Flexible Multipath Transport Protocols

Quentin De Coninck

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor in Applied Sciences

March 2020

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

Thesis Committee:

Pr. Olivier **Bonaventure** (Advisor)	UCLouvain/ICTEAM, Belgium
Pr. Ramin **Sadre**	UCLouvain/ICTEAM, Belgium
Pr. Benoit **Donnet**	Université de Liège, Belgium
Pr. Charles **Pecheur** (Chair)	UCLouvain/ICTEAM, Belgium
Pr. Costin **Raiciu**	Universitatea Politehnica din Bucuresti, Romania

Flexible Multipath Transport Protocols

by Quentin De Coninck

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

This work was partially supported by the F.R.S-FNRS.

Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it.

— EDSGER W. DIJKSTRA

Preamble

In fifty years, the Internet has drastically altered our everyday lives. In the early seventies, the ARPANET was the first computer network linking huge mainframes between a few American universities. Nowadays, the Internet connects billions of devices — sometimes smaller than a bank cart — and causes a constant revolution of our society. From an economical viewpoint, computer networks introduced the age of digital industry. Together, e-commerce platforms and social networks impact our culture, our economy and our human interactions.

The last ten years have seen a new trend in the Internet: device mobility. In the early 2000s, most of the computers were heavy desktops connected to the Internet by cable. These devices were intended to remain static on a desk. Now, mobile devices such as smartphones, tablets and connected cars become the norm. These devices include several wireless network interfaces such as Wi-Fi and LTE to keep Internet connectivity when the user moves. This ability to seamlessly switch from one network to another is critical to support such mobile use cases.

Unfortunately, the dominant TCP transport protocol [RFC793] does not enable hosts to perform such network handover. It was designed in the seventies to work in the ARPANET network. At that time, early computers were connected with a single link and communication took place over a single bidirectional path. Because TCP uses IP addresses as identifiers, a connection is bound to them and cannot shift to another network resulting in different IP addresses. For example, a smartphone losing Wi-Fi connectivity must tear down all the associated TCP connections — hence notifying a network issue to the user — and then recreates new ones over the always available LTE network. In addition to generate user's frustration, it adds complexity to applications wanting to support such network mobility.

To address this TCP design issue, Multipath TCP [Rai+12; RFC6824] allows connections to seamlessly take advantage of multiple paths. It was designed as a TCP extension to work in the same networks as plain TCP. A Multipath TCP connection can start using the Wi-Fi network and then continue on the LTE one without the intervention of the application. This solution attracted interest from industry with several large-scale deployments [Bon13; KT; BS16; CSP17; Tes19].

However, the usage of multiple paths brings its own specific concerns.

The first one is the selection of the path to use for the next packet to be sent. Some applications initiate large bulk transfers and want to aggregate the available network bandwidths. Other applications instead generate light network load but require the lowest latency. In addition, smartphone users typically care about the usage of the cellular network — more expensive and power hungry than the Wi-Fi one — while others just do not pay attention to that. How can a multipath transport protocol handle all these requests by design?

This thesis contributes to the exploration of the performance of multipath transport protocols under various scenarios and proposes solutions to let them adapt to the use case they serve. In particular, the main contributions of this thesis are the following.

- **Performing Multipath TCP measurements on smartphone with real users.** Previous works analyzed the performance of Multipath TCP to assess whether it reaches its bandwidth aggregation goal, especially in controlled environments. However, little is known about how Multipath TCP behaves on smartphones in the field. To fill this gap, we perform two measurement campaigns to observe how the two main Multipath TCP implementations — the Linux kernel one and the iOS one — operate in the smartphone environment.

- **Adapting Multipath TCP to the smartphone use case.** While our previous measurements show that Multipath TCP works in real networks, they also point out some suboptimal strategies related to the mismatch between the bandwidth aggregation tuning of Multipath TCP and the requirement of mobile device applications. Especially, interactive applications such as Apple's Siri need to keep the response latency as low as possible, even in mobile situations, while meeting the user's expectations of using the Wi-Fi network when available. For this, we consider such request/response network traffic, adapt Multipath TCP to this use case and evaluate it with Android 6 smartphones.

- **Designing Multipath extensions for the QUIC protocol.** Both in-network middleboxes and TCP design issues affect the flexibility of Multipath TCP. These hinder the deployment of innovations at the transport layer. To get rid of them, we consider the QUIC protocol whose specification is being finalized within the IETF. Thanks to its near-fully encrypted and authenticated UDP-based packets and its clean design, it makes it easy to develop new extensions without experiencing network interference. Based on our experience with Multipath TCP, we design Multipath extensions for the QUIC protocol and evaluate them in a broad range of emulated and real networks.

- **Revisiting protocol extensibility with plugins.** Protocol extensions are negotiated during the connection handshake. While this determines if an extension can be used over a given exchange, it does not solve its deployment. TCP extensions — including Multipath TCP— typically suffer from the chicken-and-egg issue where both the client and the server wait for its peer to support it first. QUIC offers us an opportunity to reconsider the whole design of transport protocols. Instead of proposing black-box transport protocol implementations, we propose a gray-box approach where an implementation exposes an API offering protocol operations. This API can be modified or even extended by protocol plugins exchanged over an encrypted QUIC connection. We design such a pluginizable protocol and demonstrate that we can provide a plugin offering multipath capabilities adapted to a specific use case.

The remaining of this thesis is organized as follows. We first introduce in Chapter 1 the background related to Multipath TCP. Chapter 2 then describes our two Multipath TCP measurement campaigns on smartphones. Based on the lessons learned, we present in Chapter 3 an adapted version of Multipath TCP tuned for interactive applications on smartphones. This work particularly motivates the need for higher flexibility at the transport layer. For this, we consider the QUIC protocol described in Chapter 4. First, Chapter 5 proposes and evaluates a Multipath design for an early version of QUIC. We then reconsider in Chapter 6 how these Multipath extensions can be adapted to the current version of QUIC and compare its benefits to the early design. Next, Chapter 7 presents Pluginized QUIC, a truly extensible transport protocol allowing bytecode injection inside its implementations, and demonstrates its generic approach by implementing the Multipath extensions for the QUIC protocol using only plugins. Finally, we conclude this thesis and discuss future research directions in Chapter 8.

Bibliographic notes

Conference Publications

1. Q. De Coninck, M. Baerts, B. Hesmans, and O. Bonaventure. "A First Analysis of Multipath TCP on Smartphones". In: *Passive and Active Measurement – PAM'16*. Springer International Publishing, April 2016, pp. 57–69. ISBN: 978-3-319-30504-2. DOI: 10.1007/978-3-319-30505-9_5.

2. Q. De Coninck and O. Bonaventure. "Multipath QUIC: Design and Evaluation". In: *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies - CoNEXT '17*. Incheon, Republic of Korea: ACM Press, December 2017, pp. 160–166. ISBN: 978-1-4503-5422-6. DOI: 10.1145/3143361.3143370.

3. Q. De Coninck and O. Bonaventure. "Tuning Multipath TCP for Interactive Applications on Smartphones". In: *Proceedings of the 17th IFIP Networking Conference and Workshops – IFIP Networking'18*. May 2018, pp. 1–9. DOI: 10.23919/IFIPNetworking.2018.8696520.

4. V.-H. Tran, H. Tazaki, Q. De Coninck, and O. Bonaventure. "Voice-Activated Applications and Multipath TCP: A Good Match?" In: *Proceedings of the 2nd Workshop on Mobile Network Measurement – MNM'18*. June 2018, pp. 1–6. DOI: 10.23919/TMA.2018.8506489.

5. 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 – EPIQ'18*. ACM, 2018, pp. 8–14. DOI: 10.1145/3284850.3284852.

6. F. Michel, Q. De Coninck, and O. Bonaventure. "QUIC-FEC: Bringing the benefits of Forward Erasure Correction to QUIC". In: *Proceedings of the 18th IFIP Networking Conference (IFIP Networking) and Workshops – IFIP Networking'19*. IEEE, May 2019. DOI: 10.23919/IFIPNetworking.2019.8816838.

7. Q. De Coninck and O. Bonaventure. "MultipathTester: Comparing MPTCP and MPQUIC in Mobile Environments". In: *Proceedings of the 3rd Workshop on Mobile Network Measurement – MNM'19*. June 2019, pp. 221–226. DOI: 10.23919/TMA.2019.8784653.

8. 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 ACM Special Interest Group on Data Communication -*

SIGCOMM '19. Beijing, China: ACM Press, 2019, pp. 59–74. ISBN: 978-1-4503-5956-6. DOI: 10.1145/3341302.3342078.

9. T. Wirtgen, C. Dénos, Q. De Coninck, M. Jadin, and O. Bonaventure. "The Case for Pluginized Routing Protocols". In: *Proceedings of the 27th IEEE International Conference on Network Protocols – ICNP'19*. Chicago, Illinois, USA: IEEE, Oct. 2019, p. 12.

Posters

1. Q. De Coninck, M. Baerts, B. Hesmans, and O. Bonaventure. "Poster: Evaluating Android Applications with Multipath TCP". In: *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking - MobiCom '15*. Paris, France: ACM Press, 2015, pp. 230–232. ISBN: 978-1-4503-3619-2. DOI: 10.1145/2789168.2795165.

2. Q. De Coninck, M. Baerts, B. Hesmans, and O. Bonaventure. "A First Analysis of Multipath TCP on Smartphones (Poster Version)". In: *Traffic Monitoring and Analysis Workshop – TMA'2016*. April 2016.

3. Q. De Coninck and O. Bonaventure. "Observing Network Handovers with Multipath TCP". In: *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*. ACM, 2018, pp. 54–56.

Journal Publications

1. Q. De Coninck, M. Baerts, B. Hesmans, and O. Bonaventure. "Observing Real Smartphone Applications over Multipath TCP". In: *IEEE Communications Magazine* 54.3 (Mar. 2016), pp. 88–93. ISSN: 0163-6804. DOI: 10.1109/MCOM.2016.7432153.

2. V.-H. Tran, Q. De Coninck, B. Hesmans, R. Sadre, and O. Bonaventure. "Observing real Multipath TCP traffic". In: *Computer Communications* 94 (2016), pp. 114–122. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2016.01.014.

Technical Reports

1. Q. De Coninck and O. Bonaventure. *Every Millisecond Counts: Tuning Multipath TCP for Interactive Applications on Smartphones*. Tech. rep. 2017. URL: http://hdl.handle.net/2078.1/185717.

2. F. Michel, Q. De Coninck, and O. Bonaventure. "Adding forward erasure correction to quic". In: *arXiv preprint arXiv:1809.04822* (2018).

3. Q. De Coninck and O. Bonaventure. *The Case for Protocol Plugins*. Tech. rep. 2019. URL: `http://hdl.handle.net/2078.1/216493`.

IETF Contributions

1. Q. De Coninck and O. Bonaventure. *Multipath Extensions for QUIC (MP-QUIC)*. Internet-Draft draft-deconinck-quic-multipath-04. Internet Engineering Task Force, Mar. 2020.

Miscellaneous Contributions

1. M. Piraux, Q. De Coninck, and F. Michel. *QUIC Tutorial: Bringing innovation back to the transport layer*. Invited talk at the Conférence d'informatique en Parallélisme, Architecture et Système – COMPAS'19. Anglet, France, June 2019.

Reproducibility Considerations

Although implementations and measurement scripts are not strictly speaking scientific contributions, they contribute to them by showing their feasibility and their reproducibility. All the software required to run our experiments are made open-source and can be freely fetched through the listed URLs.

