin operations by checking the accessed memory areas.

[1] We do not currently expose TLS keys to plugins.

Calling other protocol operations. This is required when a protocol operation depends on another one. However, such capability raises potential safety issues. As plugins can call any protocol operation, a PQUIC implementation needs to take care of possible loops due to these calls. To prevent such loops, the call graph of the protocol operations must always remain loop-free. However, ensuring this property for any combination of loop-free plugins is not practical to assess before executing them due to the combinatorial state explosion. Consider the example shown in Fig. 3.3. There are three protocol operations A, B and C, all guaranteed to terminate. Even if both $p1$ and $p2$ plugins are legitimate, their combination might introduce an infinite loop, as shown in Figure 3.3d. To avoid this situation, a PQUIC implementation keeps track of all the currently running protocol operations in the call stack. If a call is requested for an operation that is already running, PQUIC stops the connection and raises an error.

Scheduling the transmission of QUIC frames. PQUIC provides a way for pluglets to reserve a slot for sending frames, whether they define a new frame or use an existing type. However, it should enforce two rules. First, plugins must not prevent PQUIC from sending application data. Therefore, as long as there is payload data to be sent, standard QUIC frames such as STREAM, ACK and MAX_DATA should have a guaranteed fraction of the available congestion window. Second, a plugin sending many large frames should not be able to starve other plugins. Concurrently active plugins should have a potentially fair share of the sending congestion window. To achieve this, PQUIC includes a frame scheduler which is a combination of class-based queuing [FJ95] and deficit round-robin [SV96]. Frames are classified based on their origin, either from the core implementation itself or from plugins. When both classes are pushing frames, the scheduler ensures that the core ones get at least a configured x% of the available congestion window. A deficit round-robin then distributes the remaining budget between the plugin frames.

3.1.4 Interacting with Applications

We showed how plugins can interact within PQUIC. Plugins can also interact with the application using PQUIC. This allows them to extend the application-facing interface of PQUIC to bring new functionalities. For example, a plugin could implement a message mode for QUIC, such as the one provided by the DATAGRAM frame [RFC9221], to supplement the standardized ordered byte-stream abstraction [RFC9000]. This communication is established in a per-plugin bidirectional manner. First, an application can call *external* protocol operations. These are new anchor points that can be defined when injecting pluglets. The *external* mode is similar to the *replace* mode, but it makes the protocol operation only executable by the application. This allows it to

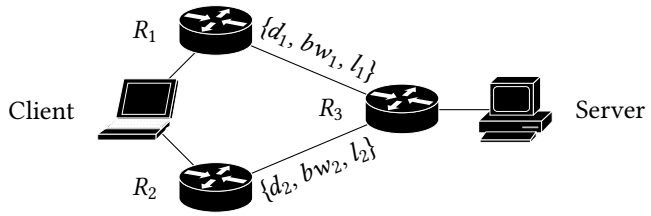


Figure 3.4: Network topology used for experiments.

Plugin	LoC	Pluglets	Proven terminating	ELF Size	Compressed size
Monitoring	500	14	13	86 kB	27 kB
Datagram	500	11	8	28 kB	25 kB

Table 3.2: Statistics for each implemented plugin.

directly invoke new methods, e.g., queuing a message to be sent. Second, a plugin can asynchronously push messages back to the application, so that it remains independent of the application control flow.

3.2 Use Cases

Protocol plugins can be used to implement various extensions to PQUIC. With less than 100 lines of C code a PQUIC plugin can add the equivalent of Tail Loss Probe in TCP [Fla+13], or support for Explicit Congestion Notification [Wes18]. This section focuses on more complex use cases that demonstrate the extensibility of PQUIC with four very different protocol plugins. Our first plugin (Section 3.2.1) adds new monitoring capabilities to PQUIC. This is a relatively simple plugin that collects statistics by reading state variables. It provides similar features as Web100 [MHR03], the MIB-2 or the TCP_INFO socket options for TCP. Our second plugin (Section 3.2.2) extends PQUIC to support unreliable messages [PKS18]. We use it to implement a VPN service that is similar to the ones using DTLS [MR04]. This demonstrates the possibility of extending the PQUIC interface proposed to the application.

We evaluate the performance of these four plugins in a lab equipped with Intel Xeon X3440 processors, 16 GB of RAM and 1 Gbps NIC, running Linux kernel 4.19 and configured as shown in Figure 3.4. The links between R_1, R_3 and R_2, R_3 are configured using *NetEm* [Hem05] to add transmission delays and using *HTB* to limit their bandwidth. Losses are generated using a seeded random loss generator attached to the routers. This allows fair performance comparisons as the same loss pattern is applied when an experiment is replayed. One-way delay d is expressed in milliseconds, bandwidth bw in Mbits and uniform losses l as a percentage of packets transmitted.

To evaluate the plugins in a wide range of environments, we use the experimental design approach [Fis35]. We define ranges on the possible values for the parameters presented and use the WSP algorithm [SCS12] to broadly sample this parameter space into 139 points. Each parameter combination is run 9 times and the median run is reported. This mitigates a possible bias in parameter selection and gives a general confidence in the experiment results. Unless otherwise noted, we use the parameters range $\{d_1 \in [2.5, 25]\text{msec}, bw_1 \in [5, 50]\text{Mbps}, l_1 = 0, d_2 = d_1, bw_2 = bw_1, l_2 = l_1\}$, and assume that both links have similar bandwidth, delay and loss characteristics. Note that when links are lossless, congestion-induced losses can still be observed due to the limited bandwidth and router buffers. All plugins are cached on both client and server.

3.2.1 Monitoring PQUIC

QUIC is an encrypted protocol that leaves only a small fraction of its headers in cleartext, preventing third-party flow performance analysis for events such as losses and retransmissions [De +18; Ste+17]. The QUIC working group has reached consensus on disclosing a single bit in the header, called the *Spin Bit* [TK18], to enable third parties to measure the RTT of a connection, but no solution is proposed for other metrics such as retransmissions, losses, etc.

Design. Our monitoring plugin adds passive pluglets, i.e. pluglets that hook to *pre* and *post* anchors, to several protocol operations in PQUIC to record the performance indicators (PI) such as the bytes/packets sent/received, lost, received out-of-order, etc. A set of PIs are recorded during the handshake and a second are updated while the connection is active. Our plugin exports these PIs to a local daemon that sends them over UDP to a collector. A similar approach could be used to feed an SNMP agent that maintains a QUIC MIB.

Implementation. Each pluglet is hooked to a particular step of the connection. Upon the arrival or transmission of a packet, the relevant PIs are updated. Once a set of PI is complete, either because the connection was established or terminated, it is sent to the local daemon.

3.2.2 QUIC VPN

Given the security of QUIC, it could be interesting to use it as a replacement for TLS-based VPNs. Experience with such VPNs indicate that DTLS [MR04] provides a better user experience that using TLS over TCP because of the well-known TCP-over-TCP performance problems [LX17; Hon+05].

We leverage the flexibility of PQUIC with a plugin that supports unreliable messages in addition to the reliable QUIC bytestreams. This plugin supports a new DATAGRAM frame [PKS18] that only maintains the transported data boundaries but not transmission order nor reliable delivery.

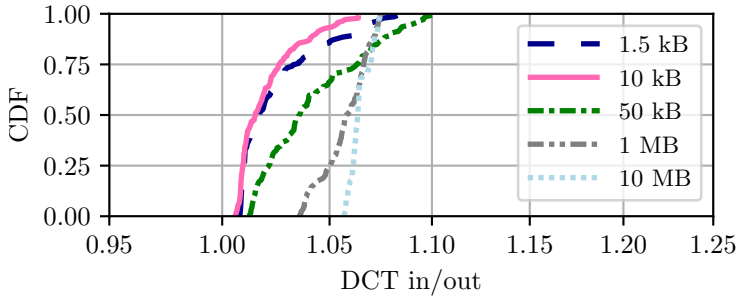


Figure 3.5: DCT ratio of TCP in and outside a client-server PQUIC tunnel.

Design. We implement a simple VPN that captures raw IP packets and passes them to PQUIC. We chose to encapsulate the user traffic at the IP level because this allows handling of all types of packets.

We modified the PQUIC client and server to carry IP datagrams between Linux tunnel interfaces. This was done by adding 70 lines, modifying 10 lines and removing 200 lines in the `picoquic` sample applications. This VPN application leverages a key point of our design presented in Section 3.1.4, i.e. a plugin is also able to extend the API exposed to the application. Our VPN application reads IP datagrams from the tunnel interface and writes them to the message socket exposed by the Datagram plugin. It reads the received IP datagrams from the message socket and writes them to the tunnel interface. This simplistic approach and the default PQUIC parameters lead to an overhead of 44 bytes per conveyed IP datagram when used over IPv4. These are caused by the IP, UDP and QUIC headers used within the QUIC connection in addition to the small header of the DATAGRAM frame. This overhead could be minimized by using techniques such as header compression on the conveyed IP packets, but these are outside the scope of this chapter.

Evaluation. We evaluate the overhead introduced by our VPN in terms of Download Completion Time (DCT) for a single file transfer using $\text{TCP}_{\text{Cubic}}$. We compare the file transfer times inside and outside the VPN tunnel for different file sizes. We only use the top path of Figure 3.4 and explore the default parameters range. We used a 1400-byte MTU inside the tunnel and 1500 outside. The VPN overhead of 44 bytes per conveyed packet translates into a bound on the DCT ratio of 1.031. Figure 3.5 illustrates the CDF of the DCT ratio obtained. For short files, the DCT is mostly below the bound computed. When transferring small amounts of data, the network path buffers and congestion windows can accommodate the VPN encapsulation overhead and only the VPN processing time affects the DCT. For longer files, the DCT ratio is more stable and mainly caused by the per-packet overhead.

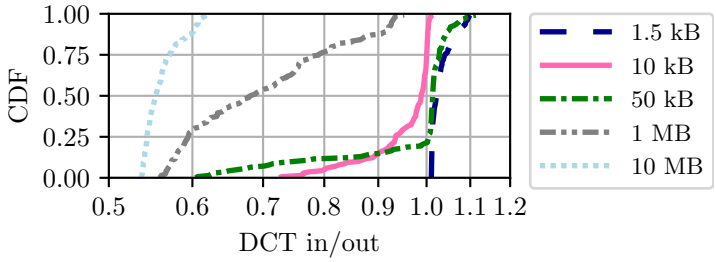


Figure 3.6: DCT ratio of TCP in and outside a multipath client-server tunnel.

3.2.3 QUIC Multipath VPN

Our PQUIC plugins provide a range of other extensions to the protocol, such as a Multipath and a FEC plugin that were described by their contributors in other theses [De 20; Mic23]. Furthermore, given the isolation provided by PQUIC, it is possible to load different plugins on a given PQUIC implementation provided that they do not replace the same protocol operation. All the plugins discussed in this section have orthogonal features. They can be combined to provide more advanced services. As an example of the flexibility of our approach, we demonstrate how the Multipath plugin can be injected together with the Datagram plugin.

We evaluate the performance of combining these two plugins in a setting akin to the evaluation of the Datagram plugin but with the two paths of Figure 3.4 and by exploring the default parameters range. We measure the ratio of Download Completion Times (DCT) when running a single $\text{TCP}_{\text{CUBIC}}$ file transfer inside and outside the multipath VPN tunnel for different file sizes.

As expected, we do not observe any benefit in using multipath for tunnelling a short TCP transfer in our setup. If one of the two paths was to exhibit a lower delay, the Lowest-RTT scheduler would improve the DCT by selecting the fastest path. However, as file size grows the benefits of multipath become clear. By spreading the traffic over the two symmetric paths, our combined plugins reach a DCT ratio that tends to 0.55 as illustrated in Figure 3.6. In this case, the tunnelled TCP CUBIC transfer grows its congestion window such that it sends as much traffic as offered by the two links together.

3.3 Conclusion

In this chapter, we revisited the paradigm for extending transport protocols by proposing the concept of a *pluginised protocol*, in which its operations can be extended or replaced by eBPF codes called *plugins*. We applied this approach to the QUIC protocol, leveraging its security features and improved extensibility through QUIC *Frames* and *Transport Parameters*. These plug-

ins can be injected locally by an application requiring a greater control on a given transport mechanism or willing to extend the transport services offered by a given QUIC implementation. In addition, these plugins can be securely exchanged over the QUIC connection, such that the server can request the client to securely execute eBPF code extending its capabilities. This greatly improves the deployability problem that all Internet protocols are facing.

We implemented this approach in a QUIC implementation along with two plugins, one that monitors performance metrics of a given connection and another that implements the DATAGRAM frame. We evaluate their overhead on a given connection and also implement a simple VPN leveraging the DATAGRAM frame to convey IP packets over a QUIC connection. We show the strength of our approach by combining this plugin with the Multipath plugin, such that the VPN can leverage several paths. We demonstrate that a TCP connection inside the multipath VPN can aggregate the bandwidth of two symmetric paths.

TCPLS: Modern Transport Services with TCP and TLS

4

Following our experience with the QUIC protocol in the past two chapters, we revisit in this chapter the way TCP and TLS are combined to securely extend the transport services they offer to include multiplexing, connection migration and multipath capabilities. As TCP remains the fallback for QUIC and is expected to remain in use by many applications in the future, we argue there is an interest in reconsidering how its use can be improved. To do so, we joined the efforts around the TCPLS protocol, which first introduced the idea of revisiting the layering between TCP and TLS in 2020 [RAB20], and then expanded in 2021 [Roc+21]. These research works were based on a first implementation of TCPLS named `picotcpls` [RAP22] that extends the `picotls` TLS 1.3 implementation written in the C language. The first article precedes our involvement in TCPLS as we joined the effort during the writing of the second article to which we contributed most of the text[1]. We also contributed minor bug fixes to `picotcpls`. These two articles are based on a first version of the protocol that we describe in detail in this subsection.

In this chapter, we first describe the original version of the TCPLS protocol in Section 4.1. From our analysis of this first version, we identify seven limitations arising from its design. In Section 4.2, we propose a new version of the TCPLS protocol which departs in several ways from the original to address all seven limitations. Our new version, TCPLS v2, retains all the possibilities offered by the first version while being simpler to implement. We develop a reference implementation for this new version that we present in Section 4.3 and evaluate in Section 4.4. We conduct experiments using the multiplexing and multipath features of our second version. They demonstrate how the modern transport services brought by TCPLS v2 can be leveraged today.

4.1 TCPLS v1

We highlight the limitations of TCPLS v1 and draw a number of conclusions in this section to design our second version described later in this chapter. To

[1] LaTeX sources available are at `https://github.com/frochet/tcpls_conext21`.

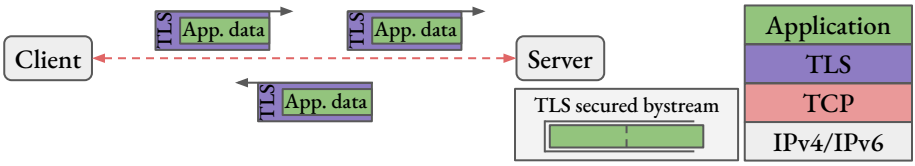


Figure 4.1: TLS provides a single secured bytestream service to the application using TLS records and a single TCP connection.

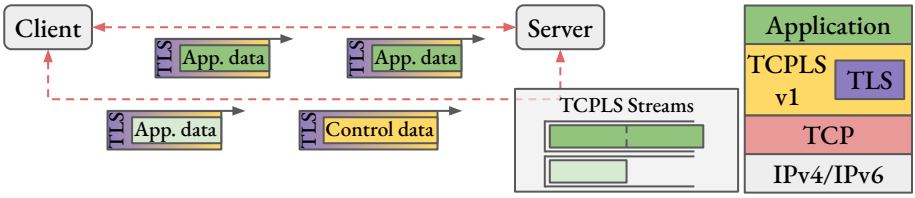


Figure 4.2: TCPLS v1 provides several TCPLS Streams, connection migration and multipath by extending the TLS record layer and joining TCP connections.

our knowledge, this level of details on this first version and its limitations was not published before and is crucial in understanding of our second version.

Figures 4.1 and 4.2 compare TLS to TCPLS v1. TLS offers a single secured bytestream to the application for which it leverages TLS records to protect and convey it. At its core, the first version of TCPLS extends the TLS record layer to bring services such as multiplexing, connection migration and multipath to the application. It also extends the control information that can be exchanged inside protected TLS records such that these new services can be offered. The TLS record layer is usually not exposed to applications as TLS only offers a secured bytestream as a service to applications. Implementing TCPLS v1 is thus very tied to the TLS library that will be chosen as a base. **Limitation 1: TCPLS v1 requires implementers to modify a substantial portion of a TLS implementation**. To achieve connection migration and multipath, TCPLS v1 leverages several TCP connections that can be joined in the same TCPLS session. The TLS session is shared among TCP connections such that TLS records can be spread for failover or bandwidth aggregation purposes.

4.1.1 TCPLS v1 records

TCPLS v1 extends the TLS record layer with new types of TLS records. Figure 4.3 compares an encrypted TLS 1.3 record to a TCPLS v1 TLS record. Both have identical fields in the non-encrypted part, as indicated by fields with a white background. They appear as AppData records to third-party observers. This ensures that TLS middleboxes will not interfere with these records. How-

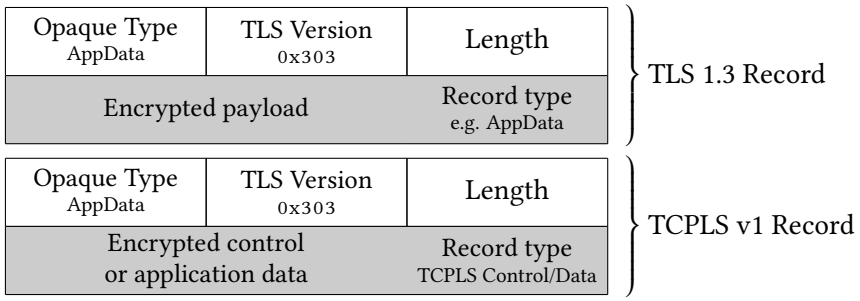
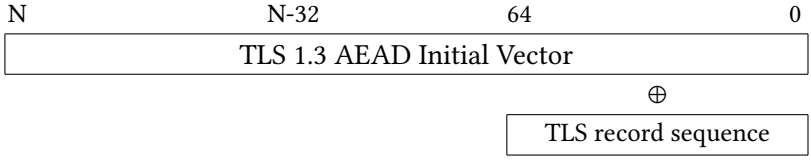


Figure 4.3: TCPLS v1 extends the encrypted TLS records to convey control and application data.

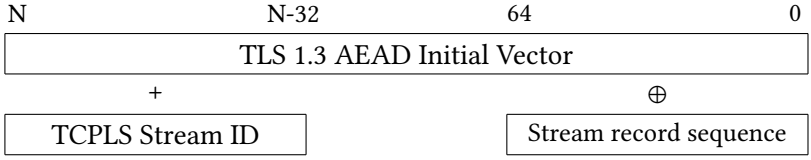
ever, in the encrypted part indicated by a gray background, TCPLS v1 defines two new record types, TCPLS Control and TCPLS Data, to convey additional control information and the application data. TCPLS Control records have a control type defining the type of control information conveyed. For instance, the record can signal the use of a TCPLS Stream, convey TCPLS Stream acknowledgements, alert from the failure of a TCP connection, securely exchange TCP Options such as the TCP User Timeout Option [RFC5482] and eBPF code that can be injected on a given TCP connection of the TCPLS session. These records are protected using the same AEAD construction as for TLS 1.3 records, which is illustrated in the top part of Figure 4.4.

When joining several TCP connections in a TCPLS session, for instance to perform migration from one connection to another or to use the two simultaneously using the multipath capabilities of TCPLS, the TLS cryptographic material derived from the TLS handshake is shared among all TCP connections to protect the TCPLS Control records exchanged. In addition, the TLS record sequence is shared and follows the transmission order of TCPLS Control records. From the receiver point of view, as the TCP connections part of TCPLS session evolve independently, the correct receipt order of these records cannot be guaranteed. **Limitation 2: TCPLS v1 receivers are not guaranteed to decrypt TCPLS Control records when using several TCP connections, as they are received and may need to guess the TLS record sequence in successive trials**.

TCPLS Data records are used to exchange chunks of application data. These chunks carry no explicit metadata on the TCPLS stream they are part of nor on their offset within this stream. To protect these records, the AEAD nonce construction is updated to include the TCPLS Stream ID as illustrated in Figure 4.4. This ID cannot be 0 to prevent nonce reuse when protecting TCPLS v1 Control records. As this Stream ID is not available to the receiver, it has to guess its value in potentially several AEAD decryption trials. The receiver can reasonably discern whether its guess is correct given the authentication



TLS 1.3 AEAD Nonce construction



TCPLS v1 AEAD Nonce construction

Figure 4.4: The AEAD Nonce of TCPLS Streams updates the TLS 1.3 construct.

properties of AEAD. The TCPLS v1 protocol helps the receiver to narrow the set of Stream IDs to try decryption for by explicitly attaching TCPLS Streams to a given TCP connection using a dedicated TCPLS Control record. Several heuristics can also be used by a receiver, such as trying the last Stream ID first, to speed up this process. The AEAD construct can theoretically help this process as the authentication tag can be checked separately from performing a full decryption with less intensive computations. However, we inspected the OpenSSL code for AES GCM and report that it performs decryption then authentication[2]. We suppose that this choice is made to prevent timing attacks and is likely to be common across TLS implementations. **Limitation 3: TCPLS v1 receivers may need to perform several decryption trials to process TCPLS Data records**.

In addition to the TCPLS Stream ID, the Stream record sequence is also implicit in the AEAD Nonce construct. This prevents a sender to send data from a TCPLS Stream over several TCP connections without making the receiver face a similar problem as with TCPLS Control records. When combined with the guessing of Stream IDs, this problem is likely to become unpractical in a number of cases. **Limitation 4: A TCPLS v1 Stream cannot be steered or split between several TCP connections**.

Given that the AEAD Nonce is constructed differently for each TCPLS stream, and TCPLS Control records use a different one, several IVs can be used for a single TCP connection part of the TCPLS session. However, in-kernel TLS offload expects a single IV per TCP connection[3]. **Limitation 5:**

[2]AES GCM decryption code in OpenSSL can be found at `https://github.com/openssl/openssl/blob/b646179/crypto/evp/e_aes.c#L2956-L3001`

[3]`https://docs.kernel.org/networking/tls-offload.html#normal-operation`

TCPLS v1 cannot leverage Kernel TLS Offload.

While the AEAD Nonce construction has a number of limitations we detailed, it was designed to enable a zero-copy receive path under a number of assumptions, i.e., the limitations aforementioned. When a receiver decrypts a TCPLS record, it can provide the TCPLS Stream receive buffer, potentially passed by the application itself, as the decryption output buffer. When the decrypted record is the next expected TCPLS Data record, its decrypted data is appended to the buffer during the decryption operation at zero extra cost.

4.1.2 Joining TCP connections

To provide connection migration, for instance in the case of the failure of a TCP connection or as a result of user mobility, TCPLS v1 enables several TCP connections to be joined in a TCPLS session. To do so, the server provides *TCPLS session cookies* to the client after the TLS handshake. After having established an additional TCP connection, the client can use one of them during the TLS handshake inside a dedicated *TCPLS Join* TLS extension. TCPLS v1 decouples session identifiers from cookie values which has the unfortunate effect of making observers able to link TCP connections to a given session. However, this could be simply fixed by using unique cookie values across TCPLS sessions of a server. **Limitation 6: TCPLS v1 enables observers to link TCP connections during their TLS handshake**.