Android measurements (§2.2)	`http://smartphone.multipath-tcp.org`
MultipathTester (§2.3 & §5.3)	`https://github.com/multipathtester`
MultiMob (§3)	`https://multipath-tcp.org/multimob`
Multipath gQUIC (§5)	`https://multipath-quic.org`
PQUIC (§6 & §7)	`https://pquic.org`

Acknowledgments

Although being unoriginal, the first person I want to thank is my advisor, Olivier Bonaventure. By proposing me a stimulating master's thesis, he contaminated me with the incredible applied research virus and pushed me to realize a Ph.D. thesis. Despite being busy with many side projects, he always found the time to provide me insightful suggestions and feedback while letting me direct my own work. If I achieved all the contributions presented in this thesis, this is mostly thanks to his support.

I also want to thank all the members of my thesis committee for the time they spent reading my thesis: Costin Raiciu, Benoit Donnet, Ramin Sadre and Charles Pecheur. We had fruitful discussions during my private defense and I am confident their comments will be very valuable for my future works.

While I am the author of this thesis, its content is the result of collaborations with other worthwhile people. I first acknowledge my master's thesis co-author, Matthieu Baerts, without whom I would not have pursued a Ph.D. I am also grateful to all my other co-authors: Benjamin Hesmans, Viet-Hoang Tran, Ramin Sadre, François Michel, Maxime Piraux, Florentin Rochet, Thomas Given-Wilson, Axel Legay, Olivier Pereira, Thomas Wirtgen, Cyril Dénos and Mathieu Jadin. A special thanks goes to Hajime Tazaki who invited me as a lecturer to the IIJ Innovation Institute seminar in Tokyo. Besides, I also want to thank Étienne Rivière for its invitation to present a tutorial at the COMPAS conference and Laurent Vanbever for having accepted my visit to ETH Zurich.

In addition, I am grateful to the whole IP Networking Lab team and the INGI department. Among them, I want to mention Fabien Duchêne for being an entertaining co-desk colleague, David Lebrun for his short but intense stay in our office, Olivier Tilmans for our fruitful discussions and Mathieu Jadin for his joyful mood. Special mention to the wonderful QUIC team: François Michel and his CAFEC time, Maxime Piraux for his precise specification knowledge and Florentin Rochet for his secure point of view. I also thank all the colleagues with which I shared the office (Raphaël Bauduin, Benjamin Hesmans, Viet-Hoang Tran, Thomas Wirtgen,...). I should also include the whole lunchtime team and all the belote players, but they are too many to list them (please forgive me). The administrative team must not be forgotten. In particular, I want to thank our secretary Vanessa Maons (which is worth more than any other secretary), our accountants (Sophie Renard,

Steve Arokium, Margaux Hubin,...) and our sysadmin team (Pierre Reinbold, Nicolas Detienne, Anthony Gégo, Ludovic Taffin,...).

Finally, I want to thank my family and in particular my mother Cathy, my father Vincent and my grand-mother Chantal for their support during this long journey. Last but not least, my last thoughts go to Lise. Since that winter day, she brings me her merry light in my life and her unfailing support, even in the thoughest times, allowed me to complete this thesis.

Quentin De Coninck
9th March, 2020

Contents

Reliable Multipath Transfer

1

In order to grasp the remaining of this thesis, we introduce in this Chapter the relevant networking concepts. Computer networks are composed of layered protocols. While the classical OSI model defines seven layers, we can simplify it to five layers, each of them relying on the lower layer to provide service to the upper one. First, the *physical layer* handles the actual electrical/optical transmission between two hosts using a given medium. Second, the *data-link layer* relies on the underlying one to define frames exchanged between two directly connected hosts. Third, the *network layer* enables non-directly attached hosts to exchange packets between each other thanks to intermediate nodes forwarding them. Fourth, the *transport layer* defines end-to-end communications between two hosts and manages their multiplexing. Fifth, the *application layer* consists in the actual data exchange between applications running on hosts.

From a network viewpoint, this layered architecture leads to the packet structure described in Figure 1.1. From the transport layer's perspective, the application data constitutes a payload to carry. To perform its operation, the transport protocol prefixes this payload with a specific header. Then, the network protocol receives the transport segment that forms the payload of the network layer. Again, a header is added before the payload to enable the network protocol to operate on that packet, and so on with the data-link header. With such a structure, each layered protocol can process the packet by looking only at its dedicated header — corresponding to the first packet bytes — without performing any operation on the remaining opaque payload.

This thesis focuses on *transport layer protocols*. To understand on what they build, we first introduce the IP addressing scheme used by the *network layer* (§1.1). Then, we introduce TCP, the most widely used transport protocol

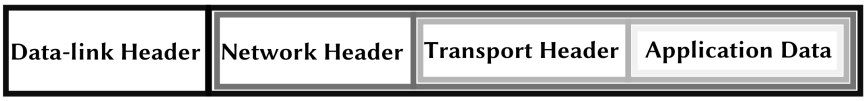


Figure 1.1: The influence of the computer network layering on the structure of in-flight data.

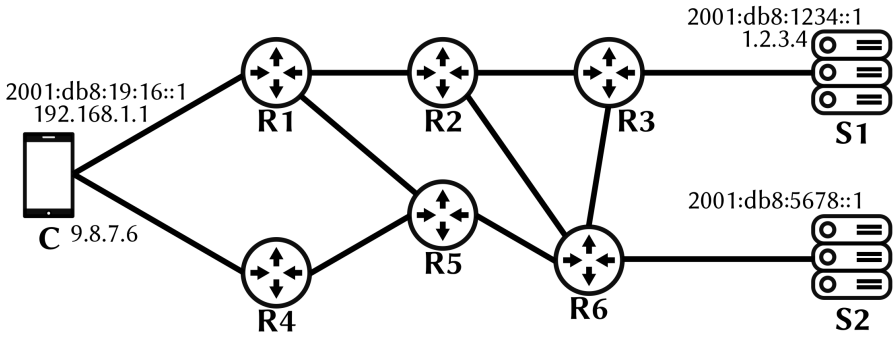


Figure 1.2: An example of IP network containing a client (C), two servers (S1 and S2) and six routers (R1 - R6).

(§1.2). Finally, we describe how TCP was extended to support the simultaneous usage of multiple paths (§1.3).

1.1 Addressing Hosts within IP Networks

Nowadays, Internet Protocol (IP) networks connect billions of devices. They provide addressing schemes such that intermediate routers can forward packets on a best-effort manner from a source node to a destination one. Figure 1.2 provides an example of such an IP network. Two addressing schemes are currently used: IP version 4 (IPv4) [RFC791] and IP version 6 (IPv6) [RFC2460].

IPv4, designed in the early eighties, supports 32-bit IP addresses. Their human-readable representation follows the format $a.b.c.d$ where $a, b, c, d \in [0, 255]$. In the provided example, **C** has two IPv4 addresses — 192.168.1.1 and 9.8.7.6 — each assigned to a different interface, while **S1** has one bounded to its network interface — 1.2.3.4. IPv4 theoretically provides 2^{32} — about 4 billions — different addresses, which was a comfortable number forty years ago. Yet, there is now no more unallocated addresses in the IPv4 space [ICA11]. To address more than 2^{32} devices with IPv4, a first solution is to rely on Network Address Translators (NATs) [RFC3022] to temporarily map an IPv4 address to another one. A typical example is a Wi-Fi access point providing wireless connectivity to a set of devices sharing a pool of private addresses, e.g., from 192.168.1.1 to 192.168.1.254. All these devices share the same globally reachable IPv4 address from the servers' viewpoint. While allowing network operators to adapt IPv4 to the growing Internet without modifying hosts, NATs have two main drawbacks. First, they break the end-to-end principle of the Internet [RFC1958], as the source IP address given by the host behind the NAT is not the one observed by the server. Second, NATs complicate the network management, as a given IP address does not refer to the same host over time.

Another solution to tackle this IPv4 address exhaustion problem is to use

IPv6. This version extends the address length to 128 bits, enabling the theoretical support of 2^{128} devices — about 3.4×10^{38}. Their human representation is composed of eight groups of four hexadecimal digits separated by colons. Picking up our example network in Figure 1.2, **C** has the IPv6 address 2001:0db8:0019:0016:0000:0000:0000:0001. This form often contains a lot of leading zeros which can be omitted, and consecutive groups of four zeros can be replaced by the double colon notation (::). Hence, we often write the previous IPv6 address as 2001:db8:19:16::1.

Hosts can exchange packets if they both have an IP address of the same version. For instance, if **C** wants to transmit a packet on the link to **R4** to reach **S1**, it can send a packet with source IPv4 address 9.8.7.6 and destination one 1.2.3.4. The packet will arrive at **R4**. It will look at its routing table to figure out on which network interface it can reach **S1** and then forward the packet to the next router. This process repeats on each router until the packet eventually reaches **S1**, which then delivers the packet's payload to the transport layer. The entries contained in the routers' forwarding table are handled by routing protocols such as Open Shortest Path First [RFC2328], Integrated System to Integrated System [RFC1195] and Border Gateway Protocol [RFC4271], although their description is outside the scope of this background. Similarly, if **C** wishes to communicate with **S2**, it can generate packets with source IPv6 2001:db8:19:16::1 and destination one 2001:db8:5678::1.

Notice that packets can be exchanged only between two IP addresses of the same version, either IPv4 or IPv6. In our example, since **C** only have an IPv4 address on the link to **R4** and **S2** is only reachable using IPv6, **C** cannot send packets to that destination on the lower link. Still, some network operators deployed NAT64 devices [RFC6146] enabling IPv6-only clients to reach IPv4-only servers.

When a device has simultaneously access to several networks, we call it *multi-homed*. In our example, **C** is multi-homed as it can send packets via either **R1** or **R4**. In the opposite situation, a device like **S2** is *single-homed* as it has only one network access point. The case of **S1** is debatable. We will consider it to be *single-homed*, although previous works [Dha+12] show that the underlying networks serving either IPv4 or IPv6 are not always the same.

1.2 Enabling Reliable Data Exchange with TCP

IP networks do not guarantee perfect delivery. In particular, packets may be lost, corrupted or delivered to the destination in a different order than sent by the source. To cope with these imperfections, end-hosts typically rely on the Transmission Control Protocol (TCP) [RFC793]. TCP guarantees *reliable*, *in-order* bidirectional byte-stream data delivery. Figure 1.3 describes its header. A TCP connection is identified by the 4-tuple (IP $_{\text{src}}$, IP $_{\text{dst}}$, port_{src}, post_{dst}).

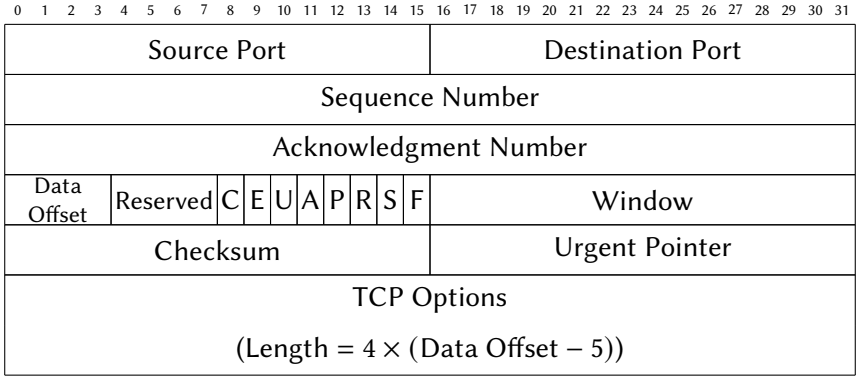


Figure 1.3: The TCP header [RFC793].