When TCPLS Data records are sent over a TCP connection, they elicit acknowledgements from the receiver in the form of TCPLS Control records containing *Data acknowledgements*. This enables a sender to discern unacknowledged Data records when a TCP connection is failing such that these records can be retransmitted on other connections part of the TCPLS session. This requires an explicit TCPLS Stream record sequence synchronisation to avoid problems identified previously. Given that records are protected separately from the TCP connection they are exchanged over, TCPLS v1 enables the sender to simply replay the *ciphertext* of these records. However, these identical transmissions of TLS records can be observed, enabling an observer to correlate TCP connections part of a TCPLS session. An active attacker could inject a TCP RST segment to trigger these retransmissions. **Limitation 7: TCPLS v1 enables observers to link TCP connections together by observing replayed TLS records exchanged over them**.

4.1.3 Multipath

To provide the stream steering and splitting features to applications, given the limitations of TCPLS Streams as designed in TCPLS v1, an additional layer named *Coupled Streams* was introduced. It adds a per-Coupled Stream record sequence inside Data records such they can be reordered before being copied

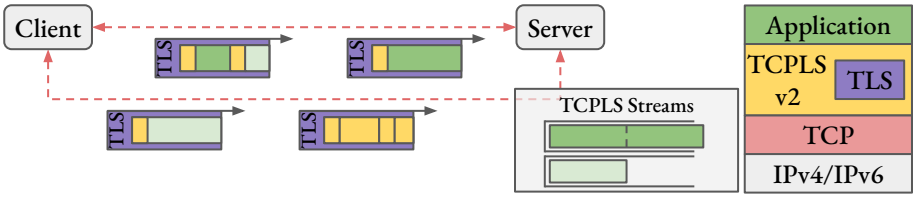


Figure 4.5: TCPLS v2 introduces framing inside TLS records to bring TCPLS Streams, connection migration and multipath. Regular TLS records are used and protected using a per-TCP connection variant of the IV.

into the required application receive buffer. Several TCPLS Streams can thus be part of the same Coupled Stream. We note that an offset field instead of a sequence number enables records to be copied at the required offset without any sorting. In practical terms, applications are likely to systematically use this construct for exchanging any application object, as TCPLS Streams alone can only be switched to another TCP connection when the connection they are attached to is failing. In the case of an under-performing TCP connection, e.g., due to temporary congestion, an application cannot temporarily transmit TLS records redundantly on other connections without having initiated a Coupled Stream at the very start of the application object transfer.

4.2 TCPLS v2

Based on our study of the first version of the TCPLS protocol, we designed a second version solving the limitations we identified while simplifying the protocol and its implementation. Our new version of the protocol, as illustrated on Figure 4.5, makes two important changes to the paradigm of TCPLS v1. First, it defines its own framing inside TLS records instead of extending them. Second, it establishes the AEAD nonce construction at the connection level instead of at the TCPLS Stream level as described earlier in Section 4.1.1 and Figure 4.3. We first describe the protocol in more details before demonstrating how TCPLS v2 tackles each limitation highlighted in Section 4.1. We also provide a detailed specification of our protocol as an IETF Internet-Draft [PRB23].

4.2.1 Establishing a TCPLS session

To establish a TCPLS session, the client initiates a TLS handshake as illustrated on Figure 4.6. To indicate its support and willingness of using TCPLS, it places the `tcpls` TLS extension in its ClientHello. The server acknowledges the creation of the TCPLS session by placing the `tcpls` extension within its EncryptedExtensions. After the TLS handshake is finished, the TCPLS session can be used.

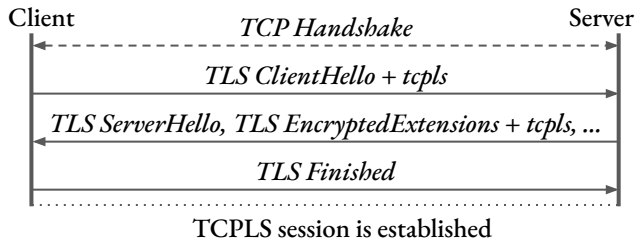


Figure 4.6: A TCPLS v2 session is established during the TLS handshake using the *tcpls* TLS Extension.

4.2.2 TCPLS Frames

A TCPLS frame is the basic unit of information for TCPLS. It is a *Type-Length-Value* tuple that always fits in a single TLS record. One or more TCPLS frames can be encoded in a TLS record. This TLS record is then reliably transported by one TCP connection. Compared to TCPLS v1, this alleviates the need to modify a TLS implementation to implement most of the TCPLS features. The only requirement our protocol imposes is that a TCPLS frame cannot be split into several TLS records. We establish this requirement to stress the importance of adequately sizing the TLS records as encryption and decryption operations are applied on them as a whole. A set of TCPLS frames within a TLS record are thus received all together in all circumstances. The TLS library is thus required to offer a tight control over the TLS record boundaries it produces when passing a set of TCPLS frames.

4.2.3 Multiple Streams

TCPLS extends the service provided by TCP with streams. Streams are independent bidirectional bytestreams that can be used by applications to concurrently convey several objects over a TCPLS session. Streams can be opened by the client and by the server.

Streams are identified by a 32-bit unsigned integer. The parity of this number indicates the initiator of the stream. The client opens even-numbered streams while the server opens odd-numbered streams. Streams are opened in sequence, e.g., a client that has opened stream 0 will use stream 2 as the next one. Each Stream frame carries a chunk of application data of a given stream. Applications can mark the end of a stream to close it. The QUIC protocol offers 2^{62} streams that can be unidirectional. A revision of TCPLS v2 could broaden the specification of streams to provide an equivalent service to applications.

Similarly to HTTP/2 [RFC7540], conveying several streams on a single TCP connection introduces Head-of-Line (HoL) blocking between the streams.

To alleviate this, TCPLS provides means to the application to choose the degree of HoL blocking resilience it needs for its application objects by spreading streams among different underlying TCP connections.

4.2.4 Multiple TCP connections

TCPLS is not restricted to using a single TCP connection to exchange frames. A TCPLS session starts with the TCP connection that was used to transport the TLS handshake. After this handshake, other TCP connections can be added to a TCPLS session, either to spread the load or for failover purposes. TCPLS manages the utilisation of the underlying TCP connections within a TCPLS session.

Multipath TCP enables both the client and the server to establish additional TCP connections. However, experience has shown that additional subflows are often only established by the clients. TCPLS focuses on this deployment and only allows clients to create additional TCP connections.

Using Multipath TCP, a client can try establishing a new TCP connection at any time. If a server wishes to restrict the number of TCP connections that correspond to one Multipath TCP connection, it has to respond with RST to the in excess connection attempts.

TCPLS takes another approach. To control the number of connections that a client can establish, a TCPLS server supplies unique tokens. A client includes one token when it attaches a new TCP connection to a TCPLS session. Each token can only be used once. This can be compared to the *maximum path identifier* mechanism of MPQUIC which gives comparable abilities to the server.

TCPLS endpoints can advertise their local addresses, allowing new TCP connections for a given TCPLS session to be established between new pairs of addresses. When an endpoint is no more willing new TCP connections to use one of its advertised addresses, it can remove this address from the TCPLS session. QUIC and MPQUIC do not support the exchange of new addresses on a connection. We proposed an extension to tackle this problem [PB24b].

4.2.5 Record protection

When adding new TCP connections to a TCPLS session, an endpoint does not complete the TLS handshake. TCPLS provides a AEAD nonce construction for TLS record protection that is used for all connections of a session. This reduces the cryptographic cost of adding connections. To avoid interference by middleboxes, endpoints can send factice TLS messages to form an apparent complete TLS handshake to middleboxes.

In order to use the TLS session over multiple connections, TCPLS adds a record sequence number space per connection that is maintained indepen-

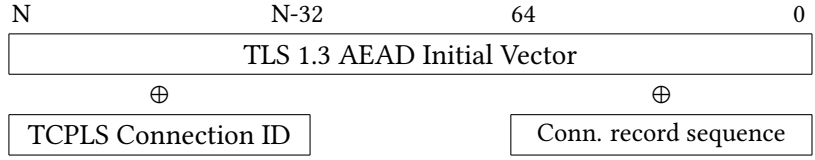


Figure 4.7: TCPLS v2 uses a per-connection AEAD Nonce based on the TLS 1.3 construct.

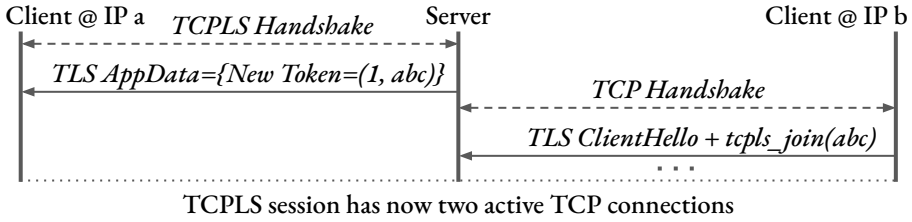


Figure 4.8: A client can join additional TCP connections using tokens provided by the server.

dently at both sides. Each record sent over a TCPLS session is identified by the Connection ID of its connection and its record sequence number. Each record nonce is constructed as defined in Figure 4.7.

This construction guarantees that every TLS record sent over the TCPLS session is protected with a unique nonce. As in TLS 1.3, the per-connection record sequence is implicit. However, given that it is maintained per TCP connection and that TLS record on a given connection are delivered in order, this record sequence is never ambiguous to the receiver.

4.2.6 Joining TCP connections

The TCPLS server provides tokens to the client in order to join new TCP connections to the TCPLS session. Figure 4.8 illustrates a client and server first establishing a new TCPLS session as described in Section 4.2.1. Then the server sends a token over this connection using the New Token frame. Each token has a sequence number (e.g., 1) and a value (e.g. "abc"). The client uses this token to open a new TCP connection and initiates the TCPLS handshake. It adds the token inside the `tcpls_join` TLS extension in the ClientHello.

When receiving a TCPLS Join Extension, the server validates the token and associates the TCP connection to the TCPLS session.

Each TCP connection that is part of a TCPLS session is identified by a unique 32-bit unsigned integer called its Connection ID. The first TCP connection of a session corresponds to Connection ID 0. When joining a new connection, the sequence number of the token, i.e., 1 in our example, becomes

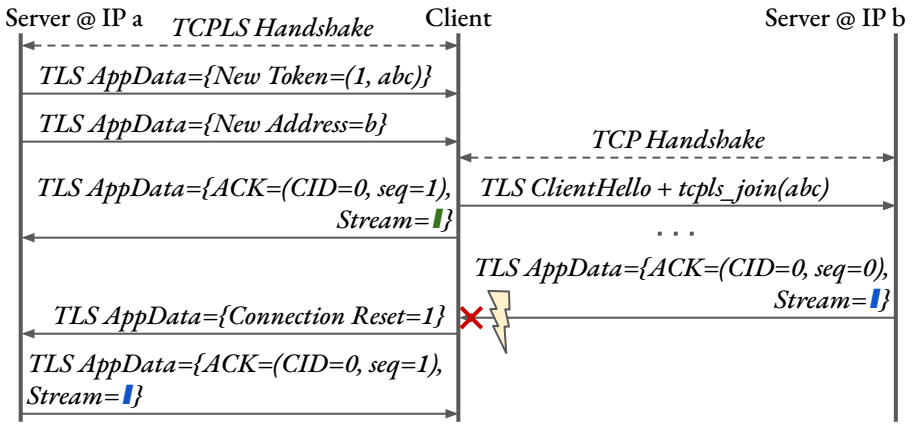


Figure 4.9: Example of TCPLS flow with two TCP connections exchanging several records and frames.

the Connection ID of the connection. The Connection ID enables the Client and the Server to identify a specific TCP connection within a given TCPLS session.

4.2.7 TCPLS Frames details

We defined 8 types of TCPLS frames. They convey application data, acknowledge received records, enable clients to add additional TCP connections, inform of new IP addresses and ports and alert of failing TCP connections. We summarise each of them in the rest of this subsection.

ACK frame. This frame is sent by the receiver to acknowledge the receipt of TLS records on a particular TCP connection of the TCPLS session. It identifies the TCP connection for which the acknowledgement was sent using its Connection ID and the highest record sequence received on that connection. Although the reliability of the data exchange on a connection is handled by TCP, there are situations such as the failure of a TCP connection where a sender does not know whether the TLS frames that it sent have been correctly received by the peer. An ACK frame can be sent over any TCP connection of a TCPLS session, not only on the one it refers to.

Ping frame. This frame is used to elicit an acknowledgement from its peer. It has no content. When an endpoint receives a Ping frame, it acknowledges the TLS record that contains this frame. This frame can be used by an endpoint to check that its peer can receive TLS records over a particular TCP connection.

Stream frame. This frame is used to carry chunks of data of a given stream. It identifies the stream, the chunk length and the chunk offset within this stream. A flag indicates the graceful closure of the sending direction by

the sender of the frame.

New Token frame. This frame is used by the server to provide tokens to the client. Each token can be used to join a new TCP connection to the TCPLS session, as described in Section 4.2.6. Clients cannot send New Token frames. By controlling the amount of tokens given to the client, the server can control the number of active TCP connections of a TCPLS session. The server should replenish the tokens when TCP connections are removed from the TCPLS session.

Connection Reset frame. This frame is used to alert that the receiving side of a TCP connection was closed. This is useful to inform the sending side when it cannot be processed any more by the receiver such that TLS records queued on this connection can be retransmitted on other TCP connections part of the TCPLS session. This frame identifies the TCP connection using its Connection ID such that it can be sent on any TCP connection of the session.

New Address frame. This frame is used by an endpoint to add a new local address to the TCPLS session. This address can then be used to establish new TCP connections. The server advertises addresses that the client can use as destination when adding TCP connections. The client advertises addresses that it can use as source when adding TCP connections.

Remove Address frame. This frame is used by an endpoint to announce that it is not willing to use a given address to establish new TCP connections. After receiving this frame, a client cannot establish new TCP connections to the given address.

An example flow of TCPLS is illustrated on Figure 4.9. In our example, a client initiates a TCPLS session towards a multihomed server. The server provides a token along with an indication of its second address. The client then starts a second TCP connection towards the server. On the first connection, the client acknowledges the received records and sends a chunk of stream data. The server answers on the second connection but unfortunately this TLS record not delivered and the connection is reset from the client perspective. In order to speed up the recovery, the client alerts the server using the Connection Reset frame. Finally, the server retransmits the lost Stream frame over the first connection.

4.2.8 Addressing limitations of TCPLS v1

In this subsection we summarise how TCPLS v2 addresses each of the seven limitations we identified in Section 4.1.

Limitation 1: TCPLS v1 requires implementers to modify a substantial portion of a TLS implementation. Our protocol significantly reduces the interactions between TCPLS and TLS while enabling modern transport services. It requires the TLS implementation to: (i) retrieve and set the

IV according to the TCP connection used; (*ii*) send and receive the `tcpls` and `tcpls_join` TLS extensions; (*iii*) prevent TCPLS frames fragmentation across TLS records. These changes are of a much smaller order of magnitude than extending the TLS record layer itself. In Section 4.3, we compare how these changes translate into the implementation in terms of code length.

Limitation 2: TCPLS v1 receivers are not guaranteed to decrypt TCPLS Control records when using several TCP connections, as they are received and may need to guess the TLS record sequence in successive trials. **Limitation 3: TCPLS v1 receivers may need to perform several decryption trials to process TCPLS Data records**. **Limitation 4: A TCPLS v1 Stream cannot be steered or split between several TCP connections**. These three limitations are lifted by our method to protect TLS records. By using a per-connection AEAD Nonce construction combined with the fact that TCP delivers protected TLS record in sequence, none of these problems arise within TCPLS v2.

Limitation 5: TCPLS v1 cannot leverage Kernel TLS Offload. Our design make a step towards enabling TLS offload by defining a per-connection IV and record sequence. There remains yet some experimentation work to be done to confirm that TLS records boundaries can be respected when sending and receiving. However, if it is not available, implementing this mechanism is of a smaller order of magnitude than the TCPLS v1 method for protecting records.

Limitation 6: TCPLS v1 enables observers to link TCP connections during their TLS handshake. TCPLS v2 Tokens are opaque values used to join additional TCP connections. They are sent by the server inside protected TLS records. Each token can only be used once and bear no similarity.

Limitation 7: TCPLS v1 enables observers to link TCP connections together by observing replayed TLS records exchanged over them. When a TCP connection is failing such that the queued TLS records cannot be delivered, their frames are retransmitted in new TLS records conveyed on other TCP connections.

In addition, the particular design of TCPLS v1 enables two features that are emphasised in its two articles [RAB20; Roc+21]. The first is the processing of TCPLS streams from a given session over multiple cores, which is claimed to arise from the per-stream AEAD Nonce construction. We argue that any TLS session can already be processed by several cores concurrently. As the TLS record sequence is implicit but non-ambiguous in TLS 1.3, any received record can be decrypted concurrently from others. This property is retained by TCPLS v2, as the AEAD Nonce construction is established per TCP connection. In addition, as TCPLS Stream frames identify the offset within the stream their chunk of data belongs to, the received data can be correctly placed in the application buffer at all times.

The second feature is the zero-copy receive path offered by TLS 1.3 and maintained by TCPLS v1. We note that while the TCPLS v2 protocol as presented in this chapter does not maintain this receive path, it can be re-established with a low complexity. By reversing the order of TCPLS frames, such that they appear as *Value-Length-Type*, and by placing a single TCPLS Stream frame at the start of the TLS record, a receiver can achieve a zero-copy receive path under the same assumptions as TCPLS v1. When the Stream frame size is known in advance, a multicore zero-copy receive path can be achieved.

As such, we believe that our TCPLS v2 protocol addresses all limitations of the first version while retaining all its transport services and properties.

4.3 Prototype

We implemented TCPLS v2 in the C language in a dedicated implementation built atop the `picotls` [OHc24] TLS library. We chose to develop a separate implementation but using the same TLS library rather than re-architect and modifying the reference implementation of TCPLS v1 to enable a better comparison.

Using 3900 single lines of C code, we were able to implement all the features presented in Section 4.2 along with a test client and server and a unit test suite. Only about 30 lines modify the TLS library to retrieve and set the IV. `picotls` already supports the exchange of TLS extensions defined outside the library as well as enforcing the boundaries of TLS records. In contrast, the reference implementation of TCPLS v1 spans 7700 single lines of C code.

4.3.1 Integrating TCPLS to web applications

In order to evaluate our TCPLS implementation, we integrated it to two web applications, a client and a web server, that we release as open source. For the client side, we chose `wrk`, an HTTP benchmarking tool[4]. It is a popular tool that can emulate many clients making repeated requests in a controlled manner. It collects the request completion times and computes statistics. We added 260 lines of code and modified 40 lines to implement the support of TCPLS within `wrk`[5]. The client is capable of establishing TCPLS sessions to perform one or more requests over several TCP connections within each session.

For the server side, we chose `h2o`, an HTTP server supporting all versions of HTTP. We chose this server as it uses `picotls` to secure connections with TLS 1.3. Our TCPLS implementation being atop `picotls`, it enabled us to

[4] `https://github.com/mpiraux/wrk-tcpls`

[5] `https://github.com/mpiraux/wrk-tcpls/compare/894f3798...rapido`

modify h2o to support TCPLS more easily. Our modifications required a similarly low amount of code, i.e., 350 added lines and 50 modified lines[6]. The h2o server is capable of accepting TCPLS sessions with several TCP connections. In this project, we focused on integrating TCPLS in h2o to support HTTP/1.1. While HTTP/2 is more advanced and could better leverage the services offered with TCPLS, such as the multiplexing capability, HTTP/1.1 provides a baseline of what can be achieved with a simple application protocol.

4.4 Evaluation

We evaluate the TCPLS v2 protocol and our prototype with two experiments involving its different features in different contexts. First we focus on its multiplexing capabilities when used with several TCP connections to prevent Head-of-Line blocking. This is evaluated using bare-metal servers from CloudLab [Dup+19].

Second, we demonstrate how the TCPLS path and record scheduler can help prioritise the lowest-latency TCP connection of a given TCPLS session when exchanging HTTP requests between two existing tools: wrk and h2o. This experiment is run on emulated networks using IPMininet [JTc23].

4.4.1 Multiplexing and Head-of-Line blocking

We first evaluate the ability of TCPLS v2 and our prototype to handle several TCPLS streams concurrently. To do so, we develop a small test client and server spanning 380 single lines of code. After connecting to the server and receiving TCPLS Tokens, the client establishes a configurable number of additional parallel TCP connections to the server. When all connections are established, the client starts sending chunks of 10 KB every 5 ms, each on a new TCPLS stream. The TCPLS streams are steered across the available connections in a round-robin manner. When receiving a complete TCPLS stream, the server closes its sending part of the TCPLS stream to form an applicative acknowledgement. When the client receives the stream closure, it computes the Request Completion Time (RCT) from the time it queued the stream for sending to the time the closure was received. To establish a baseline and comparison, we implement this same applicative protocol using quiche [Clo], such that we can compare how TCPLS and QUIC mitigate the Head-of-Line blocking problem in this context. This implementation spans 600 single lines of Rust code.

[6] https://github.com/mpiraux/h2o/compare/7876a2a...master

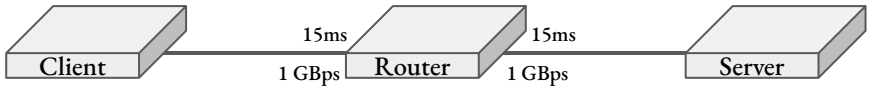


Figure 4.10: Our setup for evaluating multiplexing and head-of-line blocking resilience in TCPLS v2.

We leverage the CloudLab platform [Dup+19] to book three `sm110p`[7] computers running Linux 5.15 that we chain together. The middle computer within the chain is used as a router and applies 15 ms of delay using `tc-netem` in both directions as illustrated on Figure 4.10. When a single TCP connection is used, this will cause an average of 6 complete streams to be in-flight at all times due to the queuing delay. We also configure 0.14 % of random packet losses from the client towards the server in order to induce Head-of-Line blocking on the TCP connections. Given the stream size, frequency of streams queuing and the configured packet loss, as a single stream spans 7 TCP segments, we expect approximatively 1 % of the streams to be impacted by a loss. Conversely, 99 % of the requests are not directly affected by a packet loss. The loss percentage is also within the order of magnitude of medium-related packet loss on Starlink [Mic+22]. However, it is not bursty as our experiment does not aim at modelling losses as they would occur on the Internet but rather to trigger Head-of-Line blocking in a controlled manner. To better highlight the Head-of-Line blocking alone, we focus on its impact when using delay-based congestion controllers. Loss-based congestion controllers such as CUBIC [RFC9438] struggle to maintain the required bandwidth in the presence of random losses, as these are considered indicative of congestion in the network. In our case, as the losses are unrelated to the congestion but more analogue to medium failures, delay-based congestion controller will not overreact by reducing their congestion windows and thus better represent the sole impact of Head-of-Line blocking.

We explore the impact of the number of parallel TCP connections within the TCPLS session for the Vegas and BBR congestion controller on the RCT in the presence of these random losses. We run each combination three times for 60 s, discard the first and last 400 RCTs to leave out potential warm-up and stop side effects, and merge RCTs of a single combination into one distribution. The same experiment is performed using QUIC, as implemented by `quiche` [Clo] v0.22.0, using a single QUIC connection configured with BBR as a congestion controller. Figure 4.13 illustrates these distributions through the last 10^{th} percentiles of their Cumulative Distribution Functions and breaks them down by the number of TCP connections that were part of the TC-

[7] Hardware details are available in Section 12.2 of `https://docs.cloudlab.us/hardware.html`.