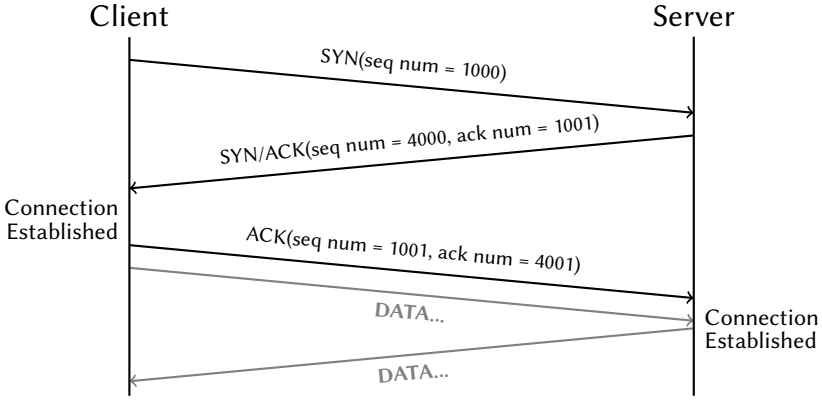


Figure 1.4: Establishing a TCP connection.

A TCP connection is established as shown in Figure 1.4. A client initiates a connection by sending a SYN packet, i.e., with the S bit set. This packet also contains a randomly chosen sequence number for the client to server data flow — 1000 in our example. Upon reception of this initial SYN, the server replies with a SYN/ACK packet, i.e., both S and A bits are set. It acknowledges the SYN packet by setting its acknowledgment number to the initial sequence number chosen by the client plus one — 1001 in Figure 1.4. This acknowledgment number indicates the next byte the packet's sender expects to receive next. In addition, the server also selects a random initial sequence number for the server to client data flow — 4000 in our case. Once the client receives the SYN/ACK packet, it replies with an ACK packet — A bit set — acknowledging it. At this point, the client considers the TCP connection to be established, and can start sending data to the remote host. At the other side, the server must wait for the ACK packet acknowledging its SYN/ACK one before starting sending data over the connection. This connection establishment process is often called the TCP three-way handshake, in reference to the number of packets required to establish the exchange.

TCP ensures reliable, in-order byte-stream data delivery thanks to its sequence and acknowledgment numbers. A receiver delivers data to its upper application only if the sequence number of the incoming data packet matches the expected acknowledgment one. When it receives a data packet, an host replies with an ACK packet with the possibly updated acknowledgment number. However, due to the unreliable nature of the underlying IP networks, packets might be either corrupted, lost or delayed. If a packet faces bit flips during its transmission, the receiver will notice it thanks to the Checksum field and discard the packet, making this equivalent to a loss. In these cases, the receiver will receive data packets which have higher sequence numbers than the one expected. The data contained in these packets cannot be delivered to the application yet, as TCP ensures in-order delivery. Instead, the receiver maintains a buffer where it keeps all these out-of-order packets. With such system, once the expected sequence number packet arrives, the host can deliver it with all the previously received ones without requiring the retransmission of all packets. An end-host advertises the size of the receive buffer to its peer using the Window field.

A sender can detect the loss of TCP packets by two ways. The first one relies on the reception of several duplicate ACK packets, i.e., ACK with the same acknowledgment number. As an example, consider a situation where a host sends five data packets, and the second one never reaches the peer. The receiver replies to the first data packet with an ACK indicating that it is ready for the second data packet. Then, it receives the third, the fourth and the fifth ones, which are not expected. The receiver stores them in its receive buffer and replies to each of the packets with ACK ones still advertising that

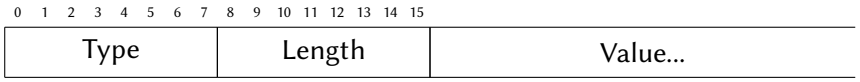


Figure 1.5: Format of a TCP option, excluding End-of-options (Type 0) and No-op (Type 1).

it expects the reception of the second data packet. If all these Acks reach the data sender, it can infer that the second packet was lost and perform a fast retransmission of this particular packet to quickly recover the data transfer. However, this approach is not sufficient to cope with all loss patterns. Consider again our example, but this time the fifth data packet never reaches the receiver. In such situation, the server will not observe duplicate Ack packets. To cope with such situation, the sender launches a timer when a packet is sent. If the packet is not acknowledged before the timer fires, a Retransmission Timeout (RTO) occurs and the sender retransmits the lost packet. Such retransmission strategies ensure the reliability of TCP transfers.

TCP connections share the common Internet infrastructure, and at some point several of them might compete for the same physical link. If their network usages exceed the actual capacity, connections will create congestion by filling router buffers. This network pressure increases the experienced latency and induces packet losses, involving a decrease of the quality of all exchanges through this bottleneck link. To prevent Internet collapse, TCP includes congestion control schemes limiting the transmission rate with a congestion window. Many algorithms have been proposed [BP95; rfc2582]. The current default one in Linux is CUBIC [HRX08].

A TCP connection can be terminated either gracefully or abruptly. In the first case, hosts perform a three-way connection release with Fin (packet with the F bit set), Fin/Ack and Ack packets, similarly to the three-way connection handshake. This method ensures that all data sent over the connection are eventually delivered before the exchange effectively closes. However, there are cases where a host wants to directly tear down a connection. To do so, the end point sends to its peer a Rst packet — the R flag is set. Notice that this abrupt closing does not ensure the reliable delivery of in-flight, unacknowledged data.

1.2.1 Extending TCP with Options

Extensibility is a key requirement for protocols to adapt to the evolution of the Internet. To support extensions, TCP includes the TCP Options field in its header, as illustrated in Figure 1.3. Except the End-of-options (Type 0) and No-op (Type 1) options each being one byte long, TCP options follow the Type-Length-Value convention as shown in Figure 1.5. The Length field

indicates the size of the whole option — at least 2 — and Value is a (Length−2)-byte long field whose format depends on the Type field. Notice that the option space is limited to 40 bytes due to the 4-bit encoding of the Data Offset field.

The usage of an extension over a specific connection is negotiated during its three-way handshake. The client includes in the SYN packet all the options it wants to use. For its part, the server replies in the SYN/ACK packet with all the client-provided options it wants to support, i.e., both hosts need to agree to adopt a given extension. Typical Linux hosts include in SYN packets the Maximum Segment Size (Type 2, 4 bytes), the Selective Acknowledgment (SACK) Permitted (Type 4, 2 bytes), the Timestamp (Type 8, 10 bytes) and Window Scale (Type 3, 3 bytes) options. The current length of the TCP Options field in SYN packets is thus 20 bytes (19 bytes plus one padding byte), leaving 20 bytes for future extensions. In the TCP packets that carry data, we mainly see the Timestamp option occupying 12 bytes (10 plus two padding ones). This leaves 28 bytes for options in non-SYN packets, which can be either used by the SACK option or other extensions.

1.2.2 Coping with Middleboxes

Figure 1.2 presents a simple network with end-hosts generating packets and routers forwarding them. Unfortunately, today's networks are much more complex, containing devices which act neither as end-hosts nor as routers. Computers connected to Wi-Fi networks are often behind a NAT changing both IPs and ports of a packet. Enterprise networks often contain load balancers to spread the traffic across servers while relying on firewalls and intrusion detection systems to drop suspicious traffic [She+12]. The cellular network is known to contain both NATs and firewalls [Wan+11]. We refer to these middle devices as *middleboxes*. They form an important part of today's networks, being sometimes more numerous than plain routers [She+12].

Previous works [Hon+11; Det+13] show that all the fields of a packet, including its payload, can be altered by middleboxes. Such network interference hindered the deployment of some TCP options. For instance, the SACK option was first defined in 1996 [RFC2018], although it took about ten years before being usable on the Internet [Fuk11]. Since some middleboxes rewrite the TCP sequence numbers [Hon+11], they also need to adapt the ones in the SACK options. However, not all these middleboxes were directly behaving well with the SACK option, and there was a transition period where it was not usable. Hence, we need to take these middleboxes into account when designing TCP extensions if we want them to be deployed on the Internet.

1.3 Using Multiple Network Paths with Multipath TCP

Designed in the early eighties, TCP enables a data exchange between hosts over a specific 4-tuple (IP $_{src}$, IP $_{dst}$, port$_{src}$, post$_{dst}$) acting as its connection identifier. This implicitly induces that the exchange can only take place over a single network path. Let us return to our example in Figure 1.2. **C** wants to connect to 1.2.3.4, pointing to **S1**. It can start then a connection using the path to **R1** with 192.168.1.1. At some point, **C** may want to leverage the path to **R4** with 9.8.7.6 to let the exchange benefit from the available network resources, e.g., to aggregate bandwidth or to increase network resiliency. However, TCP does not allow such resource pooling, as this path would lead to another 4-tuple different from the one of the original connection.

Multipath TCP [RFC6824] aims at solving this issue at the transport layer. Its design goals [Rai+12] are the following.

- Multipath TCP should be able to use several network paths for a single connection;
- Multipath TCP should work in all networks where TCP works;
- Multipath TCP should be able to use the network at least as well as regular TCP, but without starving TCP;
- Multipath TCP should be implementable in operating systems without using large memory and processing.

To achieve these goals, Multipath TCP uses the architecture depicted in Figure 1.6. Multipath TCP is designed as a TCP extension relying on TCP options. Applications interact with TCP using the Socket API. The usage of multiple paths does not change anything from the applications' viewpoint, enabling them to use Multipath TCP without requiring any code change. With regular TCP, the Socket API makes the link between the application and its underlying TCP connection. When using Multipath TCP, the Socket API instead communicates with an additional meta-connection — the Multipath TCP connection. This intermediate agent enables the application to keep a single link with the Multipath TCP connection, independently of the actual numbers of TCP paths being used. We often refer to these TCP connections forming the Multipath TCP paths as the TCP subflows of the Multipath TCP connection. The whole management of these TCP subflows is hidden from the application.

From a concrete viewpoint, Multipath TCP relies on TCP options. As we will see next, Multipath TCP requires several control messages for its operations. They all follow the pattern illustrated in Figure 1.7. Notice that this format respects the TCP Type-Length-Value one shown in Figure 1.5. The advantage of this specific pattern — using the Subtype field to identify a

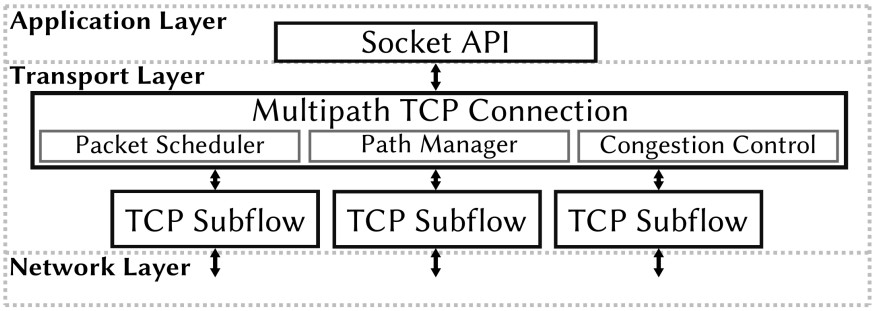


Figure 1.6: High-level architecture of Multipath TCP, here with a connection having three subflows.

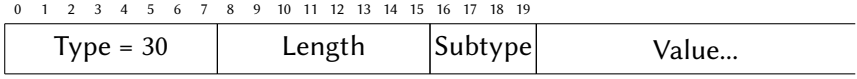


Figure 1.7: Format of a Multipath TCP option.

Multipath TCP option — is to reserve only one Type value (30) for the whole Multipath extension, leaving more space for future ones.