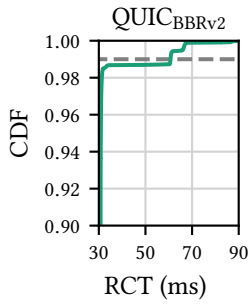


Figure 4.11: QUIC mitigates Head-of-Line blocking across QUIC streams.

PLS session. The theoretical threshold of 99 % of requests is indicated by the dashed gray line.

Let us first consider QUIC as a reference of the state of the art in terms of streams multiplexing. As depicted on Figure 4.11, the `quiche` implementation and its BBRv2 implementation exhibits a CDF with three steps. First, we note that the first step is very close to the expected proportion of requests impacted by a packet loss. For requests in the 99th percentile and above, they get delivered with an added latency to the configured RTT and are strongly clustered in two additional modes. The first one is centred around the 61 ms mark. It corresponds to the first possible criterion for QUIC to detect a packet loss, i.e., as soon as acknowledgements for 3 or more packets sent after a given packet have been received. In our experiment, a single request with a 10 KB payload results in 7 packets being sent. This criterion can be fulfilled with very little delay besides an RTT when the loss impacts one of the first four packets, i.e., one that is immediately followed by three or more packets. The next cluster is centred around the 66 ms mark. Requests experiencing these completion times are ones where the lost packet was one of the last three packets, i.e., such that the first mechanism we described does not mark the packet as lost. Instead, the QUIC sender has to either wait for one more request to arrive such that new acknowledgements are triggered or use the time threshold, set to 9/8 of the maximum of the smoothed RTT and the last RTT sample. These two values are too close together to be discernible in our experiments. One can also note that the two last steps have relative proportions of 4/7 and 3/7, corresponding to the different cases of packet loss within a series of packets constituting a request.

Next, to set a reference point for TCPLS, we develop a simple fixed congestion controller that sets a fixed congestion window allowing more than two times the required bandwidth to be sent. This controller helps to isolate the effect of Head-of-Line blocking from the different reactions that Vegas or BBR could have. Figure 4.12 reports the evolution of the RCT CDF as more TCP connections are added to the TCPLS session. We observe that each added TCP

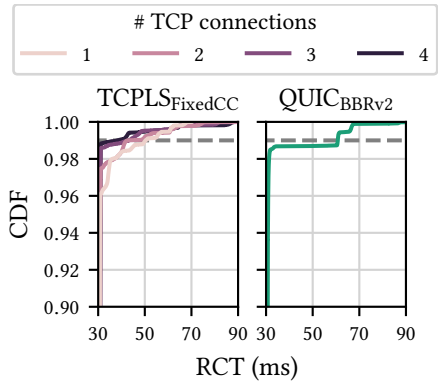


Figure 4.12: TCPLS can mitigate Head-of-Line blocking with an ideal congestion controller and several TCP connections.

connections improves the RCT and mitigates the HoL blocking effect. For the configured loss rate, three to four TCP connections seem sufficient to reduce its effect in a similar manner to QUIC.

We now consider the experiments where BBR and Vegas were used and report their results on Figure 4.13. For TCPLS$_{Vegas}$, when a single TCP connection is used, our test application becomes unstable and faces events where the congestion controller reduces the available bandwidth below our requirements. This is due to the Vegas algorithm that still considers some packet losses to be congestion-related. Starting from two TCP connections, the application can run uninterrupted. When adding a third TCP connection, a significant reduction in HoL is achieved. However, adding a fourth TCP connection only yields marginal improvements. As more TCP connections are used, each conveys requests at a slower rate and thus have fewer opportunities for their congestion controller to converge.

When looking at BBR, for both TCPLS$_{BBRv1}$ and TCPLS$_{BBRv2}$, this effect cannot be observed. Instead, adding more TCP connections can be strictly detrimental (BBRv1) or only offer marginal improvements in a limited number of cases (BBRv2). This highlights that despite providing means of mitigating the HoL blocking, interactions with the underlying congestion controllers of the TCP connections have a great impact on the final performance of the protocol. In addition, we observe that the mean latency for both BBR versions is the highest among our experiments. This could be due to the added pacing. This added latency increases with the number of TCP connections, also indicating that a reduced utilisation rate per-TCP connection has a detrimental effect on the congestion controller.

We can conclude that our second version of TCPLS protocol offers effective means of mitigating the Head-of-Line blocking problem that can arise

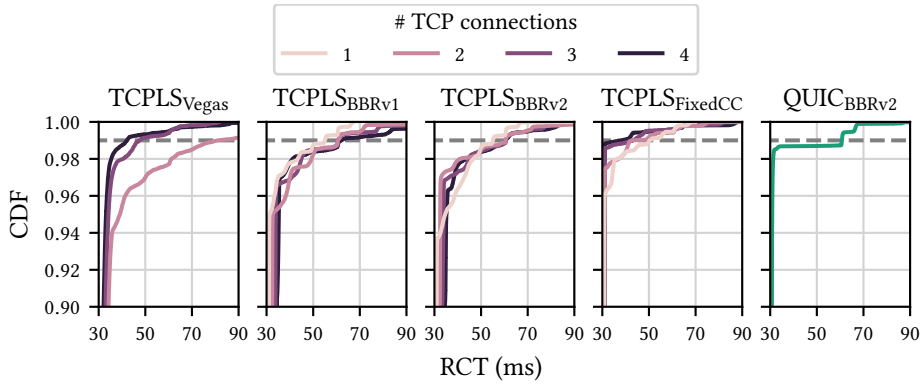


Figure 4.13: TCPLS can help mitigate Head-of-Line blocking across concurrent streams by using several TCP connections.

with TCP/TLS and MPTCP. However, more efforts and coordination with TCP congestion controllers are required as these can have a negative impact on the effective performance.

4.4.2 Latency-aware Multipath scheduling

In our second experiment, we focus on the ability of our prototype to schedule TLS records over several TCP connections. In our first experiment, the scheduling was performed in a static way as each TCPLS stream was assigned to a single connection for its entire transmission process. Given the rising importance of latency on the performance of applications, we set as a goal to develop a latency-aware Multipath scheduler.

We first developed a heuristic to determine the best TCP connection to use. In the rest of this subsection, we use the term *path* and TCP connection part of the TCPLS session interchangeably. From the point of view of the Multipath scheduler, each path represents a possible conveyor for a given set of TCPLS frames. Our heuristic is used each time a sender has application data to send, e.g., a chunk of a TCPLS stream, to determine the connection that will carry this data within a TLS record the fastest to the receiver.

The heuristic approximates the time to transfer a given amount of data based on several TCP metrics available to hosts. Algorithm 1 illustrates this heuristic. First, it defines an approximation of the time to send a given amount of bytes, represented by the function TimeToSend, by leveraging the size of the congestion window (`cwin`) and the smoothed RTT (`srtt`). The TCP congestion controller ensures that within the time frame of an RTT, the number of bytes sent equals the congestion window size. TCP provides a smoothed estimate of the RTT that we use to make the heuristic more stable.

When considering a candidate path for sending data, Algorithm 1 evalu-

Algorithm 1 Computes the time to transfer a quantity of bytes over a path

Require: n ▷ The amount of bytes to transfer
  function TIMETOSEND(n) ▷ Computes the time to send n bytes
    return $n/$`cwin` $*$ `srtt`
  function TIMETODRAIN ▷ Computes the time to drain the sending buffer of the path
    return TIMETOSEND(`queued_bytes`)
  return TIMETODRAIN + TIMETOSEND(n) + `srtt`$/2$

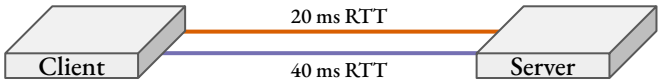


Figure 4.14: Our setup for evaluating the latency-aware TCPLS path scheduler.

ates three time quantities. First, the time to empty the bytes already queued for sending over the path. This is equal to the time required to send them. Second, the time to send the number of required bytes. Third, the time for acknowledgements to be received back by the sender. This quantity is approximated as half the smoothed RTT. Summing these three quantities give an approximation at any time of the total time required to transfer a given quantity of bytes.

When a sender receives data to send from the application and forms a set of TCPLS frames fitting a single TLS record, it selects the TCP connection that has the lowest time required to transfer this record by minimising this heuristic value. We implemented this heuristic using TCP metrics that can be obtained on Linux platforms within the `tcp_info` structure[8].

To evaluate the heuristic we designed, we conducted a series of experiments using IPMininet [JTc23]. It is an extension to the Mininet tool [LHM10] enabling researchers to emulate many types of networks with controlled characteristics. We used it to evaluate the improvements to the request completion time of a `wrk` client against a `h2o` server over a topology consisting of a client and a server connected using two separate links. All links have a capacity of 100Mbps.

4.4.2.1 A first result

As a first experiment, we configured the topology so that one link has a 20ms RTT and the other has a 40ms RTT as illustrated on Figure 4.14. We

[8] `https://elixir.bootlin.com/linux/latest/C/ident/tcp_info`

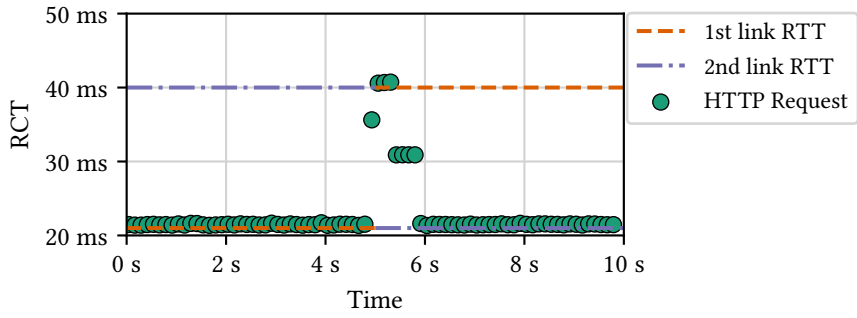


Figure 4.15: TCPLS enables `wrk` and `h2o` to rapidly reroute requests and responses to the lowest latency path.

performed HTTP requests towards the `h2o` server by configuring `wrk` to issue 8 requests per second during 10 seconds. The requested URL is the default page of the server, which is a small HTML document of 420 bytes. After 5 seconds, we reconfigure IPMininet to swap the delays of the two paths, i.e., the slow path becomes the fastest one and conversely. Such changes in delay on a given network path in the Internet is known to happen, and was studied in large cloud provider networks [Red+20].

We report the request completion time for all performed requests in Figure 4.15 and demonstrate that TCPLS is able to rapidly reroute the requests and their responses through the lowest latency path. First, when the test starts, TCPLS rapidly finds the first link to be the fastest and consistently sends request and responses on this path. At the fifth second mark, when the network is reconfigured to emulate a latency change by swapping the paths delays, we can observe that, as our heuristic leverages the smoothed RTT, it takes a couple of requests before TCPLS notices the first path to increase in delay and the second to decrease. Shortly after, one request exhibits a 30 ms completion time, indicating that one of the two endpoints has migrated to the other path, i.e, either the client or the server has sent its request or response through the second, now fastest, link. The other endpoint still utilizes the slow path, but for a short moment only as the following requests are completed again at the lowest latency available through the second link. Some outliers can be observed and are not attributed to a wrong path scheduling decisions as their added latency is too small.

4.4.2.2 Experimental design evaluation

Our first result focused on a particular network configuration. To explore more diverse scenarios, we use the experimental design approach [Fis35]. We define a range of the possible link RTT values of $[12, 70]$ms and use the WSP algorithm [SCS12] to broadly sample this parameter space into 90 points. This

# of exp.	Bandwidth	Path RTTs	Request/sec	Response	Duration
90	100 Mbps	[12, 70] ms	8 req/s	420 bytes	10 seconds

Figure 4.16: Summary of our parameters for experiments

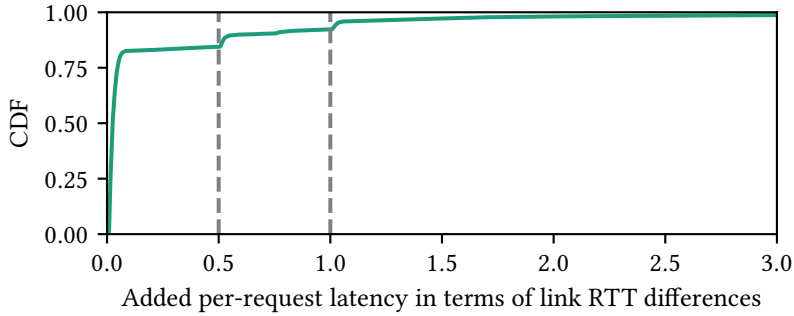


Figure 4.17: TCPLS rapidly reroute requests and responses to the lowest latency path in a diversity of networks.

mitigates the bias introduced when selecting the values. We sort the pairs of RTTs so that the first link has always the lowest one and repeat the experiment presented in Section 4.4.2.1 with each pair of link delays.

To better understand the behaviour of TCPLS across different networks, we collect all the request completion times obtained from our experimental design and represent their Cumulative Distribution Function (CDF) in Figure 4.17. The x-axis represents the added latency to each request expressed as a ratio of the link RTT differences. For instance, in our example network with RTTs of 20 and 40ms, requests that have a completion time of 20ms are represented with an added latency of 0 while requests with a completion time of 40ms are represented with an added latency of 1. Requests having an added latency of more than 1 are outliers, which could be attributed to a delay introduced by our prototype when the TCPLS session is established or to the emulation techniques used by IPMininet and their impact during the network delay change in the experiment.

We can make several observations from the CDF in Figure 4.17. First, more than 80% of the requests are using the lowest latency path. This indicates that in the presence of network delay changes, TCPLS is able to quickly and effectively reroute application data traffic to the lowest latency path. For the rest of requests, we first note that their importance in the distribution is dependent on the duration of the experiment. With a 10-second experiment, a delay change and its associated rerouting is expected to have a noticeable impact, as reported in the CDF. Moving along the x-axis, we can observe that some requests have an added latency of 0.5, which corresponds to one of endpoints using the fastest path while the other is still on the slow path. This event is

relatively short as it is not a stable point given our heuristic. Requests with an added latency of 1 constitute an even smaller set of requests, as the heuristic is able to quickly notice the change in delay.

4.5 Conclusion

In this chapter, we advanced the efforts to reconsider the way TCP and TLS are used together to provide modern transport services to application. We started by studying the first version of the TCPLS protocol and identify seven limitations within its design. Then, we designed a new version of the protocol that lift all these limitations by changing some of its fundamental principles. Our new version of the protocol retains all the modern transport services offered, i.e., multiplexing, connection migration and multipath.

We implemented this second version in a new C implementation, demonstrating how the simplification we introduced in the protocol translates into a simpler implementation. We integrated this implementation into a web server and an HTTP test tool. We evaluated it in two scenarios exploring modern transport services with an emphasis on latency. Our results demonstrate the proper functioning of our protocol and its implementation.

Adaptive Address Family Selection

5

In the past three chapters, we have focused on improvement to the transport layer with QUIC and TCPLS. These two protocols have different degrees of support for multipath capabilities, as QUIC can migrate from one network path to another and TCPLS can use several network paths simultaneously. These protocols are fit for devices that embed several Internet access technologies, such as Wi-Fi and 5G. But they are also useful when facing *path diversity* in general, e.g., when several network paths can be taken from a user device to a service provider. One of such form of path diversity arise from the coexistence of IPv4 and IPv6. In this chapter, we investigate on how latency-sensitive applications can take advantage of this diversity and focus on non-fully multipath capable protocols, such as QUIC, which has limited means to probe latency actively on several paths, and TCP, which is bound to a single network path. We design an adaptative address family selection for which we implement a prototype inside a DNS resolver. We validate its design with simulations and run experiments on real networks to evaluate its impact.

This content of this chapter was published in the article *Adaptive Address Family Selection for Latency-Sensitive Applications on Dual-stack Hosts* in the ACM SIGCOMM Computer Communications Review. We release all code from this work under an open-source licence [Pir23; wP23] and Appendix A.1 details the steps to reproduce this chapter.

5.1 IPv4-IPv6 coexistence and research questions

The Internet originally adopted IPv4 as its network protocol. With the important growth of the Internet over the years, the IPv4 address space became a constraint that the Internet Engineering Task Force (IETF) addressed by designing IPv6. It offers a much larger address space, which also brings many opportunities [Pir+22a]. The Internet being a critical infrastructure and given the key role played by IP addresses, the network community had to plan for the coexistence of IPv4 and IPv6 as a mean of transitioning. This has led to a number of challenges in several areas of the network stack. The DNS has been extended to resolve domain names into IPv6 addresses with the

AAAA [RFC3596] record type. The IPv6 addressing scheme comprises a dedicated prefix for representing IPv4 addresses [RFC4291], potentially simplifying the code of applications supporting both. The Linux socket API was also modified with *dual-stack sockets* enabling the use of IPv4 and IPv6, as well as these IPv4-mapped IPv6 addresses. Windows and macOS APIs include methods to connect a socket using domain name tackling this problem [MsCon; AplCon].

Given that the adoption of IPv6 on devices, operating systems and networks is heterogeneous [NG15; Nik+11; Col+10], very few service providers completely transitioned to IPv6 but rather became dual-stack. As a result, when an application establishes a transport connection, it needs to select one address family. This problem has seen a number of solutions over the years [RFC6724; RFC6555; RFC8305]. All of them made the hypothesis that IPv6 should be favoured to foster its transition and include a fallback mechanism in case of a broken IPv6 path. At the early stages of the IPv6 deployment, several transition solutions such as 6to4 [RFC3056] and Teredo [RFC4380] introduced tunnels to carry IPv6 packets over IPv4. These techniques added latency, could fail and become intermittently unavailable [Zan+12]. Today, as IPv6 is supported by 48 % of the 1000 top websites as reported by the ISOC Pulse platform [Soc24], a significant part of applications are facing this address family choice. Happy Eyeballs (HE) solves this problem by racing the two connection establishments while delaying the start of the IPv4 one, and selecting the one that completes first [RFC6555; RFC8305]. Its upcoming proposed version 3 [Pau+24] leverages SVCB DNS records [RFC9460; ZSC23] and honours the priority they set among resolved addresses.

In the recent years, in parallel to the deployment of IPv6, the greater importance in lowering the latency of Internet applications has started a series of changes in Internet protocols. Two recent examples are the standardisation of the QUIC protocol [RFC9000], with a faster handshake and better recovery mechanisms among its key features, and the L4S architecture [RFC9330], which provides low latency under utilisation to applications using Active Queuing Management in the network. Both help reducing the latency of packets between a host and a server.

In the light of these changes, we investigate two research questions in this chapter:

1. *Does IPv6 always provide the same latency as IPv4?*

2. *How can we improve the address family selection for latency-sensitive applications?*

The chapter addresses these questions using the following plan. First, Section 5.2 positions our work and research questions with respect to the state of

the art. Then, in Section 5.3, we show, using the RIPE Atlas dataset [RIP23a], that a diversity of latency differences between IPv4 and IPv6 exists. These differences are not common to the source and can vary across several destinations reached from a single source, with some also evolving over time. Then, in Section 5.4, we formulate the lowest-latency address family selection problem as an online learning problem balancing exploration and exploitation. Section 5.5 validates our design with simulations using the RIPE Atlas dataset. Section 5.6 presents our prototype demonstrating how DNS resolvers can aid hosts to select the lowest latency address family. Section 5.7 analyses results from real-world experiments evaluating our prototype using Chrome and popular web services.

5.2 Related works

Several researchers have studied the end-to-end performance of IPv6 that we categorise in three phases. First, transition techniques such as 6to4 [RFC3056] and Teredo [RFC4380] that were prominent in the first phase of the worldwide IPv6 deployment had a dual-sided impact, both fostering its deployment but also often degrading user experience with added latency, sometimes poorer availability and more failures [GH10]. Second, with the deployment of native IPv6 networks, these effects gradually decreased [Czy+14]. IPv6 and IPv4 had more often comparable performance when the inter-domain paths were similar, while differing paths significantly affected IPv6 performance [Dha+12]. These routing differences have been established as both the biggest contributor to poor IPv6 performance at scale and a key point of improvement through better IPv6 peering [Nik+11]. Third, as the support of IPv6 in applications increased with adoption from cloud companies and browsers, researchers analysed the impact of Happy Eyeballs (HE) on performance. IPv6 TCP connection times have improved over time, despite some still being higher than IPv4 [BS13; BS15] which sometimes is due to lacking nearby IPv6 CDN caches [BS15]. HE was also found to select IPv6 towards a large majority of popular web services despite IPv6 being slower in most cases [BS16]. Halving the fallback timer of HE to 150 ms only enables marginal improvements in avoiding higher-latency IPv6 connections [BS16; BS19]. In this work, we leverage the RIPE Atlas dataset [RIP23a] to assess the IPv4 and IPv6 performance differences today and validate our solution.

In the presence of dual-stack networks, researchers also proposed techniques beyond HE. Given several available paths, multipath transport protocols such as MPTCP [RFC8684; Rai+12] and MPQUIC [Liu+24; DB17] come as natural solutions. They can establish a connection over one path and then a subflow on the other. Then, both be can used simultaneously or selectively given application requirements, e.g., latency. An extension to MPTCP pro-

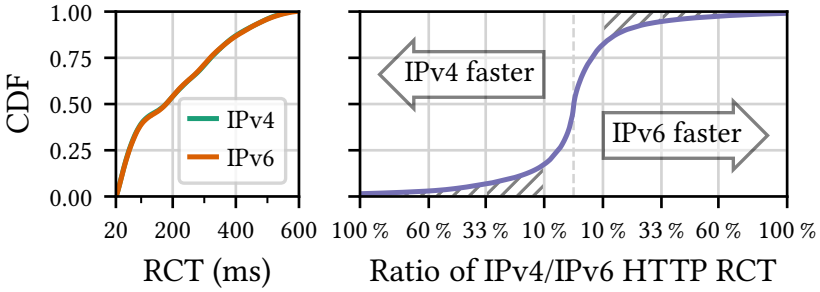


Figure 5.1: Cumulative Distribution Functions (CDF) of HTTP request completion times (RCT) are similar across address families but RCT ratios have significant disparities when paired temporally.

posed a way of racing the connection establishment over several network paths and use all succeeding attempts within the MPTCP session [Ame+18]. Researchers have also used MPTCP over IPv4 and IPv6 with a coupled congestion controller and demonstrated that bandwidth can be improved by aggregating the non-congruent parts of the two paths [Liv+15]. However, MPTCP is not widely used today [Asc+21; Wan+23] while MPQUIC is being finalised [Liu+24].

5.3 IPv4 and IPv6 end-to-end Latency

We analyse the RIPE Atlas dataset [RIP23a] to investigate the end-to-end latency of dual-stack networks. RIPE Atlas consists of more than 10000 probes which are connected, usually using Ethernet, to networks around the world and measure connectivity with several network tests. RIPE Atlas comprises also about 1000 anchors which are dedicated servers with more capacity used to co-operate with probes in their measurements. RIPE probes regularly perform HTTP tests towards anchors. This test opens a TCP connection, sends an HTTP/1.1 GET request and receives a 4 KB HTTP response in a single initial TCP congestion window. We use the request completion time of this test as a form of smoothed latency measurement from a probe to a given anchor.