The negotiation of the Multipath extension is performed during the TCP 3-way handshake, similar to any other TCP extension (§1.3.1). Once both peers agree on its usage, they can attach new TCP subflows to the Multipath TCP connection (§1.3.2). Because hosts might not be aware that their peer have several addresses — in Figure 1.2 **C** might not know that **S1** has an IPv6 address — Multipath TCP includes mechanisms to advertise them (§1.3.3). Next, hosts can spread data over their available paths (§1.3.4). Notice that these multipath operations involve the use of additional algorithms such as the packet scheduler and the path manager (§1.3.5). Finally, the Multipath TCP connection can be terminated, either gracefully or abruptly (§1.3.6).

1.3.1 Creating a Multipath TCP Connection

To negotiate the usage of multiple paths, Multipath TCP relies on the MP CAPABLE option (Subtype 0). The Multipath TCP connection establishment is depicted in Figure 1.8. The client first sends a SYN packet with the MP CAPABLE option containing a 8-byte key K_C. If the server supports Multipath TCP, it then replies with a SYN/ACK packet carrying the MP CAPABLE option along with the server's key K_S. Together, these keys serve as a basis to authenticate connection's peers when adding new paths. They also allow each peer to derive a 4-byte token identifying the Multipath TCP exchange. To confirm the Multipath TCP connection establishment, the client sends an ACK packet

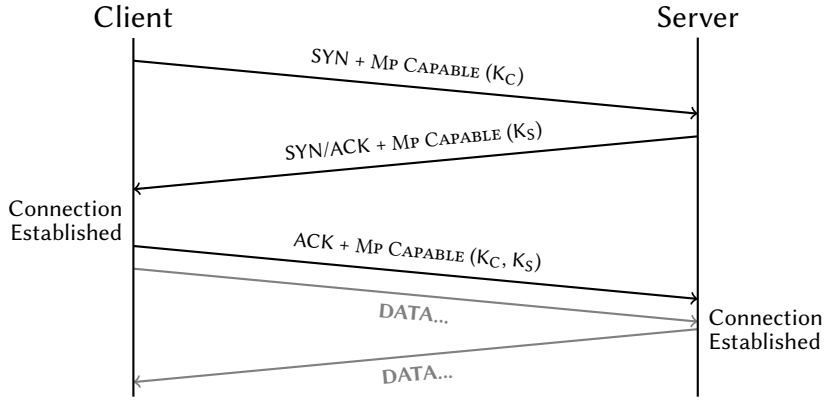


Figure 1.8: Establishing a Multipath TCP connection.

with MP CAPABLE containing both K_C and K_S. Echoing both keys enables the server to keep minimal state until the connection is fully established.

It may happen that the Multipath negotiation fails. The most common situation is when the contacted server does not support it. In such case, it replies with a SYN/ACK without the MP CAPABLE option, and the connection falls back to regular TCP. Another reason can be middleboxes which remove the MP CAPABLE option. Again, the host then continues with regular TCP. In any case, the Multipath negotiation failure does not prevent connectivity between hosts.

1.3.2 Adding Paths to an Existing Multipath TCP Connection

Once the establishment completed, additional paths can be added to an existing Multipath TCP connection. As illustrated in Figure 1.9, Multipath TCP requires a four-way handshake before allowing the transmission of data on a new subflow. This handshake serves two purposes. First, it creates state on the endpoints, and possibly on the intermediate middleboxes. Second, both the client and the server authenticate each other. This authentication is performed by using the keys exchanged during the handshake. First, the client sends a SYN with the MP JOIN option illustrated in Figure 1.10. It contains both the token identifying the connection at server side ($Token_S$) and a random value (R_C). The server then retrieves the connection associated with the token, generates a random value R_S and computes the 20-byte long Hash-based Message Authentication Code (HMAC) $HMAC_S$ as follows

$$HMAC_S = HMAC_{K_C \,||\, K_S}(R_C \,||\, R_S) \tag{1.1}$$

where K_C and K_S are respectively the connection keys of the client and server that were exchanged during the Multipath TCP connection handshake. The

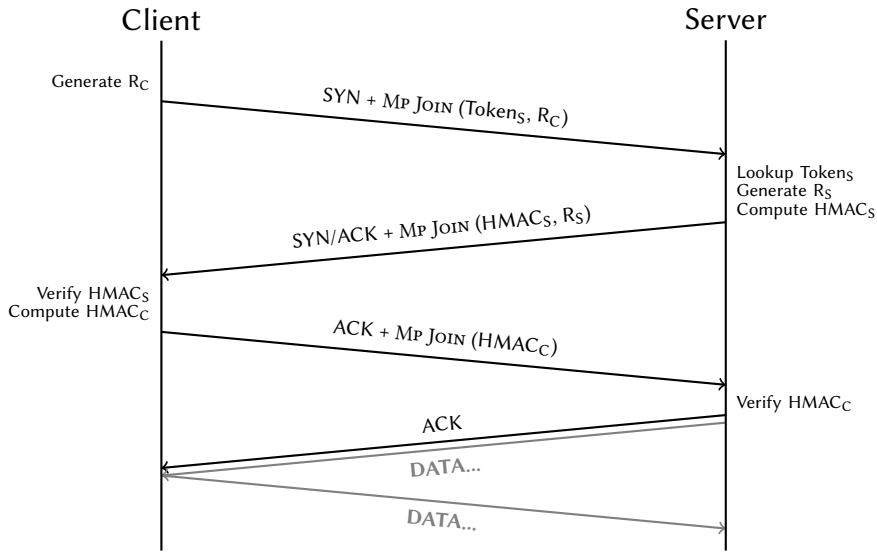


Figure 1.9: Establishing an additional Multipath TCP subflow.

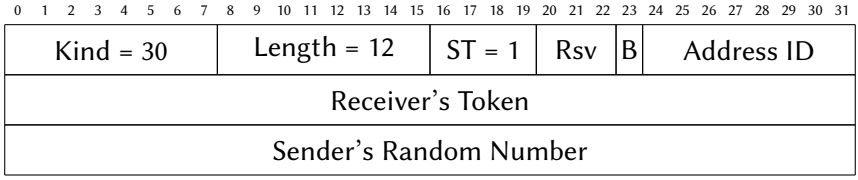


Figure 1.10: The MP JOIN option in the initial SYN.

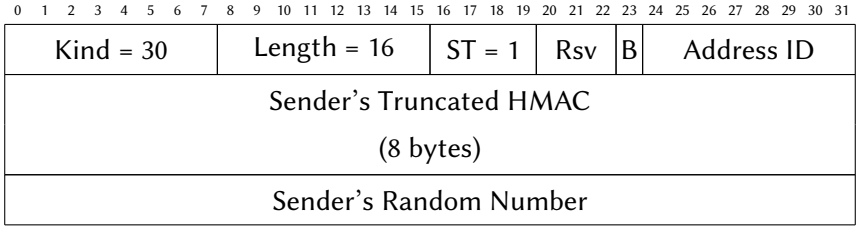


Figure 1.11: The MP JOIN option in the SYN/ACK.

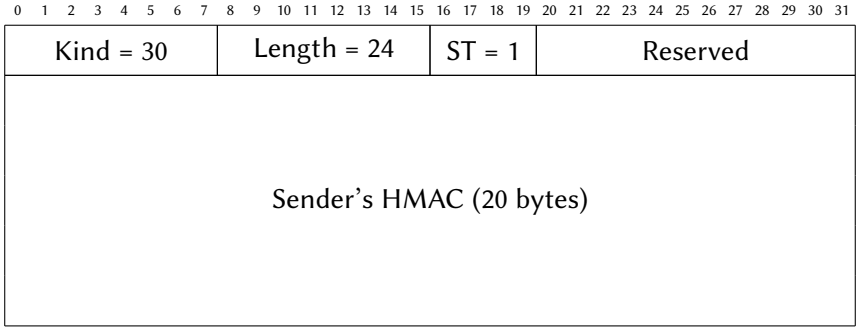


Figure 1.12: The MP JOIN option in the third ACK.

server then replies with a SYN/ACK packet containing the MP JOIN option shown in Figure 1.11. It carries both R_S and the 8-byte truncated $HMAC_S$. The need for the truncation comes from the limited TCP option space in the SYN/ACK packet. Transmitting the full 20-byte HMAC along with other MP JOIN fields would have required 28 bytes, while only 20 are available. Upon reception, the client can verify that the provided (truncated) $HMAC_S$ matches the value computed using Equation 1.1. If it does, the client then computes the $HMAC_C$ as follows

$$HMAC_C = HMAC_{K_S \,||\, K_C}(R_S \,||\, R_C) \tag{1.2}$$

and sends an ACK with the MP JOIN option presented in Figure 1.12. This one contains the full computed $HMAC_C$, as there remains more TCP option space in non-SYN packets than SYN ones. The server next checks that the provided value matches the one of Equation 1.2. If it is the case, the server replies with an ACK packet and the data transfer can start from that point.

Given the high diversity of network paths, hosts may want to prioritize some subflows using specific networks over other ones. When negotiating the use of a new path, both the client (Figure 1.10) and the server (Figure 1.11) can request its peer to treat the subflow as a backup one by setting the B bit in the MP JOIN option. A Multipath TCP host starts using backup paths once all non-backup ones failed.

Notice that at any point, the four-way handshake may fail. It might be because either one of the peers is not part of the initial connection's participants — hence failing the authentication — or the middleboxes remove the MP JOIN option from packets. In these cases, the attempted path cannot be used by the Multipath TCP connection and the TCP subflow is torn down using a RST packet. Although Multipath TCP cannot leverage that potential path, the connectivity of the related connection is not altered. In other words, the failed path addition does not prevent hosts from continuing to exchange data

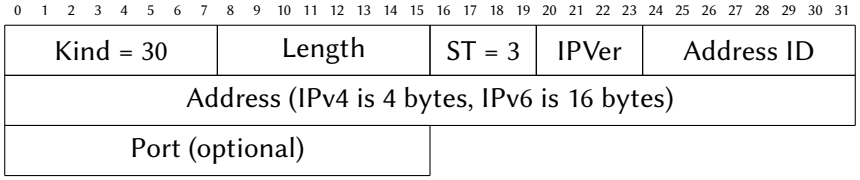


Figure 1.13: The ADD ADDR option advertising available addresses.

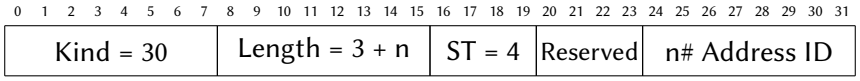


Figure 1.14: The REMOVE ADDR option advertising lost addresses.

over the other paths of the Multipath TCP connection.

1.3.3 Communicating Additional Addresses to the Peer

While a host is aware of all its local addresses, it might not know the ones of its peer. At the beginning of a connection, the client knows its addresses and the peer's one used to contact it. The server might be interested in sharing its other addresses to establish additional subflows over them. To this end, Multipath TCP provides the ADD ADDR option shown in Figure 1.13. It supports both IPv4 and IPv6 addresses thanks to its IPVer field determining the length of the Address field. Each of these communicated addresses are identified by an Address ID. This connection-specific identifier enables referencing the address in other options without rewriting it. For instance, the previously presented MP JOIN option contains the local source Address ID used to create the new subflow. This allows the peer to figure out if the 2-tuple (IP_{src}, $port_{src}$) was altered in transit by, e.g., a NAT.

Similarly, a host can advertise the loss of an address using the REMOVE ADDR option depicted in Figure 1.14. Note that since lost addresses were previously advertised, hosts can refer to them by using their corresponding Address ID. While being useful for path management, these options are not critical for the multipath operations. Therefore, Multipath TCP does not require their reliable delivery, i.e., both sent ADD ADDR and REMOVE ADDR options may never reach the remote host.