When a RIPE probe has access to a dual-stack network, the HTTP tests towards an anchor are performed over IPv4 and IPv6 within a short window of time. When considering the HTTP tests between the 1st and 8th of June 2023, we obtain more than 156 millions of such paired tests, each happening within a two-minute time window. We filter out the completion times outside the 5th and 95th percentiles. The left part of Figure 5.1 reports the Cumulative Distribution Functions (CDF) of HTTP request completion times (RCT) for both IPv4 and IPv6. These are largely similar. We also compute the completion time ratios of IPv4 over IPv6 for each paired test.

IPv4 is better	IPv6 is better	None strongly better
113092 (28.4%)	129070 (32.4%)	156212 (39.2%)

Table 5.1: Pairs of probe and anchor are rather homogeneously distributed between categories.

The right part of Figure 5.1 reports the distribution of these ratios. We can first observe that neither IPv4 nor IPv6 has a lower completion time than the other, which confirms our first finding. However, there is an important number of cases in which one of the two has a lower completion time. When looking at the Figure, we can observe an equal proportion of ratios having a lower completion time by 10 % and more in favour of one address family as indicated by the hatched areas. We can conclude that there is an important diversity in terms of latency between the IPv4 and IPv6 paths from a source to a destination. When considering the use of Happy Eyeballs today, we observe that it can choose IPv6 when it has a higher latency than IPv4. However, further analysis is required to answer two questions:

1. *How are these differences distributed over pairs of probes and anchors ?*
2. *How are these differences distributed over anchors reached from a probe ?*

(1) First, we group the HTTP tests data into about 400k pairs of probes and anchors for which at least 300 HTTP test results are available. We classify each pair into three categories by performing a *t-test* determining whether its IPv4 and IPv6 RCT distributions have different means and compute a 98 % confidence interval on the means difference. When the *p-value* is lower than 0.02 and the lower bound of the confidence interval is higher than the supposedly-higher distribution standard deviation, we define the address family with a lower mean as better. When one of the two conditions is not met, we define the pair has having no significantly better address family. The second condition eliminates instances where the distributions strongly overlap. Table 5.1 reports the number of pairs in each category. We can observe that while the dominant category contains pairs with no significant difference, the other categories combined account for more than 60% of pairs. While we considered pairs with tests spanning a week, we can observe that our statistical tests establish a significant difference in a majority of cases.

Second, to better discern how stable these differences are, we deepen our analysis by performing change-point detection [WWW15] on the time series of each pair of probe and anchor to split them into segments of at least 30 tests. Figure 5.2 illustrates this process. We then perform our classification again on each segment to determine four additional groups: (A) Pairs having a significantly better address family for which at least one segment is

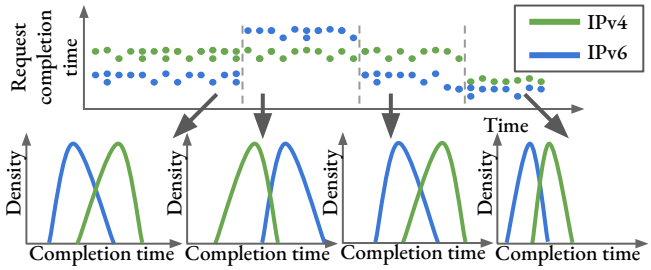


Figure 5.2: An example of probe-to-anchor RCT series split into four segments using change-point detection to highlight dynamicity inside categories of Table 5.1 and produce Table 5.2.

Group	(A)		(B)		(C)		(D)
Table 5.1 category	IPv4	IPv6	IPv4	IPv6	None	None	Consistent
∃ segment w/ category	IPv6	IPv4	None	None	IPv4	IPv6	
Number of pairs	4569		32459		27312		334034
Percentage	1.15 %		8.15 %		6.86 %		83.85 %

Table 5.2: While the classification of Table 5.1 remains consistent for 84 % of pairs, 16 % diverge in category over time.

classified in the opposite address family. (B) Pairs having a significantly better address family for which a segment is classified as having no significantly better address family. (C) Pairs having no globally significantly better address family for which one segment is classified in one of them. (D) Pairs having all segments matching the global classification.

Table 5.2 reports the number of pairs in each group. We can observe that more than 16 % of the pairs of probe and anchor have time segments that diverge in terms of best address family. Pairs can temporarily either see their best address family change from one to the other (1.15 % (A)), exhibit no significant difference among the two (8.15 % (B)) or exhibit a significant difference for one of them (6.86 % (C)). **We can conclude that differences in latency between IPv4 and IPv6 exist over a significant number of pairs of probes and anchors. These differences can be dynamic and can fluctuate over time**.

(2) Next, we investigate how the observed differences are spread among the anchors reached by a probe. We group our classification of pairs per probe and determine whether IPv4 or IPv6 performs the best in total across all destinations, i.e., having the largest number of anchors reached with the lowest latency. We then compute the ratio of anchors reached with this best address family. In the median case, only 39 % of anchors can be reached with the lowest RCT using this best address family alone. When adding anchors having no strongly better address family, in the median case, 80 % of anchors can be

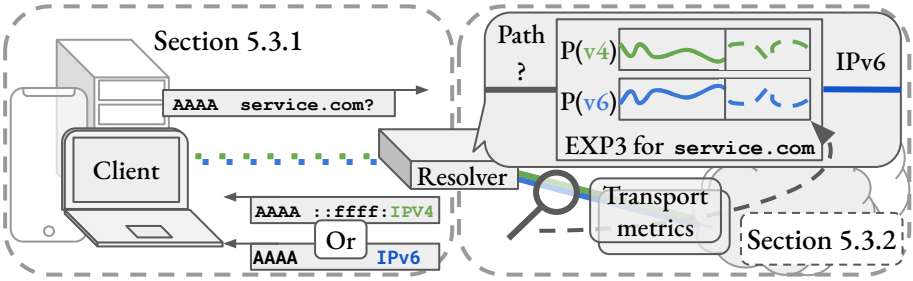


Figure 5.3: A client in a dual-stack network using the DNS resolver selecting IPv6 as the best performing address family towards a given domain name based on an online learning algorithm fed by transport metrics.

reached with the lowest RCT. **We can conclude that all dual-stack probes must use both address families to reach all anchors with the lowest completion time**.

5.4 Adaptive Address Family Selection

Based on our analysis presented in Section 5.3, we observe two facts. First, IPv6 can provide a higher latency than IPv4 in a significant number of cases. Second, providing the lowest latency address family to applications cannot be achieved by entirely disabling one address family for a given client. Moreover, we define two important properties for a latency-aware address family selection technique. First, the technique should be able to take different decisions for different destinations. Second, the technique must be able to adapt over time to changes in the address families latency.

This Section presents our design following these observations. Figure 5.3 illustrates its application on a gateway within a home or small-enterprise network. The rest of the Section further elaborates on its components.

5.4.1 Steering clients using the DNS resolver

There is an opportunity for hosts to share their knowledge of RTTs differences in their dual-stack network. We argue that the DNS resolver residing on the access gateway could influence the selected address family in its DNS responses. In a dual-stack network such as illustrated on the left-hand side of Figure 5.3, the DNS resolver can choose the address family when answering to AAAA requests by using IPv6 and IPv4-mapped addresses. The latter allows encoding IPv4 addresses as IPv6 addresses using the dedicated `::ffff:/96` prefix [RFC4291]. These IPv6 addresses are not used on the wire and represent IPv4 addresses within the IPv6 address syntax. Dual-stack applications are broadly compatible with these addresses as the Linux kernel transparently

supports them when using the BSD socket API, as well as other operating systems. An application using Happy Eyeballs will directly establish a transport connection towards these addresses as they are carried in AAAA records. Using HE v3 [Pau+24] and the `SvcPriority` attribute of SVCB DNS records is a more elegant solution but the former is not widely available yet.

5.4.2 Selecting the best address family

When our DNS resolver receives a query in a dual-stack network, it has to decide which address family yields the lowest latency towards the requested domain name to choose which address it will answer from its cache. There are several ways of tackling this problem. A first approach is for the DNS resolver to asynchronously perform pings towards the requested domain name using both address families to inform its next choice towards this destination. This has several drawbacks. First, it considers the ping time which can differ from the latency experienced by transport protocols due to network queuing and prioritisation. Second, it narrows the available information for the DNS resolver to choose the address family to a single metric, while metrics such as loss rates could be relevant to some latency-sensitive application. Third, it adds load on the network and destinations.

Another approach would be to leverage passive measurements, which alleviates these three drawbacks. Researchers have proposed efficient latency monitoring solutions that could be integrated in this work [Sol+23; SKR22]. This approach can be taken by an on-path DNS resolver that would extract from TCP headers transport-level metrics such as connection establishment times and retransmissions. For QUIC, the handshake time could still be inferred with a lesser degree of certainty by inspecting the types of packets exchanged and their timing. Off-path DNS resolvers could receive similar metrics using network telemetry from a trusted observer. Based on this knowledge, the DNS resolver can discern which address family is better for each destination or prefix. For instance, a rolling history of past performance towards a given domain name could be used to select the most favourable address family. However, when steering the clients, the DNS resolver also has to explore the address families in order to rank them.

This tradeoff between exploration of the families and exploitation of the best family is classical of multi-armed bandit problems, which are a class of reinforcement learning problems. In our context, each address family is a competing alternative action, as only one can be taken at a time, i.e., a transport connection can only be established using one address family. Each established connection can be then evaluated to estimate the gain associated with using its address family towards a given destination by extracting transport metrics (e.g, RTTs, loss rate, congestion window, . . .).

Algorithm 2 The EXP3 algorithm [Aue+02; BC+12]

Require: $\gamma \in [0, 1]$ ▷ The learning rate
Require: A ▷ The set of actions
  $w[a] = 1$ for each $a \in A$ ▷ Initialise the weights
  for each time step t **do** ▷ Each EXP3 round
    for each action a **do**
      set $p[a] = (1 - \gamma)\frac{w[a]}{\sum_{a' \in A} w[a']} + \frac{\gamma}{|A|}$ ▷ Compute the action probability
    draw a_t at random following the p probabilities
    obtain the reward $r_t \in [0, 1]$ ▷ The app. uses a_t and compute its reward
    $\hat{r}_t = r_t / p[a_t]$ ▷ Weight the reward with the action probability
    $w[a_t] = w[a_t] \exp(\gamma \hat{r}_t)$ ▷ Update the action weight

The EXP3 algorithm. The Exponential-weight algorithm for Exploration and Exploitation (EXP3) is a solution to this problem with several interesting properties [Aue+02; BC+12]. First, it makes no statistical assumptions on the process generating these gains. Second, it is resistant to adversaries. Third, it was proven to converge towards the best action at a constant rate [Aue+02]. Fourth, it is very computationally lightweight as its learning phase consists of a few float multiplications and exponentiations. EXP3 sets probabilities on available actions and updates them based on the gain obtained for each action taken.

Algorithm 2 describes EXP3 in more details. EXP3 defines a γ parameter within the range $]0, 1]$ which controls the balance between exploitation and exploration. When $\gamma = 1$, EXP3 only explores actions but never considers what it learns, acting as a random choice. Decreasing γ decreases both the learning rate and the exploration rate, i.e., the closer γ is to 0, the slower EXP3 learns but the more it will restrict its exploration to rather exploit the best action. Given a set of actions, initial probabilities and weights are set equally. For each EXP3 round, the probabilities are updated based on the weights assigned to each action. Then an action is drawn following the computed probabilities. After the action has been taken by the application using EXP3, a reward is computed and provided to EXP3 by the application. This reward is weighed according to the probability of the drawn action. A highly probable action grants a lower reward, and conversely. Finally, the weight of the drawn action is updated and a new EXP3 round can begin.

The right-hand side of Figure 5.3 illustrates how an instance of EXP3 estimates probabilities for each address family for a given destination (P(v4) and P(v6)). These probabilities are updated based on the gain observed from transport metrics for each taken action. They are used by the DNS resolver to perform a probabilistic choice on the address family it selects for a given destination.

5.5 Validation

In order to validate our solution in a controlled environment, we leverage the RIPE data used in Section 5.3 to test its performance over a large amount of data. We extract more than 400k pairs of probes and anchors with at least 300 IPv4 and IPv6 HTTP tests. Given this large amount of probes and anchors, we can use this data to evaluate our solution's ability to choose the address family with the lowest request completion time (RCT) in a large number of cases. To do so, we developed a Python simulator and an EXP3 implementation consisting of about 250 lines of code [Pir23].

We define a simulation for each pair of probe and anchor consisting of one round for each HTTP test from the probe to the anchor performed simultaneously over IPv4 and IPv6. At the simulation start, we initialise an EXP3 instance. A round consists of the following steps. First, the EXP3 instance draws the address family by forming a probabilistic choice. Second, we record whether the RCT for the chosen family is lower compared to the other family for this round based on the RIPE data. Third, we compute the gain based on the chosen family to reward and train the EXP3 instance using the following function: we give the entire reward for the chosen address family when the obtained RCT is lower than the last RCT obtained when EXP3 chose the other address family. Otherwise, we assign no reward to the action.

We set the γ EXP3 parameter to 0.1 which makes EXP3 reach 90 % probability for the best action within 60 rounds when it is strictly better than the other action. When performing a simulation, we split the first 60 rounds for training and the rest for evaluation, during which we compute the percentage of rounds where the family with the lowest RCT was chosen. We seed the random number generator (RNG) once and run each simulation 100 times to alleviate the RNG effect on the final score. We keep the median percentage of the runs as the final score for each simulation.

Figure 5.4 reports the Cumulative Distribution Functions (CDF) of three metrics computed from the results of our simulations. We compare these three metrics in three categories (Table 5.1) across three methods. These methods are: (plain lines) our EXP3 approach, (dashed lines) a fixed choice picking the address family with the largest amount of lower RCT *a posteriori* for this simulation, (dashed and dotted lines) a fixed choice picking the address family that performs the best for all simulations of this probe. The second method acts as an upper bound on the impact of selecting the best address family in a simulation while the third highlights the impact of selecting the best address family as a whole for a given probe rather than per destination.

On the top graph of Figure 5.4, we report the CDF of the median ratio of best choices, i.e., when the chosen address family leads to the lowest RCT, for

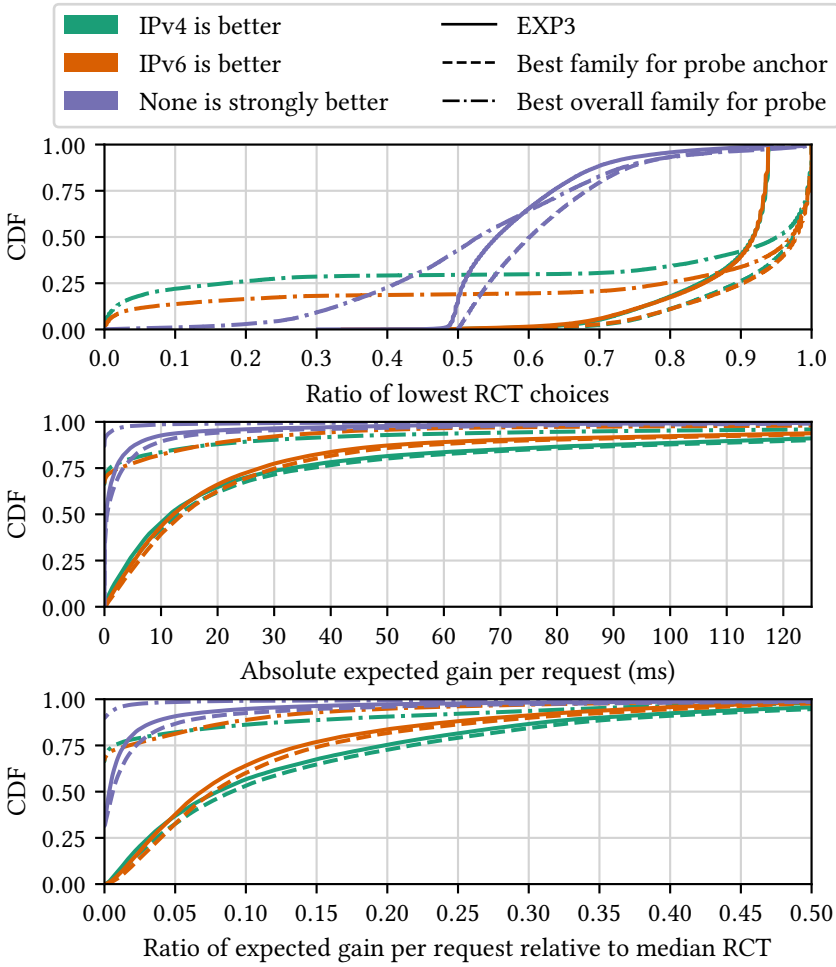


Figure 5.4: EXP3 is largely able to choose the address family with the lowest completion time in simulations using the RIPE HTTP tests data from probes to anchors.

each simulation. We can observe that when a simulation exhibits a significant difference between the completion times of IPv4 and IPv6, EXP3 is largely able to converge towards the best address family and yields a high ratio of best choices. The distance between EXP3 and our higher bound selecting the best path a posteriori for the simulation is constant and can be explained by the exploration factor of EXP3. Reducing this distance could be obtained by lowering the γ EXP3 parameter at the expense of a slower convergence. When our statistical test establishes no significant difference between IPv4 and IPv6, we can observe that EXP3 is adversely affecting performance in a very low number of simulations, i.e., simulations having a ratio less than 0.5. While EXP3 in most of these simulations gives a ratio lower than 0.55, it is also able to discern more subtle differences than established by *t-test* to select the best path. Indeed, the distance between EXP3 and selecting the best path a posteriori for the simulation is similar to the other categories.

When comparing EXP3 to our third method selecting the best path on average towards all destinations for a given probe, we observe an important regression for the latter with a different impact based on which address family is best. When IPv4 exhibits a better performance, a higher percentage of simulations that choose the probe best path have strongly adverse performance. This can also be observed for IPv6 to a lower extent. This can be explained by several factors. First, IPv4 being better in a simulation is the rarest case, which means that for a probe, IPv4 being also the best path in average towards all destinations is rarest too. Second, when IPv4 is better, the distance from its mean completion time to the mean completion time of IPv6 is higher than in the opposite case. This could exacerbate the effect of the first factor.

To better understand the request completion time gain when selecting a given path, we can observe the middle and bottom figures which depict the absolute expected gain per request and the gain relative to the median RCT. When compared to selecting the best path a posteriori (dashed lines), we can observe that EXP3 yields a gain close to this upper bound. When comparing IPv4 and IPv6, we can observe that IPv4 has a higher expected gain in a small number of cases. When IPv4 has lower RCTs as established by our statistical test, the distance from the IPv4 to the IPv6 mean RCT is higher than in the opposite case. We postulate that this could be due to networks that are still supporting IPv6 using transition solutions introducing a greater overhead than the non-congruency of IPv4 and IPv6 paths alone. When observing the right-hand side figure expressing this expected gain as a ratio of the median RCT, we can observe that the gain yielded by EXP3 can be higher than 20 % (resp. 14 %) of the median RCT for 25 % of the cases when IPv4 (resp. IPv6) is the best address family. **We can conclude that EXP3 is able to converge towards the best address family yielding the lowest request completion time and resulting in a significant gain for a large number of pairs**

of probe and anchor.

5.6 Prototype

After validating the suitability of EXP3 using the RIPE Atlas dataset, we implemented our solution in the *updns* DNS proxy [wP23] written in Rust. Our proxy receives DNS queries, queries an actual resolver and optionally alters its answers. We added 1000 lines of code to implement a DNS record cache, to improve the library used to manipulate DNS responses, and to implement EXP3 and our path selection technique.

When our prototype receives a DNS query, it uses EXP3 to probabilistically choose which address family should be used. We took several implementation decisions in this process. First, our prototype only acts when queries can be resolved from its cache in order to avoid introducing additional delays during the resolution process. When the cache is empty, our prototype simply acts as a relay to an actual resolver and caches the received responses. Second, we group in a single EXP3 instance destinations which share a common second-level domain name. This balances the need for dedicating an EXP3 instance to each destination with the benefit of grouping destinations experiencing similar address family performance differences for accelerated learning. We make the assumption that IP addresses resolved from domain names which share a common second-level domain name are likely to experience similar performance differences. Third, our prototype can also receive out-of-band feedback on the performance of address families towards destinations via a custom UDP message format. We use this feedback to compute EXP3 rewards using a similar function as used in Section 5.5: we give the full reward when the address family used gave a better performance than the last use of the other family, and none otherwise.

When receiving an AAAA query, our prototype leverages IPv4-mapped IPv6 addresses to select the address family to use towards the requested domain name. When an A query is received, our prototype respond with an empty answer to force the use of the selected address in the AAAA query. While the Happy Eyeballs algorithm as presented in Section 1.9.3 recommends using the address received in the AAAA record, we found Chrome v114 to wait for the A response to arrive before starting the handshake.

5.7 Real-world experiments

We now evaluate our prototype using real networks and towards popular Internet services. While RIPE anchors are isolated servers to which a transport connection is established directly, popular services often have distributed infrastructures in several locations and place their servers behind load-balancers

that can terminate the transport connections close to the users. We argue that measuring the end-to-end latency in this context for our work is relevant as it brings a different kind of destinations into perspective.

To find popular services, we use the Tranco list [Le +19] generated on the 5th of April 2023 [Tra23] and choose the top 40 domain names for which an IPv4 and IPv6 address can be resolved from our campus network. The resulting list has a diversity of companies and organisations providing popular services to a significant part of the Internet users.

We use the browserless Docker image [bro23] to evaluate the impact of our prototype on Chrome performance in a reproducible environment. We configure Chrome to use our prototype as its DNS resolver, which in turns leverages the DNS resolver within the network where experiments are performed to resolve queries. During an experiment, our prototype is started without any prior knowledge and Chrome is instructed to load websites one at a time from a list based on the top 40 domain names. To measure the ability of our prototype to learn and exploit the best address family towards destinations, we repeat in the list each domain name 30 times. The list is then shuffled to avoid bias. When running the experiment, we load each website from the list in three modes: IPv4 only, IPv6 only, and our prototype. This gives us a reference point for each address family to compare the quality of the choices of our prototype. These modes are also run in random order and using separate instances such that no previous run from other modes influences our prototype nor the browser. After a website is fully loaded using our prototype as a DNS resolver, we extract the handshake time of all transport connections established with the website from metrics recorder by Chrome. Then, the rewards for EXP3 are computed and attributed to the corresponding domain names using the same formula as described in Section 5.5.

We perform this experiment from several vantage points in Belgium: 1. our university campus network, 2. a DSL line, 3. a cable line, 4. a 4G Fixed Wireless Access network. All four networks are operated by different ISPs. When measuring from a vantage point, the Linux test device is connected using Ethernet. We collect all transport connections handshake times during these experiments. Then, we process them such that the first 60 tests towards a destination are not considered as they correspond to the training phase of our prototype. Then we keep the destinations for which at least 100 tests exist, such that a significant amount of connections was steered using our trained prototype. Finally, to avoid over-representing websites that use more transport connections than others, and destinations that are more frequent among these websites, such as common CDN resources, we compute the mean handshake time for each destination. We repeat this process for the two other modes.

Figure 5.5 reports the distributions of the mean handshake time per des-

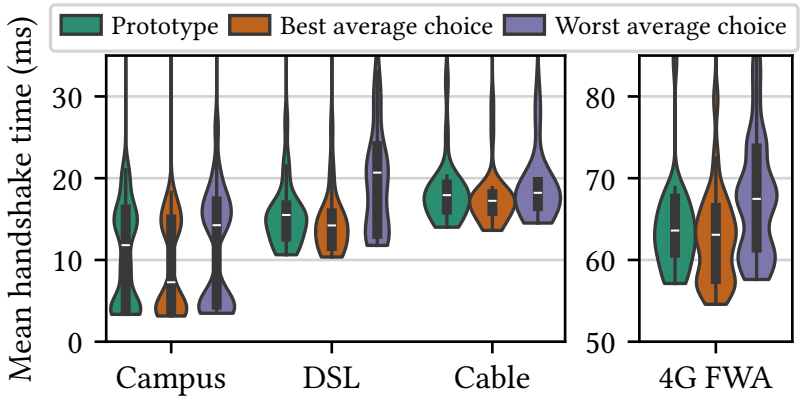


Figure 5.5: Our prototype is largely able to choose the best address family in experiments using four access networks when loading popular websites with Chrome.

	Campus	DSL	Cable	4G FWA
Prototype	11.8 ms	15.5 ms	17.9 ms	63.6 ms
Best	7.3 ms	14.2 ms	17.2 ms	63.1 ms
Worst	14.3 ms	20.7 ms	18.2 ms	67.5 ms

Table 5.3: Medians reported in Figure 5.5.

tination when loading these popular websites using our prototype on these four access networks. We compare them to the best and worst address families for each destination. We determine a posteriori which address family is better or worse using the handshake times experienced in the IPv4 and IPv6 only modes. When looking first at each distribution and Table 5.3, we can observe that our prototype improves the mean handshake time by selecting the best address family for each destination. In addition, the tail of the distributions is also reduced. There is no network in which our prototype degrades the handshake times. Its mean handshake time distribution is close to the ideal solution, and its small deviation can be explained by the EXP3 exploration. When looking at each network, the DSL line experiences the largest differences in terms of address family performance.

5.8 Discussion and further work

This work opens several directions for further work. First, we argue that the DNS protocol should be extended to allow indicating the address family that a host should use with a dedicated additional record. The Happy Eyeballs algorithm could be updated to take those hints into account when sorting the

received addresses. Second, the problem of address family selection in dual-stack networks being a subset of the path selection problem in multihomed networks, our approach could be extended to IPv6-multihomed networks. An enterprise subscribed to several network providers could use its DNS resolver to hint the source prefix to use towards a given destination. Finally, our approach could leverage other transport metrics than the connection handshake time. For instance, some latency-sensitive applications such as video conferencing also require a low loss rate and jitter while others might require a more stable bandwidth.

5.9 Conclusion

In this chapter, we revisit the choice of address family for latency-sensitive applications. Using worldwide data from RIPE Atlas, we show that Happy Eyeballs, the existing heuristic, can sometime favour IPv6 even when it has a higher latency towards a given destination compared to IPv4. We also show that this can change depending on the destination from a given source, but also over time.

We formulate the address family selection as an online learning problem. We chose the EXP3 algorithm to balance exploration of the performances of the different families towards a destination and exploitation of the best family for this destination. We simulate its effect using the RIPE Atlas data to confirm our design and then implemented it inside a DNS resolver. By placing the resolver on access gateways, clients of these networks can resolve domain names using the lowest-latency address family.

We evaluate our prototype on four access networks in Belgium by loading popular websites using the Chrome browser and using our DNS resolver. Our results indicate that the goals formulated for our problem are fulfilled and a median latency decrease can be observed over most networks.

On the benefits of delegating QUIC connections

6

In this chapter, we revisit the end-to-end principle that transport protocols abide by. It imposes hosts participating in a transport connection to remain available at all time to process packets and send acknowledgements. For battery-powered devices that often use power-costly wireless interfaces, transport connections can induce an important amount of power consumption. To alleviate this effect and reduce the energy consumed by mobile devices, we introduce more flexibility into the transport layer with our main contribution (Section 6.1) enabling a QUIC endpoint to **delegate** an established QUIC connection to another host that will maintain it and exchange the data on its behalf while it enters power-saving modes. This technique can be used by both clients and servers. We design the **QUIC delegation mechanism** and implement it in Cloudflare's `quiche` (Section 6.1.6). The evaluation of our prototype on a smartphone reveals in Section 6.2 that it improves the energy efficiency of uploads and downloads by a factor of 2 to 12 with delegated QUIC connections compared to end-to-end QUIC connections. The ability to delegate established transport connections to other hosts opens many directions for future research. We highlight a few of them in Section 6.3.

Our approach goes beyond limitations of past works. For TCP, researchers have proposed to migrate TCP connections using TCP Options [SB00; SAB01] as well as to hand-off TCP connections between servers behind a load-balancer and a switch rewriting TCP headers [Hay+21; Cho+23; Aro+00]. For QUIC, researchers have proposed to enable server-to-server connection migration using containers [Pul+22; Con+21]. Our work takes a different approach as it acts on the transport layer and does not depend on a given deployment technology, such as containers or virtual machines. The case for sharing QUIC connection state to performance enhancing-proxies as also been made. SMAQ [KSO23], enables a client to insert a middlebox in its QUIC connection, splitting it into two spliced connections with separate control loops. Our work differs as we enable the client to entirely stop processing a connection when delegated, and then retrieve and resume from its updated state.

6.1 Delegating QUIC connections

While TCP connections are identified by their 4-tuple, QUIC uses Connection Identifiers (CIDs) negotiated for each connection. Based on these identifiers, a QUIC client can use the QUIC connection migration feature to change its source address while keeping the connection active, for instance such that a smartphone moving from Wi-Fi to a cellular network can maintain its QUIC connections. We leverage the security and connection migration features of QUIC to enable delegation on established connections.

Figure 6.1 illustrates our solution when used by a client that we detail in the following subsections. We first consider an example and focus on the main steps highlighted by letters ⓐ to ⓓ. In our example, (ⓐ) a smartphone starts a large file download such as an application update. Unfortunately, factors such as congestion control and the access network capacity restrict the available bandwidth such that the smartphone will need several minutes to download the file. In order to preserve its battery, the smartphone opts to delegate the established connection to another device (e.g., a computer, laptop, access router) owned by the same user. To do so, the QUIC stack on the smartphone freezes the connection state and serializes it. Then, (ⓑ) it contacts the device that will act as a **delegate** for this connection and sends the serialized connection state. The delegate then (ⓒ) continues the connection using the QUIC connection migration feature. The delegate processes the incoming QUIC packets, stores the received application data and sends the required acknowledgements. Later, (ⓓ) the smartphone quickly retrieves the QUIC connection and all its received data from the delegate at a higher available bandwidth in its local network. A delegate differs from a proxy as the proxy maintains two transport connections, i.e., towards the client and the server, while the **delegate becomes in charge of an existing transport connection**.

We discuss these steps in more details in the following subsections. We first propose in Section 6.1.1 a method to serialize and deserialize a QUIC connection. Then in Section 6.1.2, we propose a first protocol to delegate a QUIC connection and its stream data to a delegate host. We detail how QUIC connections can be delegated securely to untrusted devices in Section 6.1.3. We expand our approach to include the delegation of QUIC connections from one server to another in Section 6.1.4. Finally, we position our design with respect to the state of the art in Section 6.1.5.

In the example above, we assumed that the same user owned the client and delegate devices. In this case, the client can trust that the delegate will not compromise the application data privacy. Given this level of trust, sharing the entire QUIC connection state, including the TLS keys, is a valid approach. However, it could also be interesting to delegate connections from a smart-

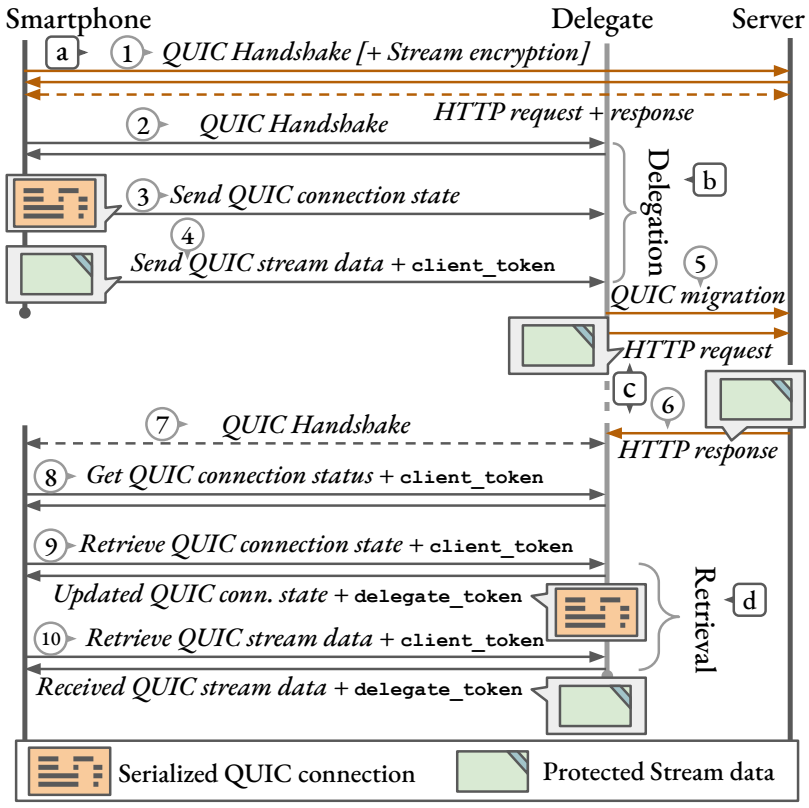


Figure 6.1: Our protocol enables to securely delegate and retrieve a QUIC connection and its stream data.

phone to an edge server, e.g., in a cellular network. In this case, the user may not fully trust its network operator as, in the past, some cellular network operators have deployed transport proxies adding tracking code and advertisements to HTTP requests and responses [Xu+15]. However, to participate in the QUIC connection, an **untrusted delegate** needs its TLS keys and thus also gains access to the application data. To remediate to this problem, we rely on a second layer of encryption between the server and the client using keys that are not shared to an untrusted delegate. This layer isolates the application data from the QUIC connection and is described in Section 6.1.3.

6.1.1 QUIC connection (de)serialization

The QUIC protocol specification [RFC9000; RFC9001] contains a significant number of requirements regarding the behaviour of QUIC clients and servers. However, it does not explicitly describe the state composing a QUIC connection. This is intended to rather emphasize on its behaviour and lets im-

plementers architect their implementations appropriately to their requirements. As a result, we argue that a mean to capture and transfer this state enables a QUIC connection to be delegated to another host. We propose to systematically encode the QUIC connection state into a binary format that can then be transferred. This state includes the TLS keys derived from the QUIC handshake and all relevant variables such that QUIC packets can be sent and received. We encode the QUIC stream data separately from the QUIC connection state to further reduce the complexity of (de)serialization. Given the exploratory nature of this work, we limit our method to transferring state between two instances of a given implementation and leave cross-implementation state transfer for further work.

6.1.2 Transferring QUIC state & application data

We design a dedicated protocol to delegate and retrieve the state of a QUIC connection and its application data, and that can authenticate each endpoint, for instance to prevent an illegitimate endpoint from retrieving a delegated QUIC connection. Our protocol uses HTTP, as its semantics fulfil these requirements. We opt for HTTP/3 to benefit from the transport and security services of QUIC.

An example sequence diagram of our protocol is illustrated on Figure 6.1. In this example, (①) a smartphone establishes a QUIC connection towards a QUIC server that is agnostic to QUIC connection delegation. When an untrusted delegate will be used, the smartphone and server negotiate our second-layer stream encryption during the handshake. In this case, the server has to support our encryption extension but remains agnostic of the delegation process. The smartphone can use the connection in a regular manner, e.g., exchanging some HTTP messages. Our technique enables QUIC connection delegation regardless of the application protocol as it acts on the QUIC connection and its QUIC streams. To delegate this connection, (②) the smartphone establishes an HTTP/3 connection to the delegate. We name the former *delegated connection* and the latter *control connection*. The smartphone authenticates the delegate using the certificate it supplied during the connection handshake. When the control connection is established, (③) the smartphone stops processing and discards incoming QUIC packets on the delegated connection, serializes and sends its state to the delegate over the control connection. At this point, the smartphone does not process the delegated connection anymore. It can provide additional QUIC stream data to be sent on the delegated connection (④). The smartphone authenticates to the delegate with a `client_token` proving its past ownership of the connection. This token is generated by successively using the TLS Exporter [RFC8446] of the delegated connection and feeding its output to the Exporter of the control

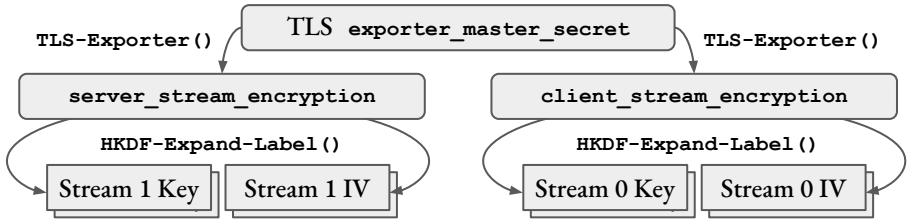


Figure 6.2: Secret and key derivations for our second-layer stream encryption.

connection using our protocol. The resulting token can be verified by each peer by reproducing these steps and comparing values. The token proves that the client has access to both the QUIC delegated connection and the control connection while preventing on-path attacker from stealing these tokens. In practice, computing this token only involves two `HKDF-Expand-Label` computations. Finally, when the delegate receives stream data, it queues it on the delegated connection. The smartphone can then stop the control connection.

After receiving the delegated connection state, (⑤) the delegate resumes the delegated connection and performs QUIC migration to transition to its source address. It sends any QUIC stream data as instructed by the smartphone, e.g. an HTTP request. It processes incoming QUIC packets, sends acknowledgements and stores the received QUIC stream data (⑥). Later, (⑦) the smartphone is reconnecting with a new control connection and (⑧) can check on the status of the delegated connection to decide when to retrieve it or access application data. When requested for state retrieval (⑨), the delegate stops processing the delegated connection, serializes its updated state and sends it in response. The delegate authenticates using a `delegate_token` constructed in a similar manner, but with a different TLS Exporter initial context value, to prove its legitimate access to the delegated QUIC connection. Finally, (⑩) the smartphone retrieves the QUIC stream data received on the delegated connection. The smartphone can resume the QUIC connection from its updated state and trigger a QUIC migration.

6.1.3 Delegating QUIC connections to untrusted devices

Delegating a QUIC connection can pose a privacy problem when the delegate device is not trusted as it will have access to the application data exchanged over the connection. When the application protocol is not encrypted, such as HTTP/3, the connection recipient can observe and modify HTTP messages. The Encrypted Content-Encoding enables securing the content of HTTP messages [RFC8188], for instance as they go through HTTP interme-

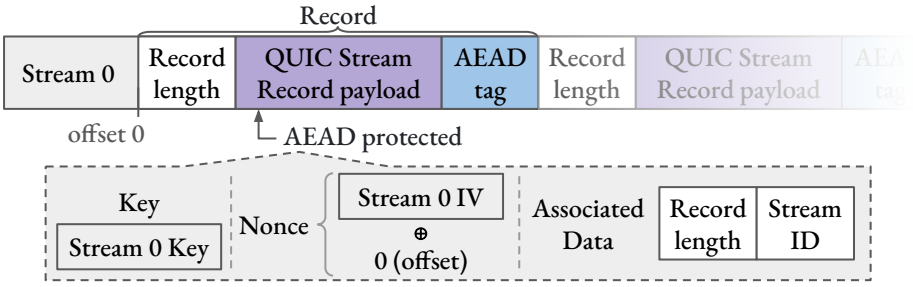


Figure 6.3: Record format of our second-layer encryption within a stream.

diaries such as push services from the Web Push protocol [RFC8291]. However, it is restricted to the HTTP protocol. We tackle this problem by proposing an optional protection that secures all the application data exchanged in QUIC streams. This additional encryption layer is negotiated during the QUIC handshake using a dedicated QUIC transport parameter named `stream_encryption`. After the handshake, as illustrated on Figure 6.2, two 48-byte secrets are generated using the TLS Exporter of the connection. The TLS Exporter context value is composed of two concatenated parts, each sent by one endpoint within its `stream_encryption` QUIC transport parameter. Each secret is used to generate keys and IVs for each sending part of a QUIC stream using `HKDF-Expand-Label`.

We introduce a thin record layer atop QUIC streams for endpoints to choose the length of each encrypted payload given an upper bound negotiated within the `stream_encryption` QUIC transport parameter. The format is illustrated on Figure 6.3. Each record starts with an integer indicating the size of the application data following it. This integer and the QUIC Stream ID are part of the Associated Data when performing AEAD operations. This guarantees that a given record cannot be swapped with another one from any other QUIC streams. The AEAD nonce is formed by combining the IV and the offset of this record within the QUIC stream. This further prevents records to be moved within QUIC streams by an attacker.

When transferring a QUIC connection that has enabled `stream_encryption`, the endpoint must redact the context parts exchanged in the QUIC transport parameters and all cryptographic materials derived as defined in this section. This ensures that, when being transferred, the new endpoint hosting the connection cannot access the application data nor tamper with it. Each endpoint should announce a context value sufficiently large to prevent brute-force attacks on these keys when transferring a QUIC connection.

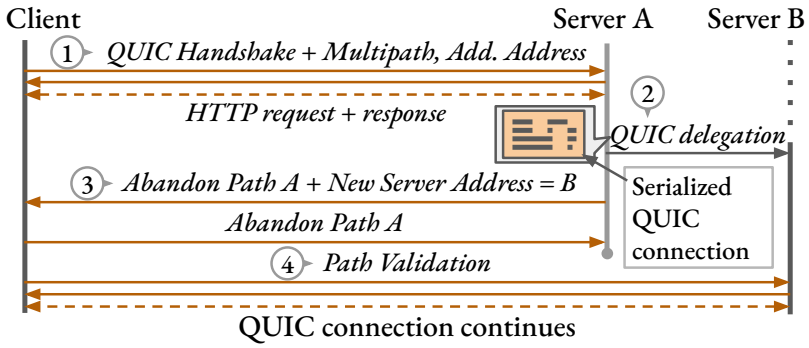


Figure 6.4: QUIC delegation can be used to move established QUIC connections between servers.

6.1.4 Server-side QUIC delegation

In addition to QUIC connection delegation on mobile clients, our technique opens opportunities on servers as well. The QUIC protocol enables a service to scale-up using load-balancers [DBH24] together with mechanisms such as Connection IDs and the `Preferred Address` transport parameter. However, it lacks a scale-down mechanism in the presence of long QUIC connections. Using delegation, a server could transfer the state of a QUIC connection to another server when scale-down is needed. However, as the QUIC protocol does not allow a server to migrate, this would require to transfer the original server IP and port as well. For TCP, past works have proposed to leverage switches to rewrite TCP headers of migrated connections [Hay+21].

Figure 6.4 illustrates how this constraint can be lifted by using the QUIC multipath extensions [DB17; Liu+24; Zhe+21] together with *Additional Addresses*, an extension enabling servers to announce their addresses [PB24b]. First, (①) a client establishes a QUIC connection towards *Server A* and enables the Multipath and Additional Addresses extensions. The client and server use the connection in a regular manner, e.g., exchanging an HTTP request and response. When Server A decides to delegate the QUIC connection, e.g., when scale-down is required to power-off Server A, (②) stops processing incoming QUIC packets, serializes the QUIC connection state and transfers it to *Server B* which will become its new host. The transfer uses the same protocol as illustrated on Figure 6.1 in steps ② to ④ and described in Section 6.1.2. Server A then (③) notifies the client that the connection cannot continue on the original path but with a new path towards the address of Server B. This is achieved with the MPQUIC `PATH_ABANDON` frame and the `ADDITIONAL_ADDRESSES` frame. The client is agnostic of the QUIC delegation and simply follows the specification of the Multipath extension. Then, the client abandons the path towards Server A and (④) triggers *Path Validation* towards

	Proxy	Sidekick [Yua+24]	SMAQ [KSO23]	QUIC Delegation
Level of delegation	App. data	Acks.	App. data	QUIC connection
Out-of-band control	None	Setup only	Setup only	Section 6.1.2
Server delegation	Limited	No	No	Section 6.1.4
Enable device sleep	No	No	No	Yes

Table 6.1: QUIC Delegation offers a greater level of delegation, enabling devices to sleep, while maintaining control over the delegated QUIC connection when needed. Our approach can be employed by QUIC servers as well.

Server B. As Server B has received the QUIC connection state, it can process the client packets and complete the validation process. This completes the server-to-server QUIC delegation.

To maintain the execution of the application from one server to another, several techniques can be used such as containers and virtual machines. We limit our work to the transport layer and do not investigate the delegation of the application.

6.1.5 Comparing QUIC Delegation to the state of the art

We compare our method to other techniques from several perspectives to establish the key differences and opportunities introduced by our solution. Table 6.1 summarises our conclusions that we detail in this subsection. We consider first the case of a transport-level proxy. In this case, the level of delegation is limited to the application data only as the proxy buffers data from the client and from the server before relaying it to their final destination. A basic proxy offers no out-of-band control and rather adds an application protocol over the client-to-proxy connection. While reverse proxies exist, they are limited in the sense that a connection must be established through the proxy, otherwise delegation cannot be performed at a later stage.

Sidekick [Yua+24] proposes to securely delegate acknowledgements to on-path devices to achieve a better link utilisation when a given hop is lossy or congested. The authors designed a discovery mechanism and a configuration protocol which are used to set up their system. While the authors only investigated acknowledgement delegation on clients, it is fair to note that their approach could probably be extended to servers as well.

SMAQ [KSO23], enables a client to insert a middlebox in its QUIC connection, splitting it into two spliced connections. It can be considered as a dynamic way of inserting a transport-level proxy in a QUIC connection. As such, it offers the same level of delegation, i.e., application data. Our work differs as we enable the client to entirely stop processing a delegated connection, and then retrieve and resume from its updated state. To secure the

application data that is conveyed by their middleboxes, they designed a construct to which ours, as detailed in Section 6.1.3, bears similarities. However, the design of SMAQ introduces a security issue that our approach does not have and that we detail in the rest of this subsection.

6.1.5.1 Security issue in SMAQ

In order to introduce a middlebox over an established QUIC connection between client and server, Kosek *et al.* proposed to share enough of the QUIC state to this middlebox such that it can act as a client to the original server and as a server to the original client. This state includes the TLS keying material used on the original QUIC connection. To foster deployment and avoid requiring further modifications of the client and server, the two spliced connections resulting of the introduction of this middlebox use the TLS keying material as defined in RFC 9001 [RFC9001]. This breaks the requirement of nonce uniqueness for TLS ciphers such as AES-GCM and ChaCha20-Poly1305. As the two connections evolve independently, they encrypt and send packets with potentially different content (e.g., acknowledgements, Stream frame boundaries, etc) but using the same key and nonce (i.e., QUIC packet number). This flaw can be used to break confidentiality and inject packets into any of the two connections by attackers that can observe packets on both of them [Böc+16]. Several measures could be taken to re-establish the uniqueness of nonces. One of them is to use the Multipath extension to QUIC [Liu+24] as each path produces a different nonce for a given packet number. The middlebox can ensure that a different logical path, i.e., a path using a different Path ID, is used on each of the two spliced connection.