1.3.4 Exchanging Data over Multiple Paths

As it is built atop TCP, Multipath TCP needs to provide reliable, in-order data delivery. While this service is guaranteed over each TCP path, this does not ensure in-order delivery when spreading data over multiple subflows. For in-

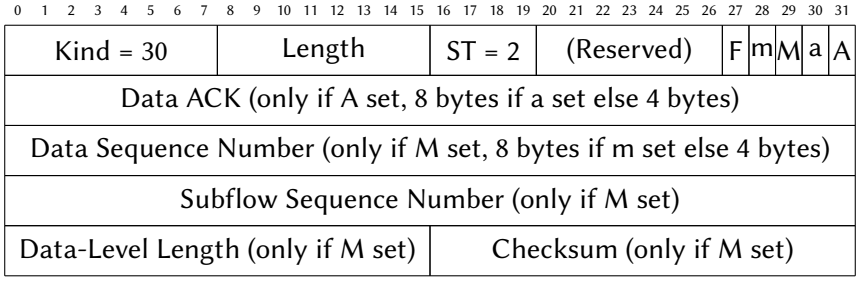


Figure 1.15: The Data Sequence Signal (DSS) option.

stance, if the first data A is sent on a slow path and the second one B on a fast subflow, the peer will receive B before A. To handle such reordering between paths, an additional sequence number space is required. The Data Sequence Signal (DSS), illustrated in Figure 1.15, provides the Multipath TCP Data Sequence Number (DSN) and its corresponding Data ACK. It notably enables Multipath TCP to retransmit a specific piece of data over another subflow. Such event is called a *reinjection*. All Multipath TCP connection packets carry either the MP CAPABLE, the MP JOIN or the DSS options. Notice that in addition to the Multipath TCP sequence number spaces, the DSS includes three other fields (Subflow Sequence Number, Data-Level Length and Checksum) enabling Multipath TCP to cope with possible network interference.

1.3.5 Multipath-specific Algorithms

Once the Multipath TCP connection is established, the host has several decisions related to multipath to take. They specifically cover the creation and deletion of paths and their usage. The simultaneous use of several paths also raises concerns about the fairness against regular TCP.

Path Manager. This component determines when and between which addresses additional subflows should be created. The default `full-mesh` path manager in the Linux implementation creates a subflow between each pair of compatible addresses as soon as possible. Considering the example in Figure 1.2, there would be 3 subflows between **C** and **S1**, 2 using IPv4 and 1 using IPv6. These path creation decisions are only taken by the client, as middleboxes like NATs usually prevent the server from reaching its peer.

Packet Scheduler. This agent determines on which established subflow a given data packet will be sent. The `default` packet scheduler in the Linux implementation selects the lowest estimated latency path whose congestion window enables data sending. The Round-Trip-Time (RTT) of a TCP path can be estimated by observing the delay between the instant the packet is

sent and the moment it is acknowledged, and those observations are then smoothed [Jac95].

Coupled Congestion Control. Classical TCP congestion controls such as CUBIC aims to share a fair amount of the bottleneck link between competing connections. If a CUBIC Multipath TCP host establishes a lot of subflows over this bottleneck link, it will starve other TCP ones. Coupled congestion controls, such as LIA [RFC6356; Rai+12] and OLIA [Kha+12], tackle this unfairness issue by controlling the overall aggressiveness of the multipath connection.

1.3.6 Terminating the Connection

Like regular TCP, a Multipath TCP connection can be closed in two ways. First, the graceful DATA FIN method consists in setting the F bit of the DSS option. Once both hosts have advertised and acknowledged the DATA FIN, the connection is closed. This ensures that all in-flight data is eventually delivered. Second, the abrupt method resides in sending a packet carrying the FAST CLOSE option (Subtype 7). In that case, nothing ensures the reliable delivery of unacknowledged data.

The definition of these additional signals enables Multipath TCP to keep the scope of the FIN and RST packets at subflow level. For instance, a subflow can be gracefully or abruptly closed without terminating the related connection itself. This allows Multipath TCP connections to continue their operations while escaping from possibly bad network paths.

Evaluating Multipath TCP on Smartphones

2

In the last decade, the Internet has seen the rise of mobile devices such as smartphones, tablets and connected cars. To handle user mobility, these devices are typically multi-homed. This means that they can be simultaneously attached to several networks. Considering smartphones, they usually have both cellular and Wi-Fi connectivity. The multi-homing ability enables mobile devices to switch from one wireless network to another one without any user intervention.

Currently, TCP is the dominant transport protocol, both on the wired Internet and in wireless networks. However, plain TCP connections are bound to their 4-tuple (IP_{src}, IP_{dst}, $port_{src}$, $port_{dst}$). This means that a TCP connection cannot benefit from the availability of multiple wireless network accesses. For instance, if a smartphone loses Wi-Fi connectivity, all its previously established TCP connections using that network interface will be torn down. In such mobile situations, the transport protocol does not ensure reliable data delivery anymore, although Internet connectivity may still be provided over another available network access.

Multipath TCP [RFC6824; Rai+12] aims at fixing the mismatch between the capabilities of modern devices and TCP limitations. Once standardized in 2013, Multipath TCP quickly captured the interest of industry to support various commercial services. In September 2013, Apple realized the first large-scale deployment of Multipath TCP to enhance the user experience of the Siri voice recognition application [Bon13] and then extended this multipath capability to all iOS applications in September 2017. In July 2015, Korean Telecom (KT) enabled high-end Android devices to use Multipath TCP to aggregate both Wi-Fi and cellular networks [KT]. These smartphones can achieve download rates of over 1 Gbps. Yet, despite these deployments, little is known about the actual performance of Multipath TCP on mobile devices.

This Chapter fills this gap by presenting two different measurement campaigns evaluating how well Multipath TCP performs on smartphones. We first explore the related works about Multipath TCP measurements (§2.1). Next, we evaluate how Multipath TCP behaves on Android smartphones using the Multipath TCP implementation in the Linux kernel [MPTCPLK] (§2.2). We then test Apple's Multipath TCP implementation used by the iOS de-

vices [BS16] under specific scenarios (§2.3). Finally, we summarize the lessons learned from our experiments (§2.4).

2.1 Related Work

In this Section, we classify the studies of various researchers that have analyzed the performance of Multipath TCP, and how our work differs from the previous contributions.

Smartphone environment. As shown by Raiciu et al. [Rai+11a], Multipath TCP achieves its bandwidth aggregation goal in homogeneous wired environments such as data centers. However, the measurements of both Falaki et al. [Fal+10] and Huang et al. [Hua+10] reveal the heterogeneity of wireless networks used by smartphones. They also show that smartphone applications generate more downlink than uplink traffic. Unlike in our measurement campaigns, these smartphone studies focus on the performance of plain TCP.

Multipath TCP in wireless networks. Chen et al. [Che+13] analyze the performance of Multipath TCP in Wi-Fi and cellular networks by running bulk transfer applications on laptops. They find that Multipath TCP achieves completion times comparable to TCP over the fastest network. Chen et al. [CT14] explore the impact of the cellular bufferbloat when performing bulk data transfers with Multipath TCP. They also evaluate the impact of applying the Opportunistic Retransmission and Penalization algorithm [PKB13]. Ferlin et al. [FDA14a] confirm that cellular networks exhibit bufferbloat which affects the performance of Multipath TCP. Ferlin et al . [FDA14b] propose a probing technique to detect low performing paths and evaluate it in wireless networks. Deng et al. [Den+14] compare the performance of single-path TCP over WiFi and LTE networks with Multipath TCP on multi-homed devices by using active measurements and replaying HTTP traffic observed on mobile applications. They show that Multipath TCP provides benefits for long flows but not for short ones, for which the selection of the interface for the initial subflow is important from a performance viewpoint. Han et al.[Han+15] analyze the behavior of Multipath TCP to load web pages with both HTTP and SPDY. They confirm that large SPDY flows are better suited for Multipath TCP than smaller HTTP ones. Different packet schedulers such as BLEST [Fer+16] and ECF [Lim+17] have been proposed to operate in wireless networks. Still, all these previous works mostly focus on the bandwidth aggregation feature of Multipath TCP under bulk transfers and do not explore network handovers.

Device mobility with Multipath TCP. Raiciu et al. [Rai+11b] first discuss how Multipath TCP can be used to support mobile devices either by di-

rect client/server connections or through a Multipath TCP proxy. They also provide early measurement results using a laptop. Paasch et al. [Paa+12] then propose three distinct path management policies to cope with different wireless networks. The *Full-MPTCP* mode creates subflows on all pairs of client's and server's addresses. These subflows are then simultaneously used to aggregate the bandwidth of the available networks. The *Backup* mode also establishes all the subflows, but only uses a subgroup. This mode is useful to prioritize some networks over other ones, e.g., to prevent hosts from using the cellular interface when Wi-Fi is still available, while keeping fast reaction if primary networks become unavailable. The *Single-Path* mode is similar to the *Backup* one, except that only one subflow is established at any time. These modes were included in the Linux implementation of Multipath TCP and their evaluation on a laptop showed that the network handovers are handled differently by each mode. Williams et al. [Wil+14] analyze the performance of Multipath TCP on moving vehicles with bulk transfers. Li et al. [Li+18] measure the performance of Multipath TCP in high-speed trains by leveraging the availability of multiple cellular networks. Our works continue in that vein by considering smartphones instead of laptops and by exploring other traffic patterns than the bulk one.

Other related Multipath TCP works. Hesmans et al. [Hes+15] analyze a one week-long trace collected at the reference server `multipath-tcp.org`. Compared to our works, they have no control on the devices contained in their network trace. Saha et al. [Sah+17] extend one of our previous work to evaluate the behavior of Multipath TCP under controlled smartphone application scenarios. They also explore how Multipath TCP impacts the energy consumption and CPU usage on such mobile devices.

2.2 Analyzing In-The-Wild Android Users

In this Section, we provide a detailed analysis of a Multipath TCP network trace generated by smartphones used by real users. We first introduce how we enable smartphones to benefit from Multipath TCP (§2.2.1). Then, we provide the basic characteristics of the studied network trace (§2.2.2). Finally, we take a closer look at the performance of Multipath TCP (§2.2.3).

2.2.1 Enabling Multipath TCP on Android Smartphones

Although Multipath TCP is already used by hundreds of millions of Apple's iOS smartphones, Multipath TCP is not yet supported at a large scale on Internet servers. Therefore, making the smartphones Multipath TCP capable is

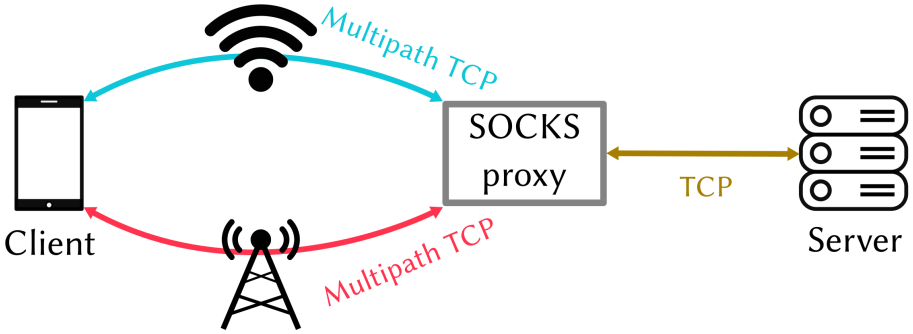


Figure 2.1: The deployed architecture allowing smartphones to use Multipath TCP.

not sufficient to generate Multipath TCP traffic. Indeed, most of the servers do not recognize the Multipath extension and fallback to plain TCP.