6.1.6 QUIC Delegation Prototype

We implemented QUIC connection delegation atop Cloudflare's QUIC implementation named `quiche` [Clo] (commit `dfeaa589`, January 2024). We added about 2000 lines of Rust code to the library to implement (de)serialization of a QUIC connection, to implement our second-layer QUIC stream encryption, fix several erratic behaviours and create new unit tests for these features. We implemented Rust traits from the `serde` crate for (de)serialization using MessagePack [Mes21]. In addition, we developed a test client delegating QUIC connections and a server that can delegate and host delegated connections, totalling 2700 lines of Rust code.

6.2 Evaluation

To evaluate the energy savings brought to mobile devices by delegating QUIC connections, we use a simple testbed illustrated on Figure 6.5. It consists of

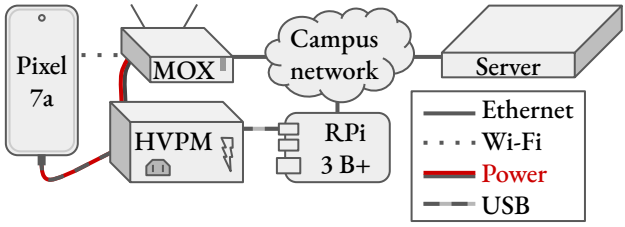


Figure 6.5: Our testbed for QUIC client delegation evaluation.

a smartphone, an access router, a two-channel power supply with monitoring capabilities and a server residing in a VM in our university cloud infrastructure. Our smartphone is a Pixel 7a running the stock OS Android 14. We removed all pre-installed applications and services, such that no transport connections are established out of our control. We set up a Linux environment using Termux [Ter24], a lightweight terminal emulator embedding ported GNU/Linux packages, to compile and test our prototype. The smartphone is remotely controlled via adb port forwarding over USB, such that commands can be executed separately from the Wi-Fi connectivity. The access router is a Turris MOX, a modular open-source access router, which we configured with Wi-Fi 6. It runs our delegate prototype, accepting the delegation of QUIC connections coming from the smartphone. The server executes a version of NGINX that we modified to support QUIC stream encryption.

The smartphone and router are powered by a Monsoon HV Power Monitor. We collect its power and voltage samples. We also collect data from the smartphone power rails through counters exposed to Perfetto, Android's tracing utility [Goo24]. When collecting all the results presented in Section 6.2.1, the smartphone is connected using the 5 GHz band. Its screen, GPS, and other communication mediums are disabled.

6.2.1 Delegating QUIC client connections

Our evaluation is based on a test scenario comprising all steps of Figure 6.1. Our smartphone establishes an HTTP/3 connection towards the server and then delegates it to a delegate running on the access router. We consider the access router trusted and do not enable our second-layer encryption. The router migrates the connection and then interacts with the server to download or upload data. A size and duration of data transfer is specified for each test. During this time, the smartphone remains idle. When the transfer finishes, we trigger the smartphone to retrieve the delegated QUIC connection and process the accumulated application data.

To explore enough scenarios during this test, we adopt the Experimental Design approach [Fis35]. We analyse two parameters that could influence

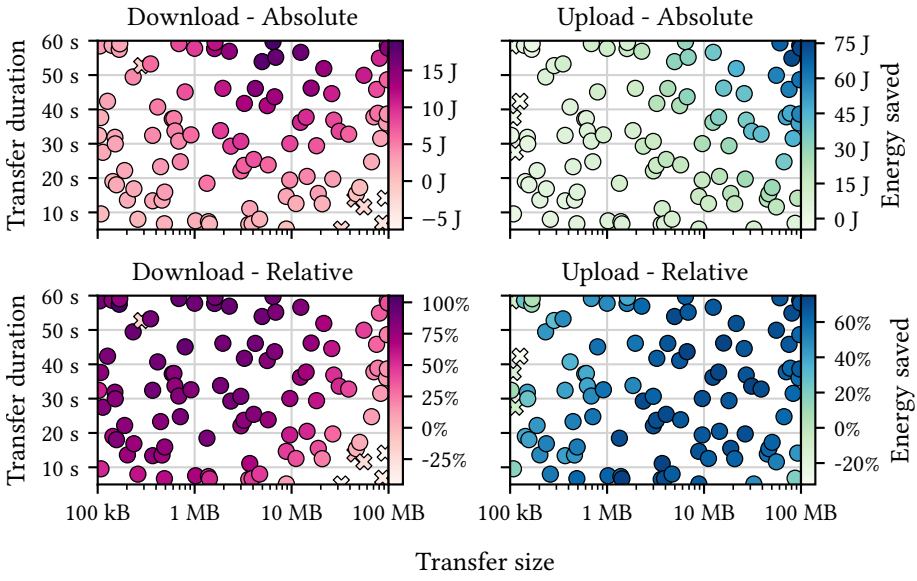


Figure 6.6: Delegating QUIC connections enables significant total energy savings for both smartphone and router.

the energy consumption: the transfer duration and the amount of data exchanged. We set ranges of [5 s, 60 s] and [100 KB, 100 MB] for the two parameters, which forms a 2D-space that we explore using the WSP space-filling algorithm [SCS12]. This limits the bias of parameter values selection and guarantees a minimum distance between values sets. We chose this distance such that 100 sets of parameter values are explored and repeat each with and without QUIC connection delegation, both for downloads and uploads. We configured NGINX such that the bandwidth and duration of downloads can be requested, and used `tc` to shape uploads to the required bandwidth value. When delegation is enabled, the smartphone executes our prototype client that uses delegation. Otherwise, it executes a `curl` version that supports QUIC using our version of `quiche`. We run each scenario three times to capture possible variance. For each test, we collect a series of metrics that we detail along with the results of this experiment in this section.

Energy savings. First, we investigate the energy saved using QUIC connection delegation. To do so, we record the power consumption of both smartphone and router. We then subtract the base idle power, which we measured at 290 mW for the smartphone and 6180 mW for the router. We compute the median amount of energy saved for each set of values for experiment parameters and comparing tests where QUIC connection delegation is enabled with ones using our end-to-end QUIC. Figure 6.6 breaks down these savings as absolute and relative values for downloads and uploads. Each point reports

the median total energy saved across both smartphone and router. In a small number of case, when this amount is negative, i.e., when delegating the QUIC connection costed more energy, the point is depicted by a filled X.

Looking at the results for downloads transfers, we can observe that in 93 % of the cases, **delegating a QUIC connection saves energy**. These savings can reach more than 15 Joules when the transfer duration is longer than 50 s. In relative terms, our delegation technique can save most of the energy spent during downloads when the transfer size is under 1 MB. For larger transfers, more resources are used by the smartphone to retrieve them from the delegate. When the connection transfers a lot of data in short amount of time, i.e., when looking at the bottom right corner of the graphs, there is less interest in delegating the QUIC connection. This is expected since the smartphone Wi-Fi interface is active for almost the same duration in both cases.

For uploads, the savings are higher in absolute terms. They increase both with transfer duration and sizes up to 75 Joules, surpassing 1 ‰ of the smartphone battery capacity. In 55 % of the explored parameter points, our QUIC delegation technique can save 60 % of the energy. These two observations are expected as uploading requires the smartphone to format, encrypt and transmit packets using its radio, which is more costly than receiving them as for downloads.

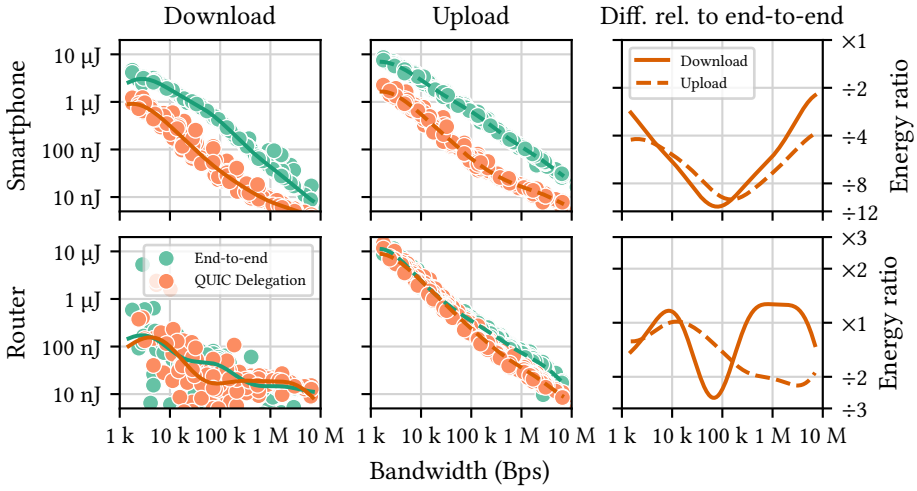


Figure 6.7: Energy efficiency is improved by a factor of 2 to 12 on smartphone for both uploads and downloads. For uploads at 200 KBps or more, QUIC delegation halved the energy used on the router.

Energy efficiency. Next, we reconsider the energy efficiency with respect to the transfer rate to evaluate the savings brought by QUIC connection delegation. For each test, we measure the transfer bandwidth based on the duration and size. Then, we compute the per-bit energy. Figure 6.7 com-

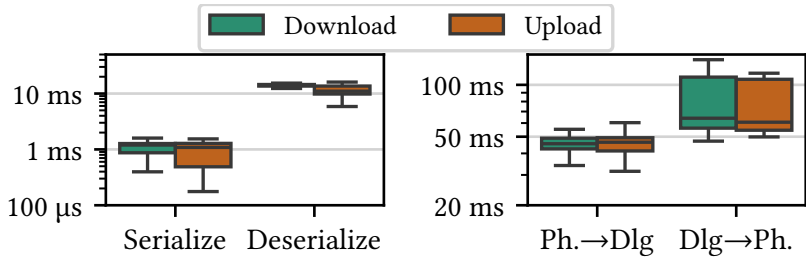


Figure 6.8: A QUIC connection can be delegated and retrieved in 124 ms in the median case.

pares the energy efficiency for downloads and uploads on both devices. For each set of points, we fit a regression using a Support Vector Machine to highlight the trend and enable comparison. The "End-to-end" points corresponds to our baseline and "QUIC Delegation" to our technique.

On the smartphone, we observe that delegating QUIC connections greatly improves the energy efficiency at all transfer rates, for both downloads and uploads. The relative difference, reported on the rightmost top graph, shows that delegating QUIC connection reduces the energy used by 2 to 12 times. Both downloads and uploads have similar gains. Regarding the router, when considering downloads, there is no clear gain or reduction of efficiency when delegating QUIC connections. This means that the additional processing that it induces is balanced with the difference in Wi-Fi usage patterns. However, for uploads at 200 KBps or more, in our experiments, delegating QUIC connections can halve the energy used on the router.

Delegation overhead. To conclude our experiments analysis, we discuss the overhead of delegating a QUIC connection. We measure the two steps required to delegate a QUIC connection: (*i*) the (de)serialization of its state and (*ii*) the exchange of the serialized state. The left graph of Figure 6.8 shows the time taken to serialize a QUIC connection. We also plot the time required to deserialise it after it has been retrieved back from the delegate at the end of the transfer. Both happen on the smartphone in a short amount of time. Our smartphone serializes the state of a QUIC connection in about 1 ms while the deserialisation takes 15 ms.

The right graph plots the time spent to transfer the serialized QUIC connection state over the local Wi-Fi network from the smartphone to the delegate (Ph.→Dlg) and back (Dlg→Ph.). Most of the QUIC connection state transfers happen in two or three RTTs.

To perform a complete QUIC delegation and retrieval, in addition to these operations and state transfers described above, the application data exchanged on the QUIC connection must also be transferred from or to the router. To measure the increase in completion time when using QUIC connection del-

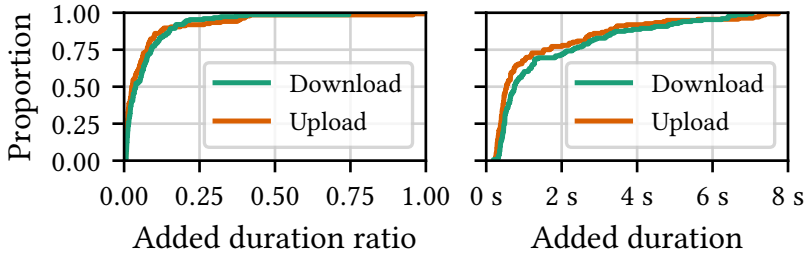


Figure 6.9: QUIC connection delegation introduces a low overhead to transfer completion time.

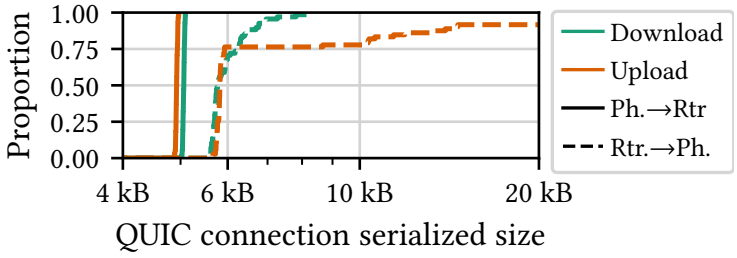


Figure 6.10: Serialized QUIC connection state remains relatively small in size.

egation, we represent the total duration of all QUIC delegation operations in Figure 6.9, represented as both a ratio of the regular transfer time and an absolute duration.

When looking at the left-hand side graph, we can observe that for 85 % of the results, the delay introduced by QUIC connection delegation was lower than 10 % of the undelegated completion time. In absolute terms, 85 % of the results, the added duration is lower than 2.5 seconds. We argue that this added delay is reasonable for applications that exchange long files, e.g., OS updates and user data backups, compared to the energy savings.

Finally, we discuss the size of serialized QUIC connections when transferred in our experiment, without considering the size of exchanged application data. Figure 6.10 reports the Cumulative Distribution Functions of this size and breaks it down by transfer direction and delegation step. The solid lines represent the state size before delegation by the smartphone while the dashed lines represent the updated state size after retrieval by the smartphone. When looking first at download transfers, we can see that the connection state and the updated state sizes remain fairly small. For uploads, the connection state is similar to downloads but in a quarter of the cases, the updated state can increase significantly, reaching hundreds of kB. By manually inspecting these cases, we identified state regarding the packets sent as the

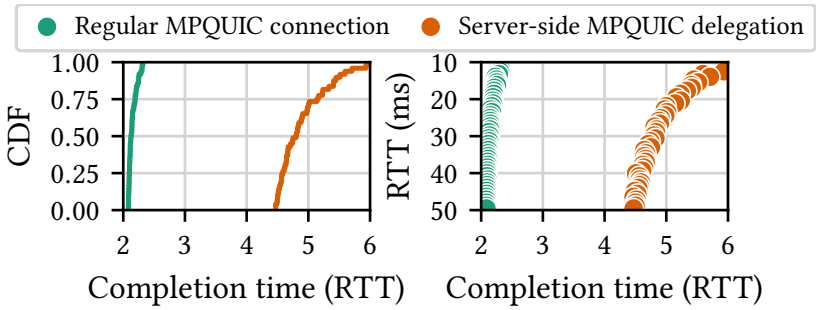


Figure 6.11: Server-side QUIC delegation happens in a few RTTs.

difference causing the size increase but we could not find a use for this state after the connection is retrieved by the smartphone. There remains work to be done on the prototype to trim down such unneeded state when present.

6.2.2 Delegation QUIC server connections

We perform a first experiment with the delegation of a QUIC server connection and focus on its impact on the client as the different steps of Figure 6.4 are undertaken. We leverage the CloudLab platform [Dup+19] to book three `c6525-25g` computers that we connect together[1]. One is used as a client and the two others as servers. We use `tc-netem` to configure a transmission delay between the client and the servers but not between the latter. This reflects the case of servers collocated within a datacenter.

Our test scenario focuses on a worst-case scenario for the client. First, the client establish a QUIC connection enabling the Multipath extension towards a first server. When established, the client sends a short HTTP request. When the server receives the request, it does not send its response but queues it and delegates the QUIC connection to the second server. After the delegation is completed, the second server sends the queued HTTP response. We measure the total request completion time from the start of the QUIC connection to the receipt of the response. We compare the impact on this time of delegating the connection or not. We use the WSP space-filling algorithm [SCS12] to select 50 RTT values between 10 and 50 ms and repeat each experiment three times.

The left part of Figure 6.11 reports the Cumulative Distribution Functions of median request completion times relative to the RTT for tests with regular QUIC connections and server-delegated QUIC connection. When using a regular connection, the request completes in two RTTs. One is spent during

[1]Hardware details are available in Section 12.1 of `https://docs.cloudlab.us/hardware.html`.

the QUIC handshake and the other for the request round-trip. The impact of delegating the connection to another server adds two and a half more RTTs in addition to the time necessary to serialize, transfer and deserialize the connection state to the other server. The half RTT correspond to step ③ and the two others to step ④ in Figure 6.4, i.e., notifying the client the path closure and validating the new path for the other server to send the HTTP response.

6.3 Discussion

In this chapter, we have shown that it is more efficient from an energy viewpoint to delegate medium and long-lived QUIC connections than to execute them entirely from a wireless client device. When the delegate is trusted, this requires no server-side additional support besides the QUIC connection migration part of the original protocol. Servers can also benefit from QUIC connection delegation to scale-down their load when needed. Our QUIC delegation scheme can be seen as an evolution of the end-to-end principle that has applied to Internet transport protocols. The idea of delegating transport connections opens several directions for future research that we briefly discuss.

Supporting delegation on other implementations and protocols. While this chapter leveraged some features of QUIC to develop its prototype, the same idea could be applied to other protocols like (Multipath) TCP combined with TLS [RFC9293; RFC8446], SCTP [RFC9260] or TCPLS [Roc+21; PRB23]. Application layer protocols could also provide delegation. In some sense, SMTP already supports delegation through its relays. For HTTP, this would complement its proxying capabilities [RFC9110; RFC9298]. For QUIC and other protocols, one of the problems to be solved is to design an efficient technique to share the state of transport (and security) connections to different implementations. This could require defining a common format akin to the efforts to standardize the `qlog` logging format for QUIC [Mar+20b; Mar+24]. Based on this common format, translation code can be written for each implementation such that a given connection can be delegated from one implementation to another. Delegation could also be adapted to MIMIQ [GWR20] to improve user's privacy.

Delegating transport connections enables to rethink explicitly about middleboxes. The introduction of delegates brings a new type of devices in the network. These delegates have some common features with various forms of middleboxes that have been deployed [She+12; HG23] without a clear place in the architecture. Since delegates are explicit, it should be easier to integrate them [RFC3234]. This could also be an opportunity to revisit the integration of content distribution networks and different types of information or content centric networking [Jac+09; Dan+13; Ahl+12] in the architecture.

Lower-layer protocols need to be better integrated with transport decisions. The lower layer protocols, such as Wi-Fi, cellular, Ethernet or Bluetooth, include various energy savings techniques [FKK12; Sah+15; LC23]. Unfortunately, these techniques are not directly exposed to the transport layer and applications. Our QUIC delegation mechanisms can enable better cross-layer interactions when the network interface and other parts of a device go to sleep [Ned+08].

Delegating transport connections could bring benefits to other applications. In this chapter, we focused on a simple application that transfers some amount of data over medium-length connections. While OS updates and user data cloud backups are expected to be prime use-cases, integrating with real applications remains to be done. This will help to further understand the implications and interactions of QUIC connection delegation on applications.

Conclusion | 7

In this thesis, we presented five contributions to the transport protocols and their uses on the Internet. In this last chapter, we conclude this thesis by discussing the future works that could stem from them and the perspectives they open.

We first developed QUIC-Tracker, the first test suite for QUIC and used it to analyse the evolution of the QUIC implementations during its standardisation phase. In addition to the results we presented, our tool has been adopted by the research community. Some other works leveraged the QLog format to analyse the diversity of QUIC implementations [Mar+20a] and their clients. However, a continuation of our active-testing approach that considers QUIC clients is yet to be undertaken. In addition, while the efforts to implement QUIC at the time of writing this thesis are lower than during the QUIC standardisation phase, we believe that new implementations will continue to arrive throughout the protocol lifetime. In this context, having a robust and complete test tool is an asset to the Internet community.

Second, several other works considered the paradigm we first explored for the extensibility of QUIC with PQUIC in other Internet protocols [Wir+23; Wir23]. Some further reconsidered how a QUIC implementation could be structured to become pluginised [De 24]. However, the integrations of QUIC plugins with applications is yet to be further investigated. Plugins can influence the behaviour of very important mechanisms, including in the extensions of QUIC, such as the MPQUIC path scheduler, or the FEC scheduler [Mic23]. When considering that they can be securely injected over a QUIC connection, they open important improvements for many applications. In the case of a Multipath QUIC VPN, the client that establish the tunnel can precisely tune the steering of data received by the server on its behalf. By injecting code acting on the transport layer to improve the performance in this scenario, this could also serve the client privacy by directly instructing the server on the choices to take instead of relying on it inspecting the application data to derive them. This could prove especially relevant in cases in which the VPN server does not have access to the user data to enhance their privacy, such as in the MASQUE proposal [RFC9298].

Third, while the usage of QUIC has rapidly increased, there remain a number of TCP applications today. For these cases, TCPLS v2 brings important improvements that may be more easily reached by application developers than

with QUIC. In this regard, an in-depth study could consider how an application benefits from a tighter integration of its network stack using TCPLS than using HTTP/2, TLS and MPTCP. Our strongly-integrated approach enables more of the application knowledge to be used by the transport layer to make impactful decisions.

Fourth, our approach to address family selection could be expanded to also benefits multipath transport protocols by starting the connection establishment on the lowest-latency path. While we considered its impact on web browsers, there are many latency-sensitive applications, such as real-time media, that could benefit from a more careful address family selection. The path diversity on the Internet is also broader and not restricted to IP versions. For instance, several home and small enterprise networks are multihomed and could leverage our technique to establish connections using the most suited provider per destination. Other metrics than latency could also be optimised, such as throughput and packet loss. Our technique could be adapted to suit these cases as well. A larger study of the impact of our technique on real networks could be conducted using experimental research clouds such as EdgeNet [Cap+18], enabling the evaluation of our prototype from many user networks across the globe.

Lastly, our chapter on QUIC delegation sets the foundation for the technique and reports promising results to enable energy savings on mobile devices. Its integration within a load-balancer and a cluster of servers remains to be done to enable a scale-down mechanism for the QUIC protocol, with potentially important energy savings. QUIC delegation also enables to reconsider the significant amount of transport connections that are established by clients to receive notifications from servers. As the time at which data is available cannot be known in advance, and the server is in control of its transfer, delegating such connections to in-network equipments could enable many devices to enter power-preserving modes.

Aside from these incremental improvements to the contributions of this thesis described in this chapter, we foresee a bigger perspective stemming from these works. During the course of this thesis, the research community became more concerned with the sustainability of the Internet infrastructure. We made a first step in this direction, our last contribution in Chapter 6. However, considerations for sustainability can be broadened to many aspects of the Internet infrastructure. As an example, while we investigated a large timescale for enabling savings with QUIC delegation, there could be other opportunities in smaller timescales to improve the energy efficiency of how transport protocols leverage wireless interfaces by forming short packet bursts instead of regularly paced packets influx. This technique could prove resistant to the *rebound effect*, as periods of silence are key to realise these savings, inciting a more frugal Internet.

A more complete research plan would be to first investigate more of these frugal patterns of use within transport protocols. Then, understand the factors that inhibit their adoptions, considering the Internet infrastructure as a network of actors defining a power dynamic regarding its evolution. In this aspect, integrating more interdisciplinarity into the research is key. Then, design techniques leveraging these patterns of use that are resistant to abuse diverting them from making the Internet infrastructure sustainable.

Bibliography

[AB18] V. Arun and H. Balakrishnan. "Copa: Practical {Delay-Based} congestion control for the internet". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 2018, pp. 329–342.

[Ahl+12] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman. "A survey of information-centric networking". In: *IEEE Communications Magazine* 50.7 (2012), pp. 26–36.

[Ame+18] M. Amend, V. Rakocevic, A. P. Matz, and E. Bogenfeld. "RobE: Robust connection establishment for multipath TCP". In: *Proceedings of the Applied Networking Research Workshop*. 2018, pp. 76–82.

[AplCon] Apple Inc. *TN3151: Choosing the right networking API*. 2023. URL: `https://developer.apple.com/documentation/technotes/tn3151-choosing-the-right-networking-api#Connect-by-name`.

[APN24] APNIC. *HTTP/3 Use — stats.labs.apnic.net*. `https://stats.labs.apnic.net/quic/XE?o=cBEw111`. [Accessed 18-09-2024]. 2024.

[Aro+00] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel. "Scalable Content-Aware Request Distribution in Cluster-Based Network Servers". In: *2000 USENIX Annual Technical Conference (USENIX ATC 00)*. 2000.

[Asc+21] F. Aschenbrenner, T. Shreedhar, O. Gasser, N. Mohan, and J. Ott. "From single lane to highways: Analyzing the adoption of multipath tcp in the internet". In: *2021 IFIP Networking Conference (IFIP Networking)*. IEEE. 2021, pp. 1–9.

[Aue+02] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire. "The nonstochastic multiarmed bandit problem". In: *SIAM journal on computing* 32.1 (2002), pp. 48–77.

[AW18] N. Amit and M. Wei. "The design and implementation of hyperupcalls". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018, pp. 97–112.

[BC+12] S. Bubeck, N. Cesa-Bianchi, et al. "Regret analysis of stochastic and nonstochastic multi-armed bandit problems". In: *Foundations and Trends® in Machine Learning* 5.1 (2012), pp. 1–122.

[BCV+07] A. Baiocchi, A. P. Castellani, F. Vacirca, et al. "YeAH-TCP: yet another highspeed TCP". In: *Proc. PFLDnet*. Vol. 7. 2007, pp. 37–42.

[Bis+05] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. "Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and sockets". In: *ACM SIGCOMM Computer Communication Review*. Vol. 35. 4. ACM. 2005, pp. 265–276.

[BL21] C. Bueger and T. Liebetrau. "Protecting hidden infrastructure: The security politics of the global submarine data cable network". In: *Contemporary Security Policy* 42.3 (2021), pp. 391–413.

[BMG99] A. Begel, S. McCanne, and S. L. Graham. "BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture". In: *ACM SIGCOMM Computer Communication Review* 29.4 (1999), pp. 123–134.

[Böc+16] H. Böck, A. Zauner, S. Devlin, J. Somorovsky, and P. Jovanovic. "Nonce-Disrespecting adversaries: practical forgery attacks on GCM in TLS". In: *10th USENIX Workshop on Offensive Technologies (WOOT 16)*. 2016.

[Bon+20b] O. Bonaventure, M. Piraux, Q. De Coninck, M. Baerts, C. Paasch, and M. Amend. "Multipath schedulers". In: *Internet Engineering Task Force, Internet-Draft draft-bonaventure-iccrg-schedulers-00* (2020).

[BP95] L. S. Brakmo and L. L. Peterson. "TCP Vegas: End to end congestion avoidance on a global Internet". In: *IEEE Journal on selected Areas in communications* 13.8 (1995), pp. 1465–1480.

[Bra17] L. Brakmo. "TCP-BPF: Programmatically tuning TCP behavior through BPF". In: *NetDev 2.2* (2017).

[Bri07] B. Briscoe. "Flow rate fairness: Dismantling a religion". In: *ACM SIGCOMM Computer Communication Review* 37.2 (2007), pp. 63–74.

[bro23] browserless.io. *browserless*. 2023. URL: `https://github.com/browserless/chrome`.

[BS13] V. Bajpai and J. Schönwälder. "Measuring TCP connection establishment times of dual-stacked web services". In: *Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013)*. IEEE. 2013, pp. 130–133.

[BS15] V. Bajpai and J. Schönwälder. "IPv4 versus IPv6-who connects faster?" In: *2015 IFIP Networking Conference (IFIP Networking)*. IEEE. 2015, pp. 1–9.

[BS16] V. Bajpai and J. Schönwälder. "Measuring the effects of happy eyeballs". In: *Proceedings of the 2016 Applied Networking Research Workshop*. 2016, pp. 38–44.

[BS19] V. Bajpai and J. Schönwälder. "A longitudinal view of dual-stacked websites—failures, latency and happy eyeballs". In: *IEEE/ACM Transactions on Networking* 27.2 (2019), pp. 577–590.

[BT16] M. Bellare and B. Tackmann. "The multi-user security of authenticated encryption: AES-GCM in TLS 1.3". In: *Advances in Cryptology–CRYPTO 2016: 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I 36*. Springer. 2016, pp. 247–276.

[Cap+18] J. Cappos, M. Hemmings, R. McGeer, A. Rafetseder, and G. Ricart. "EdgeNet: A global cloud that spreads by local action". In: *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE. 2018, pp. 359–360.

[Car+16] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson. "Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time". In: *Queue* 14.5 (2016), pp. 20–53.

[Cho+23] I. Choi, N. Wadekar, R. Joshi, J. Fried, D. R. Ports, I. Zhang, and J. Li. "Capybara: μSecond-Scale Live TCP Migration". In: *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*. 2023, pp. 30–36.

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

[Clo24] Cloudflare. *Cloudflare Radar*. `https://radar.cloudflare.com/`. 2024.

[Col+10] L. Colitti, S. H. Gunderson, E. Kline, and T. Refice. "Evaluating IPv6 adoption in the Internet". In: *International Conference on Passive and Active Network Measurement*. Springer. 2010, pp. 141–150.

[Con+21] L. Conforti, A. Virdis, C. Puliafito, and E. Mingozzi. "Extending the QUIC protocol to support live container migration at the edge". In: *2021 IEEE 22nd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. IEEE. 2021, pp. 61–70.

[Czy+14] J. Czyz, M. Allman, J. Zhang, S. Iekel-Johnson, E. Osterweil, and M. Bailey. "Measuring ipv6 adoption". In: *Proceedings of the 2014 ACM conference on SIGCOMM*. 2014, pp. 87–98.

[Dan+13] C. Dannewitz, D. Kutscher, B. Ohlman, S. Farrell, B. Ahlgren, and H. Karl. "Network of information (netinf)–an information-centric networking architecture". In: *Computer Communications* 36.7 (2013), pp. 721–735.

[Dav20] A. Davies. *RIPE Atlas on BigQuery*. 2020. URL: https://labs.ripe.net/tools/ripe-atlas-on-bigquery/ripe-atlas-on-bigquery/.

[DB17] Q. De Coninck and O. Bonaventure. "Multipath quic: Design and evaluation". In: *Proceedings of the 13th international conference on emerging networking experiments and technologies*. 2017, pp. 160–166.

[DBH24] M. Duke, N. Banks, and C. Huitema. *QUIC-LB: Generating Routable QUIC Connection IDs*. Internet-Draft draft-ietf-quic-load-balancers-19. Work in Progress. Internet Engineering Task Force, Feb. 2024. 42 pp.

[De +18] P. De Vaere, T. Bühler, M. Kühlewind, and B. Trammell. "Three Bits Suffice: Explicit Support for Passive Measurement of Internet Latency in QUIC and TCP". In: *Proceedings of the Internet Measurement Conference 2018*. ACM. 2018, pp. 22–28.

[De 20] Q. De Coninck. "Flexible multipath transport protocols". PhD thesis. UCL-Université Catholique de Louvain, 2020.

[De 22] Q. De Coninck. "The packet number space debate in multipath QUIC". In: *ACM SIGCOMM Computer Communication Review* 52.3 (2022), pp. 2–9.

[De 24] Q. De Coninck. "Core QUIC: Enabling Dynamic, Implementation-Agnostic Protocol Extensions". In: *2024 IFIP Networking Conference (IFIP Networking)*. 2024, pp. 640–646. DOI: 10.23919/IFIPNetworking62109.2024.10619827.

[Dha+12] A. Dhamdhere, M. Luckie, B. Huffaker, K. Claffy, A. Elmokashfi, and E. Aben. "Measuring the deployment of IPv6: topology, routing and performance". In: *Proceedings of the 2012 Internet Measurement Conference*. 2012, pp. 537–550.

[DL19] C. Diguet and F. Lopez. "The spatial and energy impact of data centers on the territories". In: *ENERNUM Project* (2019).

[DM06] N. Dukkipati and N. McKeown. "Why flow-completion time is the right metric for congestion control". In: *ACM SIGCOMM Computer Communication Review* 36.1 (2006), pp. 59–62.

[Don+15] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira. "{PCC}: Re-architecting congestion control for consistent high performance". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 2015, pp. 395–408.

[Don+18] M. Dong, T. Meng, D. Zarchy, E. Arslan, Y. Gilad, B. Godfrey, and M. Schapira. "{PCC} vivace:{Online-Learning} congestion control". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 2018, pp. 343–356.

[Dup+19] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. "The Design and Operation of CloudLab". In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. July 2019, pp. 1–14.

[Edg15] J. Edge. "A seccomp overview". In: *Linux Weekly News* (Sept. 2015). https://old.lwn.net/Articles/656307/.

[ERA11] N. Ekiz, A. H. Rahman, and P. D. Amer. "Misbehaviors in TCP SACK generation". In: *ACM SIGCOMM Computer Communication Review* 41.2 (2011), pp. 16–23.

[Fer+21] T. Ferreira, H. Brewton, L. D'Antoni, and A. Silva. "Prognosis: closed-box analysis of network protocol implementations". In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 2021, pp. 762–774.

[Fis35] R. A. Fisher. *The design of experiments*. Oliver & Boyd, 1935.

[FJ95] S. Floyd and V. Jacobson. "Link-sharing and resource management models for packet networks". In: *IEEE/ACM transactions on Networking* 3.4 (1995), pp. 365–386.

[FKK12] R. Friedman, A. Kogan, and Y. Krivolapov. "On power and throughput tradeoffs of wifi and bluetooth in smartphones". In: *IEEE Transactions on Mobile Computing* 12.7 (2012), pp. 1363–1376.

[Fla+13] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. "Reducing web latency: the virtue of gentle aggression". In: *ACM SIGCOMM Computer Communication Review*. Vol. 43. 4. ACM. 2013, pp. 159–170.

[Fle17] M. Fleming. "A thorough introduction to eBPF". In: *Linux Weekly News* (Dec. 2017). https://old.lwn.net/Articles/740157/.

[Geo+10] N. Geoffray, G. Thomas, J. Lawall, G. Muller, and B. Folliot. "VMKit: a substrate for managed runtime environments". In: *ACM Sigplan Notices* 45.7 (2010), pp. 51–62.

[GH10] R. Guérin and K. Hosanagar. "Fostering IPv6 migration through network quality differentials". In: *ACM SIGCOMM Computer Communication Review* 40.3 (2010), pp. 17–25.

[Gol16] D. Golumbia. *The politics of Bitcoin: Software as right-wing extremism*. U of Minnesota Press, 2016.

[Goo24] Google. *Perfetto*. `https://perfetto.dev/`. 2024.

[Gou18] A. L. Goutte. *Bug 13881 - Add (IETF) QUIC Dissector*. 2018. URL: `https://bugs.wireshark.org/bugzilla/show_bug.cgi?id=13881`.

[Gra+03] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock. "An introduction to the testing and test control notation (TTCN-3)". In: *Computer Networks* 42.3 (2003), pp. 375–403.

[Gre15] B. Gregg. *eBPF: One Small Step*. May 2015. URL: `http://www.brendangregg.com/blog/2015-05-15/ebpf-one-small-step.html`.

[Gro18a] Q. W. Group. *QUIC Implementations*. 2018. URL: `https://github.com/quicwg/base-drafts/wiki/Implementations`.

[Gro18b] Q. W. Group. *quicdev Slack*. 2018. URL: `https://quicdev.slack.com/`.

[GWR20] Y. Govil, L. Wang, and J. Rexford. "MIMIQ: Masking IPs with Migration in QUIC". In: *10th USENIX Workshop on Free and Open Communications on the Internet (FOCI 20)*. USENIX Association, Aug. 2020.

[GZ23] R. W. Gehl and D. Zulli. "The digital covenant: non-centralized platform governance on the mastodon social network". In: *Information, Communication & Society* 26.16 (2023), pp. 3275–3291.

[Haa+17] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. "Bringing the web up to speed with WebAssembly". In: *ACM SIGPLAN Notices* 52.6 (2017), pp. 185–200.

[Hät+10] S. Hätönen, A. Nyrhinen, L. Eggert, S. Strowes, P. Sarolahti, and M. Kojo. "An experimental study of home gateway characteristics". In: *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. 2010, pp. 260–266.

[Hay+21] Y. Hayakawa, M. Honda, D. Santry, and L. Eggert. "Prism: Proxies without the pain". In: *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 2021, pp. 535–549.

[Hem05] S. Hemminger. "Network emulation with NetEm". In: *Australia's National Linux Conference*. 2005, pp. 18–23.

[Hes+13] B. Hesmans, F. Duchene, C. Paasch, G. Detal, and O. Bonaventure. "Are TCP extensions middlebox-proof?" In: *Proceedings of the 2013 workshop on Hot topics in middleboxes and network function virtualization*. ACM. 2013, pp. 37–42.

[HG23] F. Hilal and O. Gasser. "Yarrpbox: Detecting middleboxes at internet-scale". In: *Proceedings of the ACM on Networking* 1.CoNEXT1 (2023), pp. 1–23.

[HL91] G. J. Holzmann and W. S. Lieberman. *Design and validation of computer protocols*. Vol. 512. Prentice hall Englewood Cliffs, 1991.

[Hon+05] O. Honda, H. Ohsaki, M. Imase, M. Ishizuka, and J. Murayama. "Understanding TCP over TCP: effects of TCP tunneling on end-to-end throughput and latency". In: *Performance, Quality of Service, and Control of Next-Generation Communication and Sensor Networks III*. Vol. 6011. International Society for Optics and Photonics. 2005, 60110H.

[Hon+11] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. "Is it still possible to extend TCP?" In: *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. ACM. 2011, pp. 181–194.

[Høs17] R. Høstaker. "The rhizome, the net and the book". In: *Empedocles: European Journal for the Philosophy of Communication* 8.2 (2017), pp. 151–165.

[Hui18] C. Huitema. *picoquic*. Source code. `https://github.com/private-octopus/picoquic`. 2018.

[IET24] IETF. *Internet standards process — ietf.org*. `https://www.ietf.org/process/process`. [Accessed 18-09-2024]. 2024.

[IO 18] IO Visor Project. *Userspace eBPF VM*. Source code. 2018.

[Jac+09] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. "Networking named content". In: *Proceedings of the 5th international conference on Emerging networking experiments and technologies*. 2009, pp. 1–12.

[Jac88] V. Jacobson. "Congestion avoidance and control". In: *ACM SIGCOMM computer communication review* 18.4 (1988), pp. 314–329.

[Jai+04] S. Jaiswal, G. Iannaccone, C. Diot, J. Kurose, and D. Towsley. "Inferring TCP connection characteristics through passive measurements". In: *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*. Vol. 3. IEEE. 2004, pp. 1582–1592.

[JRC98] R. Jain, K. Ramakrishnan, and D.-M. Chiu. "Congestion avoidance in computer networks with a connectionless network layer". In: *arXiv preprint cs/9809094* (1998).

[JTc23] M. Jadin, O. Tilmans, and other contributors. *cnp3/ipmininet: Mininet extension to make experimenting with IP networks easy — github.com*. Jan. 18, 2023.

[Ken12] B. Kenwright. "Fast Efficient Fixed-Size Memory Pool: No Loops and No Overhead". In: *The Third International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking*. 2012.

[Kha+13] R. Khalili, N. Gast, M. Popovic, and J.-Y. Le Boudec. "MPTCP is not Pareto-optimal: Performance issues and a possible solution". In: *IEEE/ACM Transactions On Networking* 21.5 (2013), pp. 1651–1665.

[KK19] Kopiersperre and Ke4roh. *Internet Hosts Count*. 2019. URL: `https://commons.wikimedia.org/wiki/File:Internet_Hosts_Count_log.svg`.

[KPH16] J. Keniston, P. S. Panchamukhi, and M. Hiramatsu. *Kernel probes (kprobes)*. Documentation provided with the Linux kernel sources. 2016.

[KSO23] M. Kosek, B. Spies, and J. Ott. "Secure Middlebox-Assisted QUIC". In: *2023 IFIP Networking Conference (IFIP Networking)*. IEEE. 2023, pp. 1–9.

[LC23] R. Liu and N. Choi. "A First Look at Wi-Fi 6 in Action: Throughput, Latency, Energy Efficiency, and Security". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 7.1 (2023), pp. 1–25.

[Le +19] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen. "Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation". In: *Proceedings of the 26th Annual Network and Distributed System Security Symposium*. NDSS 2019. Feb. 2019. DOI: 10.14722/ndss.2019.23386.

[LHM10] B. Lantz, B. Heller, and N. McKeown. "A network in a laptop: rapid prototyping for software-defined networks". In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. 2010, pp. 1–6.

[Lim+17] Y.-s. Lim, E. M. Nahum, D. Towsley, and R. J. Gibbens. "ECF: An MPTCP path scheduler to manage heterogeneous paths". In: *Proceedings of the 13th international conference on emerging networking experiments and technologies*. 2017, pp. 147–159.

[Lin+14] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[Lin24] Linux contributors. *TCP__MIN constant in Linux source code — github.com*. https://github.com/torvalds/linux/blob/65249feb6b3df9e17bab5911ee56fa7b0971e231/include/net/tcp.h#{}L147. [Accessed 19-09-2024]. 2024.

[Liu+24] Y. Liu, Y. Ma, Q. D. Coninck, O. Bonaventure, C. Huitema, and M. Kühlewind. *Multipath Extension for QUIC*. Internet-Draft draft-ietf-quic-multipath-10. Work in Progress. Internet Engineering Task Force, July 2024. 36 pp.

[Liv+15] I. Livadariu, S. Ferlin, Ö. Alay, T. Dreibholz, A. Dhamdhere, and A. Elmokashfi. "Leveraging the IPv4/IPv6 identity duality by using multi-path transport". In: *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE. 2015, pp. 312–317.

[LP24] A. Luykx and K. G. Paterson. "Limits on authenticated encryption use in TLS". In: *Cryptology ePrint Archive* (2024).

[LX17] D. Lukaszewski and G. Xie. "Multipath transport for virtual private networks". In: *10th USENIX Workshop on Cyber Security Experimentation and Test (CSET 17)*. USENIX. 2017.

[MAF04] A. Medina, M. Allman, and S. Floyd. "Measuring interactions between transport protocols and middleboxes". In: *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*. ACM. 2004, pp. 336–341.

[Mar+20a] R. Marx, J. Herbots, W. Lamotte, and P. Quax. "Same standards, different decisions: A study of QUIC and HTTP/3 implementation diversity". In: *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. 2020, pp. 14–20.

[Mar+24] R. Marx, L. Niccolini, M. Seemann, and L. Pardue. *QUIC event definitions for qlog*. Internet-Draft draft-ietf-quic-qlog-quic-events-07. Work in Progress. Internet Engineering Task Force, Mar. 2024. 60 pp.

[Mas+01] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang. "TCP Westwood: Bandwidth estimation for enhanced transport over wireless links". In: *Proceedings of the 7th annual international conference on Mobile computing and networking*. 2001, pp. 287–297.

[McQ+20] S. McQuistin, V. Band, D. Jacob, and C. Perkins. "Parsing protocol standards to parse standard protocols". In: *Proceedings of the Applied Networking Research Workshop*. 2020, pp. 25–31.

[ME+04] M. Musuvathi, D. R. Engler, et al. "Model Checking Large Network Protocol Implementations." In: *USENIX NSDI'04*. 2004, pp. 12–12.

[Mes21] MessagePack contributors. *MessagePack*. `https://github.com/msgpack/msgpack`. 2021.

[MHR03] M. Mathis, J. Heffner, and R. Reddy. "Web100: extended TCP instrumentation for research, education and diagnosis". In: *ACM SIGCOMM Computer Communication Review* 33.3 (2003), pp. 69–79.

[Mic+22] F. Michel, M. Trevisan, D. Giordano, and O. Bonaventure. "A first look at starlink performance". In: *Proceedings of the 22nd ACM Internet Measurement Conference*. 2022, pp. 130–136.

[Mic23] F. Michel. "Flexible QUIC loss recovery mechanisms for latency-sensitive applications". PhD thesis. University of Kassel, Germany, 2023.

[Mis+19] A. Mishra, X. Sun, A. Jain, S. Pande, R. Joshi, and B. Leong. "The great internet TCP congestion control census". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 3.3 (2019), pp. 1–24.

[Mön24] S. d. Mönnink. "From Network to Platform to Protocol: Mastodon's Ethos and the Sociotechnical Imaginary of the Fediverse". MA thesis. 2024.

[MR04] N. Modadugu and E. Rescorla. "The Design and Implementation of Datagram TLS". In: *Network and Distributed System Security Symposium (NDSS'04)*. 2004.

[MsCon] Microsoft. *TcpClient.Connect Method*. 2023. URL: https://learn.microsoft.com/en-us/dotnet/api/system.net.sockets.tcpclient.connect?view=net-7.0#system-net-sockets-tcpclient-connect(system-string-system-int32).

[MsRTO] Microsoft Support. *Minimum RTO value in Windows — support.microsoft.com*. https://support.microsoft.com/en-au/topic/you-cannot-customize-some-tcp-configurations-by-using-the-netsh-command-in-windows-server-2008-r2-c1feebea-82a8-cb05-83c7-46ffb5fd9cec. [Accessed 19-09-2024]. 2024.

[MTL22] A. Mishra, W. H. Tiu, and B. Leong. "Are we heading towards a BBR-dominant Internet?" In: *Proceedings of the 22nd ACM Internet Measurement Conference*. 2022, pp. 538–550.

[MZ19] K. L. McMillan and L. D. Zuck. "Formal specification and testing of QUIC". In: *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 227–240.

[Ned+08] S. Nedevschi, L. Popa, G. Iannaccone, S. Ratnasamy, and D. Wetherall. "Reducing Network Energy Consumption via Sleeping and Rate-Adaptation." In: *NSDI*. Vol. 8. 2008, pp. 323–336.

[NG15] M. Nikkhah and R. Guérin. "Migrating the internet to IPv6: An exploration of the when and why". In: *IEEE/ACM Transactions on Networking* 24.4 (2015), pp. 2291–2304.

[Nik+11] M. Nikkhah, R. Guérin, Y. Lee, and R. Woundy. "Assessing IPv6 through web access a measurement study and its findings". In: *Proceedings of the Seventh conference on emerging Networking EXperiments and Technologies*. 2011, pp. 1–12.

[OHc24] K. Oku, C. Huitema, and other contributors. *h2o/picotls: TLS 1.3 implementation in C — github.com*. 2024. URL: https://github.com/h2o/picotls.