To address this deployment issue, we deployed the infrastructure depicted in Figure 2.1, which is the same architecture as the one used by Korean Telecom [KT]. As Android relies on the Linux kernel, one can integrate the Linux implementation of Multipath TCP on our smartphones. In our measurement campaign, the smartphones run Android 4.4 with a modified Linux kernel that includes Multipath TCP v0.89.5 [DB]. To force smartphone applications to use Multipath TCP, ShadowSocks[1] was installed on each smartphone and configured to use a SOCKS server that supports Multipath TCP (v0.89.5) for all TCP connections. The smartphones thus use Multipath TCP over their Wi-Fi and cellular interfaces to reach the SOCKS server. This proxy initiates Multipath TCP connections to the final destinations, but most of them fallback to regular TCP as very few servers on the Internet support Multipath TCP. Using the SOCKS proxy enables us to collect both Wi-Fi and cellular network traffic using `tcpdump` without requiring cooperation from an ISP. From the server side, all the connections from the smartphones appear as coming from the SOCKS server. This implies that the external (cellular or Wi-Fi) IP address of the smartphone is not visible to the contacted servers. This might affect the operation of some servers that adapt their behavior (e.g., the initial congestion window) in function of the client IP address. Moreover, our ShadowSocks client introduces two additional characteristics. First, it sends DNS requests over TCP. Second, ShadowSocks did not support IPv6 when we collected our measurements, so our trace only contains IPv4 packets.

As introduced in §1.3.5, the usage of Multipath TCP involves the operation of several dedicated algorithms. First, the *path manager* specifies the strategy used to create and delete subflows. In our measurements, smartphones

[1] Available at `http://shadowsocks.org`.

employ the `full-mesh` path manager that initiates one subflow over each pair of interface addresses as soon as the initial subflow has been fully established or as soon as a new address has been learned. Second, the *packet scheduler* [Paa+14] selects, among the active subflows having an open congestion window, the subflow on which the next packet containing data will be sent. Both the smartphones and the proxy used the `default` Multipath TCP scheduler that prefers the subflow with the smallest RTT. This scheduler also includes the Opportunistic Retransmission and Penalization (ORP) mechanism [PKB13] aiming at mitigating head-of-line blocking when facing receive window limitations. Third, the *congestion controller* defines the congestion window of each Multipath TCP subflow. Here, we rely on the standard Linked Increases Algorithm (LIA) [RFC6356].

Android 4.4 smartphones assume that only one wireless interface is active at a time. When such a smartphone switches from cellular to Wi-Fi, it automatically resets all existing TCP connections by using Android specific functions. To counter this behavior, we installed a special Android application[2] that enables the simultaneous usage of the Wi-Fi and cellular interfaces. It also controls the routing tables and updates the policy routes that are required for Multipath TCP every time the smartphone connects to a wireless network. Thanks to this application, the modified Nexus 5 can be used by any user since it does not require any networking knowledge.

The dataset covers the traffic produced by a dozen of users using Nexus 5 smartphones. These users were either professors, Ph.D. or Master students at UCLouvain. While some of them used their device to go only on the Internet, others were still using them as their main phone after our measurement campaign. Measurements were performed in Belgium from March 8th to April 28th 2015. Over this period of 7 weeks, more than 71 millions Multipath TCP packets were collected for a total of 25.4 GBytes over 390,782 Multipath TCP connections.

2.2.2 Characterization of the Trace

Before going into a more detailed analysis, we first look the main characteristics of the Multipath TCP connections in our dataset.

Destination port. Usually, the application level protocol can be identified by looking at the destination port of the captured packets. However, as smartphones connect through a SOCKS proxy, all the packets are sent towards the destination port used by the proxy, which is 443 in our case to prevent middlebox interference. To find the application level protocol, we extract the

[2] Available at `https://github.com/MPTCP-smartphone-thesis/MultipathControl`.

Port	# connections	% connections	Bytes	% bytes
53	107,012	27.4	17.4 MB	< 0.1
80	103,597	26.5	14,943 MB	58.8
443	104,223	26.7	9,253 MB	36.4
4070	571	0.1	91.7 MB	0.4
5228	10,602	2.7	27.3 MB	0.1
8009	10,765	2.8	0.97 MB	< 0.1
Others	54,012	13.8	1,090 MB	4.3

Table 2.1: Statistics about destination port fetched by smartphones.

destination port from the SOCKS command sent by the `ShadowSocks` client at the beginning of each connection. As shown in Table 2.1, most of the connections and data bytes are related to Web traffic (ports 80 and 443). Since `ShadowSocks` sends DNS requests over TCP, it is expected to have a large fraction of the connections using port 53. Among other popular port numbers, there are ports such as 4070 — e.g., used by Spotify —, Google Services (5228) and Google Chromecast (8009).

Duration of the connections. 65% of the observed connections last less than 10 seconds. In particular, 4.3% are failed connections, i.e., the first SYN was received and answered by the proxy, but the third ACK was lost (or a RST occurred). 20.8% of the connections last more than 100 seconds. Six of them last for more than one entire day (up to nearly two days).

Bytes carried by connections. Most (86.9%) of the connections carry less than 10 KBytes. In particular, 3.1% of the connections carry between 9 and 11 bytes. Actually, those are empty connections, since the SOCKS command are 7 bytes long, two bytes are consumed by the SYNs and the use of the remaining two bytes depend on how the connections were closed (RST or FIN). The longest connection in terms of bytes transported around 450 MBytes and was spread over five subflows.

2.2.3 Analysis

In the following, the analysis will focus on relevant subsets of the trace such as connections with at least two subflows, connections using at least two subflows or connections experiencing handover. Table 2.2 gives the characteristics of these subsets. They help to analyze how Multipath TCP subflows are created (§2.2.3.1), study the heterogeneity of the available networks in

Name	Description	# connections	Bytes S→P	Bytes P→S
$\mathcal{T}_0$	Full trace	390 782	652 MB	24 771 MB
$\mathcal{T}_1$	≥ 2 established subflows	126 040	238 MB	13 496 MB
$\mathcal{T}_2$	≥ 2 used subflows	32 889	152 MB	11 856 MB
$\mathcal{T}_3$	With handover	8 461	36.7 MB	4 626 MB

Table 2.2: The different (sub)traces analyzed in this section. S→P: from smartphones to proxy. P→S: from proxy to smartphones.

Number of subflows	1	2	3	4	5	>5
Percentage of connections	67.75%	29.96%	1.07%	0.48%	0.26%	0.48%

Table 2.3: Number of subflows per Multipath TCP connection.

terms of round-trip-times (§2.2.3.2), estimate the packet reordering of Multipath TCP (§2.2.3.3), study how subflows are used (§2.2.3.4), quantify the reinjection overhead (§2.2.3.5) and identify connections experiencing handovers (§2.2.3.6).

2.2.3.1 Creation of the Subflows

With Multipath TCP, a smartphone can send data over different paths. The number of subflows that a smartphone creates depends on the number of active interfaces that it has and on the availability of the wireless networks.

Table 2.3 reports the number of (not necessarily concurrent) subflows that are observed in $\mathcal{T}_0$. Most of the connections only have one subflow. On another side, 2.29% of the connections have more than two subflows. Having more subflows than the number of network interfaces is a sign of mobility over different Wi-Fi and/or cellular access points since IPv6 was not used. A connection establishing 42 different subflows was observed.

Another interesting point is the delay between the establishment of the connection (i.e., the first subflow) and the establishment of the other subflows. As it relies on the `full-mesh` path manager, the smartphone tries to create subflows shortly after the creation of the Multipath TCP connection and as soon as a new interface gets an IP address. Late joins can mainly be expected when a smartphone moves from one network access point to another. To quantify this effect, Figure 2.2 plots the cumulative distribution function (CDF) of the delays between the creation of each Multipath TCP connection and all its additional subflows that are linked to. 65% of all the additional subflows are established within 200 ms. This percentage increases to 73% if this limit is set to one second. If the analysis is restricted to the first additional

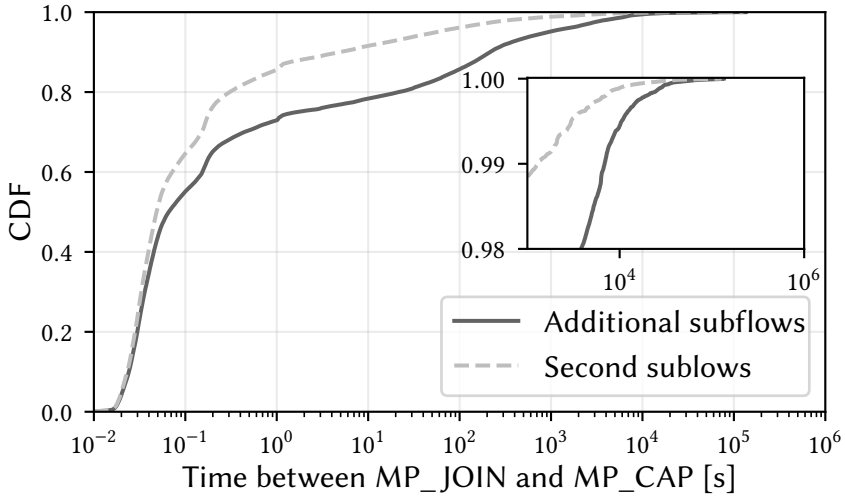


Figure 2.2: Delay between the creation of the Multipath TCP connection and the establishment of a subflow.

subflow, these percentages are respectively 76% and 86%. Joins can occur much after the connection is established. Indeed, 17% of the additional subflows were established one minute after the establishment of the connection, and 2.1% of them were added one hour later. The maximal observed delay is 134,563 seconds (more than 37 hours) and this connection was related to the Google Services. Those late joins suggest network handovers, and late second subflow establishments can be explained by smartphones having only one network interface available, e.g., switching from a Wi-Fi access point to another one while having cellular connectivity turned off.

2.2.3.2 Subflow Round-Trip-Times

From now on, we focus on the subtrace $\mathcal{T}_1$ that includes all the connections with at least two subflows. A subflow is established using a three-way handshake. Thanks to this exchange, the communicating hosts agree on the sequence numbers and TCP options and also measure the initial value of the round-trip-time for the subflow. For the Linux implementation of Multipath TCP, the round-trip-time (RTT) measurement is an important performance metric because the `default` packet scheduler prefers the subflows having the lowest RTTs.

To evaluate the RTT heterogeneity of the Multipath TCP connections, the analysis uses `tstat` [MCC03] to compute the average RTT over all the subflows that a connection contains. Then, it extracts for each connection the

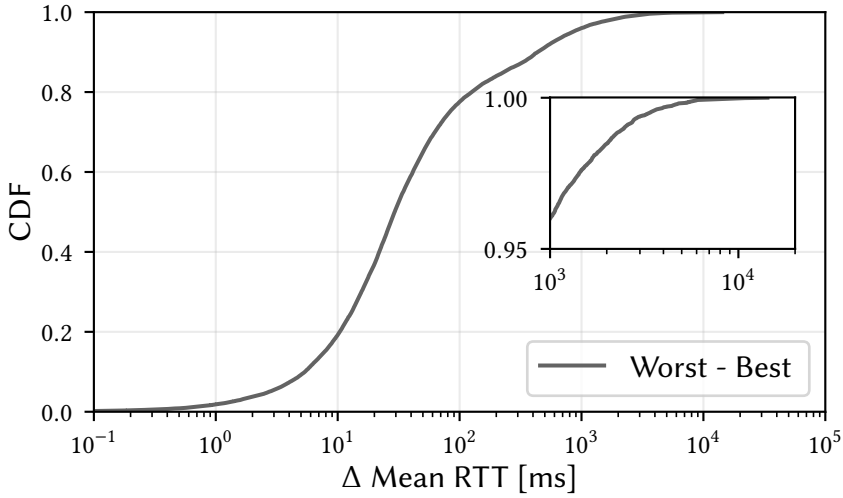


Figure 2.3: Difference of average RTT seen by the proxy between the worst and the best subflows with at least 3 RTT samples.

minimum and the maximum of these average rRTTs. To have consistent values, it only takes into account the subflows having at least 3 RTT estimation samples. Figure 2.3 plots the CDF of the difference in the average RTT between the subflows having the largest and the smallest RTTs over all connections in $\mathcal{T}_1$. Only 19.2% of the connections are composed of subflows whose round-trip-times are within 10 ms or less whereas 77.5% have RTTs within 100 ms or less. 4% of the connections experience subflows having 1 second or more of difference in their average RTT. With such network heterogeneity, if a packet is sent on a low-bandwidth and high-delay subflow s_0 and following packets are sent on another high-bandwidth low-delay one s_1, the sender may encounter head-of-line blocking.