[Pau+24] T. Pauly, D. Schinazi, N. Jaju, and K. Ishibashi. *Happy Eyeballs Version 3: Better Connectivity Using Concurrency*. Internet-Draft draft-pauly-v6ops-happy-eyeballs-v3-01. Work in Progress. Internet Engineering Task Force, Mar. 2024. 19 pp.

[Pax97] V. Paxson. "Automated packet trace analysis of TCP implementations". In: *ACM SIGCOMM Computer Communication Review* 27.4 (1997), pp. 167–179.

[PB24b] M. Piraux and O. Bonaventure. *Additional addresses for QUIC*. Internet-Draft draft-piraux-quic-additional-addresses-02. Work in Progress. Internet Engineering Task Force, Mar. 2024. 8 pp.

[PF01] J. Pahdye and S. Floyd. "On inferring TCP behavior". In: *ACM SIGCOMM Computer Communication Review* 31.4 (2001), pp. 287–298.

[Pir18a] M. Piraux. *pigotls – A very minimal Go binding for picotls*. 2018. URL: https://github.com/mpiraux/pigotls.

[Pir18b] M. Piraux. *QUIC-Tracker — github.com*. https://github.com/QUIC-Tracker. 2018.

[Pir18c] M. Piraux. *QUIC-Tracker web application*. 2018. URL: https://quic-tracker.info.ucl.ac.be.

[Pir22a] M. Piraux. *h2o web server with TCPLS v2*. 2022. URL: https://github.com/mpiraux/h2o.

[Pir22b] M. Piraux. *wrk tool with TCPLS v2*. 2022. URL: https://github.com/mpiraux/wrk-tcpls.

[Pir23] M. Piraux. *Artefacts for the article Adaptive Address Family Selection for Latency-Sensitive Applications on Dual-stack Hosts*. 2023. URL: https://github.com/mpiraux/ccr23-artefacts/.

[Pir24a] M. Piraux. *QUIC-Tracker dataset*. Version V1. 2024. DOI: 10.14428/DVN/OTOIZE.

[Pir24b] M. Piraux. *rapido, an implementation of TCPLS v2*. 2024. URL: https://github.com/mpiraux/rapido.

[PKS18] T. Pauly, E. Kinnear, and D. Schinazi. *An Unreliable Datagram Extension to QUIC*. Internet-Draft draft-pauly-quic-datagram-01. Dec. 2018.

[PRB23] M. Piraux, F. Rochet, and O. Bonaventure. *TCPLS: Modern Transport Services with TCP and TLS*. Internet-Draft draft-piraux-tcpls-04. Work in Progress. Internet Engineering Task Force, Oct. 2023. 25 pp.

[PRT21] M. Pearson, M. Rithmire, and K. S. Tsai. "Party-state capitalism in China". In: *Current History* 120.827 (2021), pp. 207–213.

[Pul+22] C. Puliafito, L. Conforti, A. Virdis, and E. Mingozzi. "Server-side QUIC connection migration to support microservice deployment at the edge". In: *Pervasive and mobile computing* 83 (2022), p. 101580.

[RAB20] F. Rochet, E. Assogba, and O. Bonaventure. "TCPLS: Closely Integrating TCP and TLS". In: *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. 2020, pp. 45–52.

[Rai+12] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. "How hard can it be? designing and implementing a deployable multipath TCP". In: *9th USENIX symposium on networked systems design and implementation (NSDI 12)*. 2012, pp. 399–412.

[Ram12] A. Ramaiah. *TCP option space extension*. Internet-Draft draft-ananth-tcpm-tcpoptext-00. Work in Progress. Internet Engineering Task Force, Mar. 2012. 20 pp.

[RAP22] F. Rochet, E. Assogba, and M. Piraux. *GitHub - pluginized-protocols/-picotcpls: TCPLS implementation using a TLS 1.3 implementation in C – Published in ACM CoNEXT'21 – Best Community Award — github.com*. `https://github.com/pluginized-protocols/picotcpls`. 2022.

[Red+20] W. Reda, K. Bogdanov, A. Milolidakis, H. Ghasemirahni, M. Chiesa, G. Q. Maguire Jr, and D. Kostić. "Path persistence in the cloud: A study of the effects of inter-region traffic engineering in a large cloud provider's network". In: *ACM SIGCOMM Computer Communication Review* 50.2 (2020), pp. 11–23.

[RFC2018] S. Floyd, J. Mahdavi, M. Mathis, and D. A. Romanow. *TCP Selective Acknowledgment Options*. RFC 2018. Oct. 1996. DOI: 10.17487/RFC2018.

[RFC2119] S. O. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. RFC 2119. Mar. 1997. DOI: 10.17487/RFC2119.

[RFC2473] D. S. E. Deering and A. Conta. *Generic Packet Tunneling in IPv6 Specification*. RFC 2473. Dec. 1998. DOI: 10.17487/RFC2473.

[RFC2525] M. Allman, B. Fenner, J. Griner, I. Heavens, K. Lahey, D. V. Paxson, J. Semke, and B. Volz. *Known TCP Implementation Problems*. RFC 2525. Mar. 1999. DOI: 10.17487/RFC2525.

[RFC2595] C. Newman. *Using TLS with IMAP, POP3 and ACAP*. RFC 2595. June 1999. DOI: 10.17487/RFC2595.

[RFC3056] B. E. Carpenter and K. Moore. *Connection of IPv6 Domains via IPv4 Clouds*. RFC 3056. Feb. 2001. DOI: 10.17487/RFC3056.

[RFC3234] S. W. Brim and B. E. Carpenter. *Middleboxes: Taxonomy and Issues*. RFC 3234. Feb. 2002. DOI: 10.17487/RFC3234.

[RFC3596] V. Ksinant, C. Huitema, D. S. Thomson, and M. Souissi. *DNS Extensions to Support IP Version 6*. RFC 3596. Oct. 2003. DOI: 10.17487/RFC3596.

[RFC4291] D. S. E. Deering and B. Hinden. *IP Version 6 Addressing Architecture*. RFC 4291. Feb. 2006. DOI: 10.17487/RFC4291.

[RFC4380] C. Huitema. *Teredo: Tunneling IPv6 over UDP through Network Address Translations (NATs)*. RFC 4380. Feb. 2006. DOI: 10.17487/RFC4380.

[RFC4513] R. Harrison. *Lightweight Directory Access Protocol (LDAP): Authentication Methods and Security Mechanisms*. RFC 4513. June 2006. DOI: 10.17487/RFC4513.

[RFC5116] D. McGrew. *An Interface and Algorithms for Authenticated Encryption*. RFC 5116. Jan. 2008. DOI: 10.17487/RFC5116.

[RFC5482] F. Gont and L. Eggert. *TCP User Timeout Option*. RFC 5482. Mar. 2009. DOI: 10.17487/RFC5482.

[RFC5925] D. J. D. Touch, R. Bonica, and A. J. Mankin. *The TCP Authentication Option*. RFC 5925. June 2010. DOI: 10.17487/RFC5925.

[RFC6298] M. Sargent, J. Chu, D. V. Paxson, and M. Allman. *Computing TCP's Retransmission Timer*. RFC 6298. June 2011. DOI: 10.17487/RFC6298.

[RFC6356] C. Raiciu, M. J. Handley, and D. Wischik. *Coupled Congestion Control for Multipath Transport Protocols*. RFC 6356. Oct. 2011. DOI: 10.17487/RFC6356.

[RFC6555] D. Wing and A. Yourtchenko. *Happy Eyeballs: Success with Dual-Stack Hosts*. RFC 6555. Apr. 2012. DOI: 10.17487/RFC6555.

[RFC6724] D. Thaler, R. P. Draves, A. Matsumoto, and T. Chown. *Default Address Selection for Internet Protocol Version 6 (IPv6)*. RFC 6724. Sept. 2012. DOI: 10.17487/RFC6724.

[RFC7258] S. Farrell and H. Tschofenig. *Pervasive Monitoring Is an Attack*. RFC 7258. May 2014. DOI: 10.17487/RFC7258.

[RFC7323] D. Borman, R. T. Braden, V. Jacobson, and R. Scheffenegger. *TCP Extensions for High Performance*. RFC 7323. Sept. 2014. DOI: 10.17487/RFC7323.

[RFC7540] M. Belshe, R. Peon, and M. Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. May 2015. DOI: 10.17487/RFC7540.

[RFC768] J. Postel. *User Datagram Protocol*. RFC 768. Aug. 1980. DOI: 10.17487/RFC0768.

[RFC7858] Z. Hu, L. Zhu, J. Heidemann, A. Mankin, D. Wessels, and P. E. Hoffman. *Specification for DNS over Transport Layer Security (TLS)*. RFC 7858. May 2016. DOI: 10.17487/RFC7858.

[RFC8188] M. Thomson. *Encrypted Content-Encoding for HTTP*. RFC 8188. June 2017. DOI: 10.17487/RFC8188.

[RFC8291] M. Thomson. *Message Encryption for Web Push*. RFC 8291. Nov. 2017. DOI: 10.17487/RFC8291.

[RFC8305] D. Schinazi and T. Pauly. *Happy Eyeballs Version 2: Better Connectivity Using Concurrency*. RFC 8305. Dec. 2017. DOI: 10.17487/RFC8305.

[RFC8446] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446.

[RFC8484] P. E. Hoffman and P. McManus. *DNS Queries over HTTPS (DoH)*. RFC 8484. Oct. 2018. DOI: 10.17487/RFC8484.

[RFC8684] A. Ford, C. Raiciu, M. J. Handley, O. Bonaventure, and C. Paasch. *TCP Extensions for Multipath Operation with Multiple Addresses*. RFC 8684. Mar. 2020. DOI: 10.17487/RFC8684.

[RFC8689] J. Fenton. *SMTP Require TLS Option*. RFC 8689. Nov. 2019. DOI: 10.17487/RFC8689.

[RFC8999] M. Thomson. *Version-Independent Properties of QUIC*. RFC 8999. May 2021. DOI: 10.17487/RFC8999.

[RFC9000] J. Iyengar and M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. May 2021. DOI: 10.17487/RFC9000.

[RFC9001] M. Thomson and S. Turner. *Using TLS to Secure QUIC*. RFC 9001. May 2021. DOI: 10.17487/RFC9001.

[RFC9002] J. Iyengar and I. Swett. *QUIC Loss Detection and Congestion Control*. RFC 9002. May 2021. DOI: 10.17487/RFC9002.

[RFC9110] R. T. Fielding, M. Nottingham, and J. Reschke. *HTTP Semantics*. RFC 9110. June 2022. DOI: 10.17487/RFC9110.

[RFC9113] M. Thomson and C. Benfield. *HTTP/2*. RFC 9113. June 2022. DOI: 10.17487/RFC9113.

[RFC9114] M. Bishop. *HTTP/3*. RFC 9114. June 2022. DOI: 10.17487/RFC9114.

[RFC9221] T. Pauly, E. Kinnear, and D. Schinazi. *An Unreliable Datagram Extension to QUIC*. RFC 9221. Mar. 2022. DOI: 10.17487/RFC9221.

[RFC9250] C. Huitema, S. Dickinson, and A. Mankin. *DNS over Dedicated QUIC Connections*. RFC 9250. May 2022. DOI: 10.17487/RFC9250.

[RFC9260] R. R. Stewart, M. Tüxen, and karen Nielsen. *Stream Control Transmission Protocol*. RFC 9260. June 2022. DOI: 10.17487/RFC9260.

[RFC9293] W. Eddy. *Transmission Control Protocol (TCP)*. RFC 9293. Aug. 2022. DOI: 10.17487/RFC9293.

[RFC9298] D. Schinazi. *Proxying UDP in HTTP*. RFC 9298. Aug. 2022. DOI: 10.17487/RFC9298.

[RFC9312] M. Kühlewind and B. Trammell. *Manageability of the QUIC Transport Protocol*. RFC 9312. Sept. 2022. DOI: 10.17487/RFC9312.

[RFC9330] B. Briscoe, K. D. Schepper, M. Bagnulo, and G. White. *Low Latency, Low Loss, and Scalable Throughput (L4S) Internet Service: Architecture*. RFC 9330. Jan. 2023. DOI: 10.17487/RFC9330.

[RFC9406] P. Balasubramanian, Y. Huang, and M. Olson. *HyStart++: Modified Slow Start for TCP*. RFC 9406. May 2023. DOI: 10.17487/RFC9406.

[RFC9438] L. Xu, S. Ha, I. Rhee, V. Goel, and L. Eggert. *CUBIC for Fast and Long-Distance Networks*. RFC 9438. Aug. 2023. DOI: 10.17487/RFC9438.

[RFC9460] B. M. Schwartz, M. Bishop, and E. Nygren. *Service Binding and Parameter Specification via the DNS (SVCB and HTTPS Resource Records)*. RFC 9460. Nov. 2023. DOI: 10.17487/RFC9460.

[RIP23a] RIPE NCC. *RIPE Atlas*. 2023. URL: https://atlas.ripe.net.

[RIP23b] RIPE NCC. *RIPE Atlas Service Terms and Conditions*. 2023. URL: https://www.ripe.net/about-us/legal/ripe-atlas-service-terms-and-conditions.

[RKS06] S. Rewaskar, J. Kaur, and F. D. Smith. "A passive state-machine approach for accurate analysis of TCP out-of-sequence segments". In: *ACM SIGCOMM Computer Communication Review* 36.3 (2006), pp. 51–64.

[RR20] G. S. Reen and C. Rossow. "Dpifuzz: A differential fuzzing framework to detect dpi elusion strategies for quic". In: *Proceedings of the 36th Annual Computer Security Applications Conference*. 2020, pp. 332–344.

[Rüt+18] J. Rüth, I. Poese, C. Dietzel, and O. Hohlfeld. "A First Look at QUIC in the Wild". In: *Passive and Active Measurement*. Springer International Publishing, 2018, pp. 255–268. ISBN: 978-3-319-76481-8.

[SAB01] A. C. Snoeren, D. G. Andersen, and H. Balakrishnan. "Fine-Grained Failover Using Connection Migration." In: *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*. 2001.

[Sah+15] S. K. Saha, P. Deshpande, P. P. Inamdar, R. K. Sheshadri, and D. Koutsonikolas. "Power-throughput tradeoffs of 802.11 n/ac in smartphones". In: *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE. 2015, pp. 100–108.

[SB00] A. C. Snoeren and H. Balakrishnan. "An end-to-end approach to host mobility". In: *Proceedings of the 6th annual international conference on Mobile computing and networking*. 2000, pp. 155–166.

[SCS12] J. Santiago, M. Claeys-Bruno, and M. Sergent. "Construction of space-filling designs using WSP algorithm for high dimensional spaces". In: *Chemometrics and Intelligent Laboratory Systems* 113 (2012), pp. 26–31.

[She+12] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. "Making middleboxes someone else's problem: Network processing as a cloud service". In: *ACM SIGCOMM Computer Communication Review* 42.4 (2012), pp. 13–24.

[SKR22] S. Sengupta, H. Kim, and J. Rexford. "Continuous in-network round-trip time monitoring". In: *Proceedings of the ACM SIGCOMM 2022 Conference*. 2022, pp. 473–485.

[Soc24] I. Society. *Pulse*. 2024. URL: `https://pulse.internetsociety.org/technologies`.

[Sol+23] D. Soldani, P. Nahi, H. Bour, S. Jafarizadeh, M. Soliman, L. Di Giovanna, F. Monaco, G. Ognibene, and F. Risso. "eBPF: A New Approach to Cloud-Native Observability, Networking and Security for Current (5G) and Future Mobile Networks (6G and Beyond)". In: *IEEE Access* (2023).

[Ste+17] E. Stephan, M. Cayla, A. Braud, and F. Fieau. "QUIC Interdomain Troubleshooting". Internet draft, draft-stephan-quic-interdomain-troubleshooting-00.txt, work in progress. July 2017.

[SV96] M. Shreedhar and G. Varghese. "Efficient fair queuing using deficit round-robin". In: *IEEE/ACM Transactions on networking* 4.3 (1996), pp. 375–385.

[TE18] M. Thomson and L. Eggert. *7th Implementation Draft*. 2018. URL: https://github.com/quicwg/base-drafts/wiki/7th-Implementation-Draft.

[TE24] D. J. D. Touch and W. Eddy. *TCP Extended Data Offset Option*. Internet-Draft draft-ietf-tcpm-tcp-edo-14. Work in Progress. Internet Engineering Task Force, Sept. 2024. 23 pp.

[Tel24] TeleGeography. *Submarine Cable Map*. [Accessed 04-12-2024]. 2024. URL: https://www.submarinecablemap.com/.

[Ter24] Termux contributors. *Termux*. https://termux.dev/en/. 2024.

[TK18] B. Trammell and M. Kuehlewind. *The QUIC Latency Spin Bit*. Internet-Draft draft-ietf-quic-spin-exp-01. Oct. 2018.

[Tra23] Tranco list. *Information on the Tranco list with ID 6JL4X*. Apr. 2023. URL: https://tranco-list.eu/list/6JL4X.

[Wah+94] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. "Efficient software-based fault isolation". In: *ACM SIGOPS Operating Systems Review* 27.5 (1994), pp. 203–216.

[Wan+15] K. Wang, Y. Lin, S. M. Blackburn, M. Norrish, and A. L. Hosking. "Draining the swamp: Micro virtual machines as solid foundation for language development". In: *LIPIcs-Leibniz International Proceedings in Informatics*. Vol. 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2015.

[Wan+23] C. Wang, H. Wang, F. Qian, K. Zheng, C. Wang, F. Mao, X. Guo, and C. Xu. "Experience: A Three-Year Retrospective of Large-scale Multipath Transport Deployment for Mobile Applications". In: *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*. 2023, pp. 1–15.

[War+19] R. Ware, M. K. Mukerjee, S. Seshan, and J. Sherry. "Modeling BBR's interactions with loss-based congestion control". In: *Proceedings of the internet measurement conference*. 2019, pp. 137–143.

[Wes18] M. Westerlund. "Proposal for adding ECN support to QUIC." 2018.

[Wir+23] T. Wirtgen, T. Rousseaux, Q. De Coninck, N. Rybowski, R. Bush, L. Vanbever, A. Legay, and O. Bonaventure. "xBGP: Faster Innovation in Routing Protocols". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 2023, pp. 575–592.

[Wir23] T. Wirtgen. “Improving the agility of BGP routing”. PhD thesis. UCL-Université Catholique de Louvain, 2023.

[Wis+11] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. “Design, implementation and evaluation of congestion control for multipath TCP”. In: *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. 2011.

[wP23] wyhaya and M. Piraux. *updns*. 2023. URL: `https://github.com/mpiraux/updns`.

[WWW15] G. D. Wambui, G. A. Waititu, and A. Wanjoya. “The power of the pruned exact linear time (PELT) test in multiple change-point detection”. In: *American Journal of Theoretical and Applied Statistics* 4.6 (2015), p. 581.

[Xu+15] X. Xu, Y. Jiang, T. Flach, E. Katz-Bassett, D. Choffnes, and R. Govindan. “Investigating transparent web proxies in cellular networks”. In: *Passive and Active Measurement: 16th International Conference, PAM 2015, New York, NY, USA, March 19-20, 2015, Proceedings 16*. Springer. 2015, pp. 262–276.

[Yan+14] P. Yang, J. Shao, W. Luo, L. Xu, J. Deogun, and Y. Lu. “TCP congestion avoidance algorithm identification”. In: *IEEE/ACM Transactions On Networking* 22.4 (2014), pp. 1311–1324.

[Yee+09] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. “Native client: A sandbox for portable, untrusted x86 native code”. In: *30th IEEE Symposium on Security and Privacy*. IEEE. 2009, pp. 79–93.

[Yua+24] G. Yuan, M. Sotoudeh, D. K. Zhang, M. Welzl, D. Mazières, and K. Winstein. “Sidekick:In-Network Assistance for Secure End-to-End Transport Protocols”. In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 2024, pp. 1813–1830.

[Zan+12] S. Zander, L. L. Andrew, G. Armitage, G. Huston, and G. Michaelson. “Investigating the IPv6 teredo tunnelling capability and performance of internet clients”. In: *ACM SIGCOMM Computer Communication Review* 42.5 (2012), pp. 13–20.

[Zhe+21] Z. Zheng, Y. Ma, Y. Liu, F. Yang, Z. Li, Y. Zhang, J. Zhang, W. Shi, W. Chen, D. Li, Q. An, H. Hong, H. H. Liu, and M. Zhang. “XLINK: QoE-driven multi-path QUIC transport in large-scale video services”. In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. SIGCOMM ’21. Virtual Event, USA: Association for

Computing Machinery, 2021, pp. 418–432. ISBN: 9781450383837. DOI: 10.1145/3452296.3472893.

[ZK23] A. Zapletal and F. Kuipers. "Slowdown as a metric for congestion control fairness". In: *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. 2023, pp. 205–212.

[ZSC23] J. Zirngibl, P. Sattler, and G. Carle. "A first look at SVCB and HTTPS DNS resource records in the wild". In: *2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE. 2023, pp. 470–474.

Appendix A

A.1 Reproducibility of Chapter 5

This Appendix details the required steps and material to reproduce the results presented in this work. All the code developed and data referenced in this section will be made publicly available under an open-source licence [Pir23; wP23]. An archive containing those two repositories was submitted along with this thesis. The RIPE data that we downloaded and processed is subject to their legal terms [RIP23b].

A.1.1 IPv4 and IPv6 end-to-end latency

The RIPE data presented was first extracted using the BigQuery interface [Dav20]. The query used is in the file `/e2e_latency/query.sql`. The resulting data should be saved in a CSV file. For this purpose, the helper Python script `/e2e_latency/download_bq_results.py` connects to BigQuery and downloads a `data.csv` file. It must be adapted with credentials and BigQuery Job ID. The expected file size is around 13.4 GB. The `/e2e_latency/figure_1.py` script produces Figure 5.1. We ran it on machine with 128 GB of RAM as it loads the entire CSV file.

To produce Table 5.1, 5.2 and the numbers cited in (2) of Section 5.3, the required data is produced by the Python simulator described in Section 5.5. The rest of the output of the simulator is discussed in the other parts of the Appendix. The simulator use a condensed view of the data, so the `data.csv` must be processed into the `paths.cbor` file using the `/simulations/prepare\_data.py` Python script. This script was also run on machine with 128 GB of RAM. Then, running the `/simulations/ripe.py` Python script produces the required output. The rest of the Python scripts in the `/e2e_latency` folder (i.e `table_1.py`, `table_2.py`) can be run to obtain the corresponding Table.

A.1.2 Validation

Figure 5.4 requires running first the simulator as described in Appendix A.1.1. Then `/validation/figure_4.py` can be run to produce the Figure.

A.1.3 Real-world experiments

The `/experiments` folder contains the scripts necessary to run experiments and parse the results into the JSON format used to produce Figure 5.5. The real-word experiments require a Linux machine with Docker installed. We used the `browserless/chrome` Docker image hosted on `docker.io` with ID `e6460f3f8f60`. Our prototype is available in a separate repository [wP23]. Its Docker image should be built before running experiments. Then, the

`/experiments/run_browserless.py` Python script starts all the experiments described Section 5.7 over a given network. This script needs to be adapted to specify the interface to use through indicating its IPv4 address in the `IP` global variable. The results are stored individually in a directory. This directory can be parsed by the `/experiments/parse_netlogs.py` Python script into a JSON file format summarising all results. The `/experiments/figure_5.py` Python script illustrates how Figure 5.5 was produced and should be modified when the number of networks tested changes.