2.2.3.3 Multipath TCP Acknowledgements

As explained in §1.3, Multipath TCP uses two acknowledgment levels: the regular TCP Acks at the subflow level and the cumulative Multipath TCP Acks at the connection level. It is possible to have some data acknowledged at the TCP level but not at the Multipath TCP one, typically if previous data was sent on another subflow but not yet acknowledged. Figure 2.4 plots in gray-dotted curve the CDF of the number of bytes sent by the proxy that are acknowledged by non-duplicate TCP Acks. This plot is a weighted CDF where the contribution of each Ack is weighted by the number of bytes that it acknowledges. TCP Acks of 1428 bytes or less cover 50.7% of all acknowl-

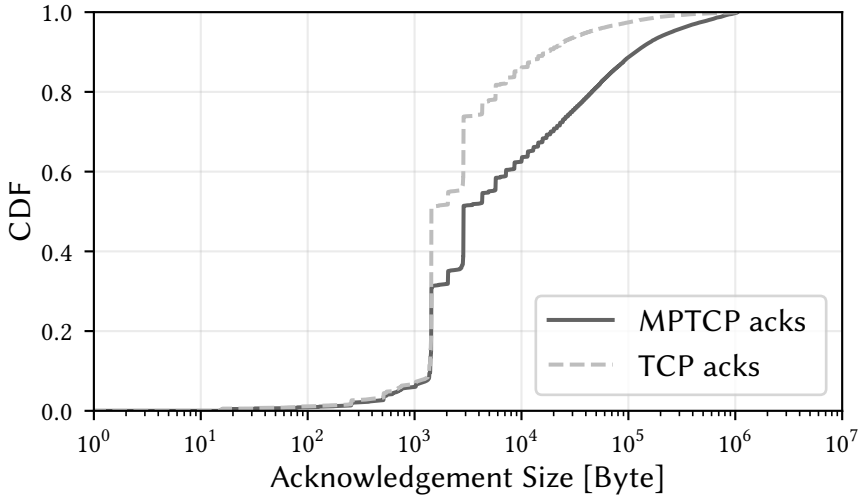


Figure 2.4: Size of the Multipath TCP and TCP Acks received by the proxy.

edged bytes and considering Acks of 20 KB or less the percentage is 91.1%.

The same analysis is now performed by looking at the DSS option that carries the Multipath TCP Data ACK s with `mptcptrace` [HB14]. The orange curve in Figure 2.4 shows the weighted cumulative distribution of the number of bytes acknowledged per Data ACK. Compared with the regular TCP Acks, the Multipath TCP Acks cover more bytes. Indeed, 51.4% of all bytes acknowledged by Multipath TCP are covered with Data ACKs of 2856 bytes or less, and this percentage increases to 70.8% considering Data ACKs of 20 KB or less.

The difference between the regular TCPAcks and the Data ACKs is caused by the reordering that occurs when data is sent over different subflows. Since the Data ACKs are cumulative they can only be updated once all the previous data have been received on all subflows. If subflows with very different RTTs are used, reordering will occur and data will possibly fill the receiver's window during a long period. This can also change the way applications read data which would be more by large bursts instead of frequent small reads.

2.2.3.4 Utilization of the Subflows

The next question is how data is spread among the subflows. Does Multipath TCP alternates packets between the different subflows or does it send them in bursts? Again, to be relevant, we consider the subtrace $\mathcal{T}_1$.

To quantify the spread of data, we introduce the notion of *subflow block*. Intuitively, a *subflow block* is a sequence of packets from a connection sent

over a given subflow without any packet transmitted over another subflow. Consider a connection where a host sends N data packets. Number them as $0, ..., N-1$ with 0 the first data packet sent and $N-1$ the last one. Let f_i denote the subflow on which packet i was sent. The n^{th} subflow block b_n is defined as

$$b_n = \{\max(b_{n-1}) + 1\} \cup \{i \mid i - 1 \in b_n \text{ and } f_i = f_{i-1}\}$$

with $b_0 = \{-1\}$ and $f_{-1} = \perp$. For example, if the proxy sends two data packets on s_0, then three on s_1, retransmits the second packet on s_0 and sends the last two packets on s_1, we will have $b_1 = \{0, 1\}$, $b_2 = \{2, 3, 4\}$, $b_3 = \{5\}$ and $b_4 = \{6, 7\}$. A connection balancing the traffic on several subflows will produce lot of small subflow blocks whereas a connection sending all its data over a single subflow will have only one subflow block containing all the connection's data. Figure 2.5 shows the number of subflow blocks that each connection contains. Each curve includes connections carrying their labeled amount of total bytes from proxy to smartphones. For most of the large connections, Multipath TCP balances well the packets over different subflows. In particular, 26.8% of connections carrying more than 1 MB have more than 100 subflow blocks. As expected, the shorter the connection is, more the subflow blocks tend to contain most of the connection packets. For short connections carrying less than 10 KBytes, 72.4% of them contain only one subflow block, and therefore they only use one subflow. This number raises concerns about unused subflows. If connections having at least two subflows are considered, over their 276,133 subflows, 41.2% of them are unused in both directions. It is worth noting that nearly all of these unused subflows are actually additional subflows, leading to 75.6% of the additional subflows being unused. This is clearly an overhead, since creating subflows that are not used consumes bytes and energy on smartphones [Pen+14] since the interface over which these subflows are established is kept active.

There are three reasons that explain these unused subflows. Firstly, a subflow can become active after all the data has been exchanged. This happens frequently since 62.9% of the connections carry less than 2000 bytes of data. In practice, for 21% of the unused additional subflows the proxy received their third Ack after that it had finished sending data. Secondly, as suggested previously, the difference in round-trip-times between the two available subflows can be so large that the subflow with the highest RTT is never selected by the packet scheduler. If the server does not transmit too much data, the congestion window on the lowest-RTT subflow remains open and the second subflow is not used. Nonetheless, 36.2% of the unused additional subflows have a better RTT for the newly-established subflow than the other available one. Yet, 59.9% of these subflows belong to connections carrying less than 1000 bytes (90.1% less than 10 KBytes). Thirdly, a subflow can be established

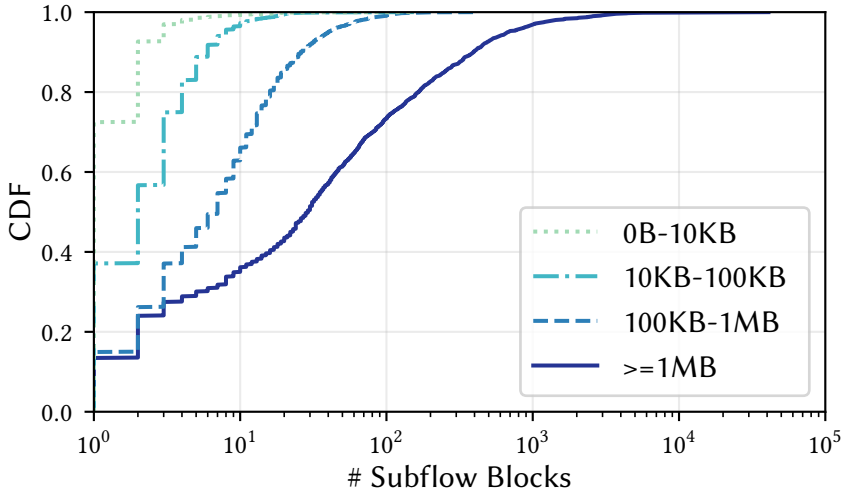


Figure 2.5: Size of the subflow blocks from proxy to smartphones on $\mathcal{T}_1$.

as a backup subflow [RFC6824]. Indeed, a user can set the cellular subflow as a backup one, e.g., for cost purposes. 2.1% of the unused additional subflows in $\mathcal{T}_1$ were backup subflows.

2.2.3.5 Reinjections and Retransmissions

In addition to unused subflows, another Multipath TCP specific overhead is the reinjections. A *reinjection* [Rai+12] is the transmission of the same data over two or more subflows. Since by definition, reinjections can only occur on connections that use at least two subflows, this analysis considers the subtrace $\mathcal{T}_2$. A reinjection can be detected by looking at the Multipath TCP Data Sequence Number (DSN). If a packet A with DSN x is sent first on subflow 1 and later another packet B with the same DSN x is sent on subflow 2, then B is a reinjection of A. We extended `mptcptrace` [HB14] to detect them. A reinjection can occur for several reasons: *(i)* handover, *(ii)* excessive losses over one subflow or *(iii)* the utilization of the Opportunistic Retransmission and Penalization (ORP) algorithm [PKB13; Rai+12]. This phenomenon has been shown to limit the performance of Multipath TCP in some wireless networks [Lim+14b]. Typically, Multipath TCP reinjections are closely coupled with regular TCP retransmissions. Figure 2.6 shows the CDF of the reinjections and retransmissions sent by our proxy. The quantity of retransmitted and reinjected bytes are normalized with the number of unique bytes sent by the proxy over each connection. 51.5% of the connections using at least two subflows experience retransmissions on one of their subflows whereas rein-

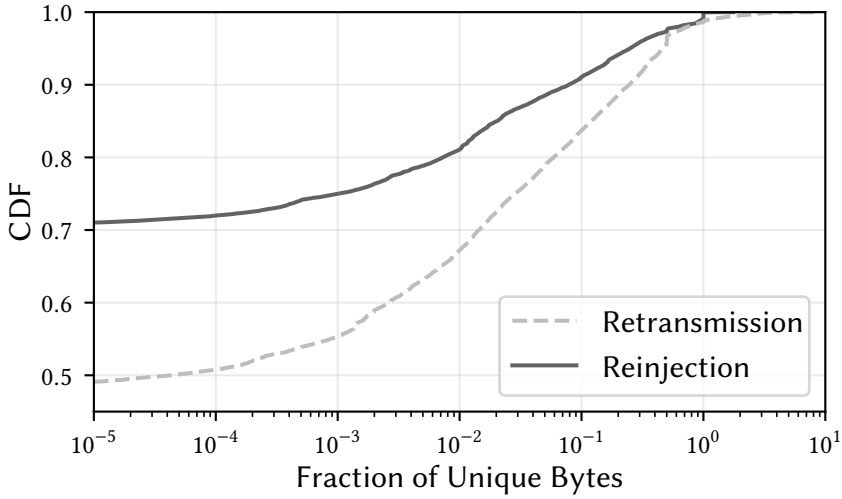


Figure 2.6: Fraction of bytes that are reinjected/retransmitted by the proxy on $\mathcal{T}_2$.

jections occur on 29.2% of them. This percentage of retransmissions tends to match previous analysis of TCP on smartphones [Fal+10; Hua+10]. 67.2% of $\mathcal{T}_2$ connections have less than 1% of their unique bytes retransmitted, and 83.7% less than 10%. 81.1% of the connections have less than 1% of their unique bytes reinjected, and 91% less than 10%. Observing more retransmissions than reinjections is expected since retransmissions can trigger reinjections. In the studied trace, the impact of reinjections remains limited since over more than 11.8 GBytes of unique data sent by proxy, there are only 86.8 MB of retransmissions and 65 MB of reinjections. On some small connections, we observe more retransmitted and reinjected bytes than the unique bytes. This is because all the data sent over the connection was retransmitted several times. In Figure 2.6 the half-thousand of connections having a fraction of retransmitted bytes over unique bytes greater or equal to 1 carried fewer than 10 KB of unique data, and 83.3% of them fewer than 1 KB. Concerning the reinjections, the few hundred of such connections carried less than 14 KB, 63.4% of them carried less than 1 KB and 76.1% of them less than 1428 bytes.

2.2.3.6 Handovers

One of the main benefits of Multipath TCP is that it supports seamless handovers which enable mobility scenarios [RFC6824; Paa+12]. A *handover* is here defined as a recovery of a failed subflow by another one. A naive solution is to rely on REMOVE ADDRS to detect handover. However, this TCP

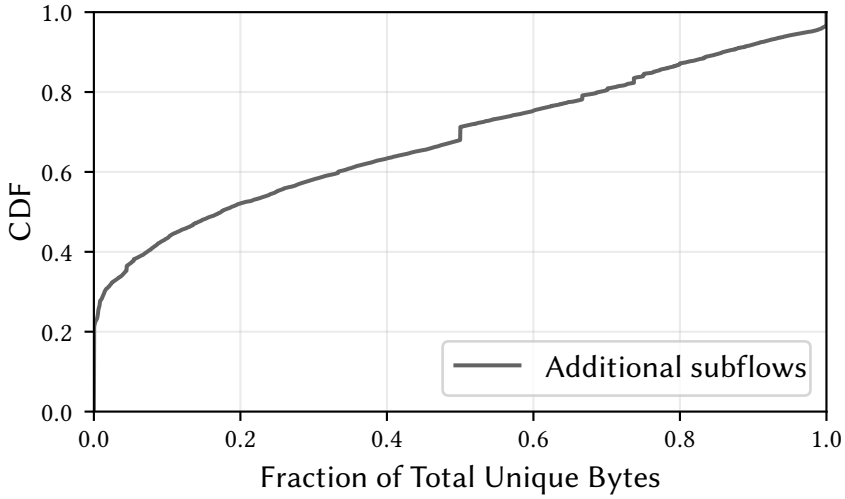


Figure 2.7: Fraction of total data bytes on non-initial subflows sent by the proxy on $\mathcal{T}_3$.

option is sent unreliably. Indeed, 22.1% of the connections experiencing handover do not exchange the Remove Addr option.

Instead, we propose an alternative methodology that relies on the TCP segments exchanged. Let LA_i be the time of the last (non-Rst) Ack sent by the smartphone seen on subflow i (that was used to send data) and LP_j the time of the last (non-retransmitted) segment containing data on subflow j. If $\exists\ k, l \mid k \neq l$, no Fin seen from the smartphone on subflow k, $LA_l > LA_k$ and $LP_l > LA_k$, then the connection experiences handover. Notice that only handovers on the subflows carrying data are detected. Among the connections that use at least two subflows, 25.7% experience handover. It has also the advantage to be implementation independent since it does not use the Add Addr or Remove Addr options that were not always supported by all implementations [Ear13].

Based on the subtrace $\mathcal{T}_3$, Figure 2.7 shows the fraction of unique bytes that were sent by our proxy on the additional subflows on connections experiencing handover. This illustrates the connections that could not be possible if regular TCP was used on these mobile devices. Indeed, a handover is typically related to the mobility of the user who can go out of the reachability of a network. Notice that this methodology can also detect handover in the smartphone to proxy flow. Indeed, 20.4% of connections experience handover with all data sent by the proxy on the initial subflow because the smartphone sent data on another subflow after having lost the initial one.

2.3 Observing the iOS Multipath TCP Implementation

Apple's iPhone devices represent the largest fraction of Multipath TCP capable hosts [BS16]. Initially, only the Siri application took advantage of the availability of multiple paths. Since the deployment of iOS11 in September 2017, any application can request the usage of Multipath TCP for the connections it initiates. To encourage the evaluation of Multipath TCP, we designed and implemented MultipathTester, an iOS application testing how Multipath TCP behaves under various network conditions. It provides two experimentation modes. The first one generates different traffic patterns ranging from bulk transfer to delay-sensitive request-responses and observes how Multipath TCP operates within stable network conditions. The second one requires the user to move until it triggers a network handover from Wi-Fi to cellular network and observes how Multipath TCP handles such changing environments.

The remaining of this section first presents the architecture of our measurement platform (§2.3.1). After introducing the operations of the studied *interactive* mode of the iOS Multipath TCP implementation (§2.3.2), we quickly characterize its performance under stable network conditions (§2.3.3). Then, we focus on how Multipath TCP handles network handovers (§2.3.4).

2.3.1 Design of MultipathTester

We now introduce MultipathTester, an iOS application aiming at evaluating the performance of Multipath TCP. Our framework enables users to observe how the iOS implementation behaves under our different test modes (§2.3.1.1). We then elaborate on our measurement infrastructure and implementation details (§2.3.1.2). We finally provide a few statistics about the usage of our application (§2.3.1.3).

2.3.1.1 Test Modes

MultipathTester relies on active measurements performed by voluntary users. For this purpose, at the first run and before perfoming any measurement, the application provides a consent form describing its research purpose to the user. Once agreed, the user can explicitly start active measurements. Two kinds of experiments are available. First, the stable network mode benchmarks the connectivity using different traffic patterns. Second, the mobile mode studies the impact of network handovers on Multipath TCP.

Stable Network Mode. During a stable network test, the application expects the user to stay at the same place so that the smartphone remains attached to the same network(s) during the entire test. In these stable condi-

tions, we benchmark the access point(s) to check if Multipath TCP behaves correctly in these networks. MultipathTester monitors the network connectivity during the test to detect user motion. It takes advantage of the `Reachability` API [Reach] to keep the list of available network interfaces from the smartphone's point of view. If during its run, the availability of one network interface changes (e.g., Wi-Fi being declared as lost by the device, cellular just getting Internet connectivity,...), the test is interrupted and classified as invalid.

To perform crowd-sourced measurements, the application needs to provide some service to the user. We present the stable mode as a network benchmark of the connected access point(s). On the initial screen, the user interface indicates, among others, the support of IPv4 and IPv6 for both Wi-Fi and cellular networks. When the tests run, MultipathTester shows real-time updated graphs about variables of interest such as the experienced latency or the achieved throughput. Thereafter, the user can consult details about the test in a dedicated results screen.

Different traffic patterns can be used with the stable mode. These network tests launch these traffic patterns sequentially, one transport protocol at a time. The order of the runs are randomized to avoid possible traffic correlation. MultipathTester provides a common interface to define traffic patterns, making it easy to add new ones. From an implementation viewpoint, each pattern implements two interfaces. The `Test` interface includes, inter alia, the `run` method called when launching the experiment and `getChartData` providing some graph visualization during the run. `TestResult` enables the results menu to fetch valuable information about the studied traffic to present them to the user. MultipathTester implements four simple traffic patterns: *ping*, *bulk*, *iperf* and *streaming*.

- **Ping.** This test simply sends a stream of five HTTP GET requests for a 10-byte file and computes the median delay. Its main purpose is to check the connectivity and to select the test server with which further experiments will be performed.

- **Bulk.** This test performs an HTTP GET request for a 10 MB file, and records the download time.

- **IPerf.** It generates a bulk transfer similar to the `iperf` tool [iPerf]. The client sends new data as fast as possible during a few seconds. We only use it to estimate the uplink bandwidth.

- **Streaming.** This traffic simulates a situation where the user interacts with a voice-activated application such as Siri while listening to an Internet radio. To achieve this, the traffic pattern follows a bidirectional

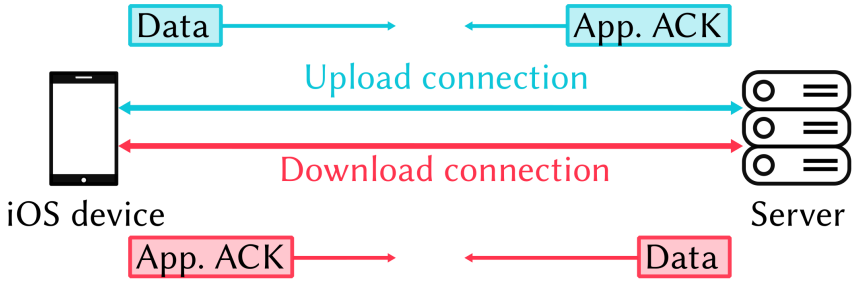


Figure 2.8: The streaming traffic pattern.

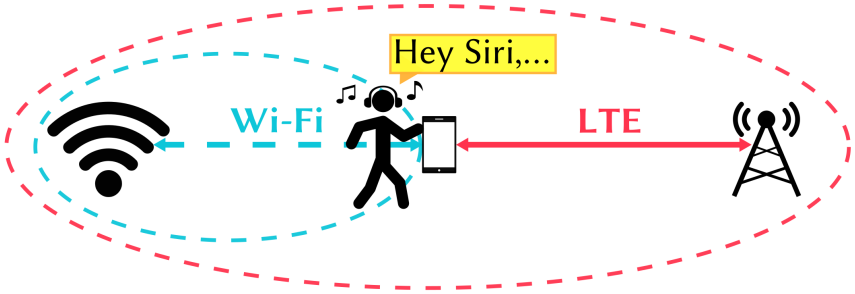


Figure 2.9: Mobility enables handover tests.

request-response fashion as shown in Figure 2.8. Every 100 ms, both the client and the server send a 2 KB request to the peer that replies with a 9-byte response. This short reply acts as a application-level acknowledgment that confirms the reception of each 2 KB chunk. The sending host then computes the delay between the request and the corresponding acknowledgment. With such low-volume exchanges, we do not expect any interference with the receive nor the congestion windows. Notice that we implement the traffic pattern with two independent Multipath TCP connections to prevent head-of-line blocking when a lost response blocks the delivery of the next request.

Mobile Mode. Our mobile tests focus on the situation illustrated in Figure 2.9. The smartphone is initially connected to both Wi-Fi and cellular networks while sending and receiving data simultaneously using the *streaming* traffic pattern. Then, the device moves away from the Wi-Fi access point. After some time, as the Wi-Fi connectivity starts fading, Multipath TCP switches to the cellular one. The mobile experiment evaluates how Multipath TCP han-

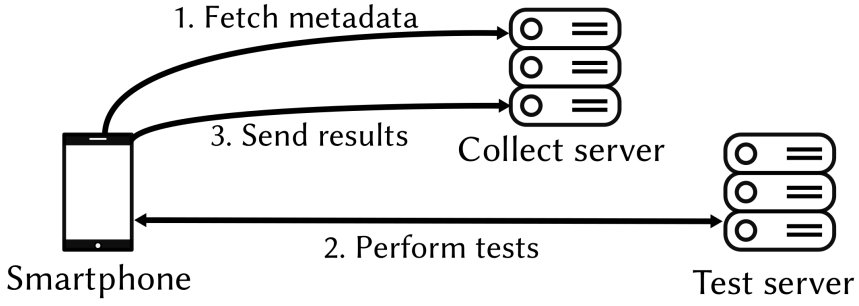


Figure 2.10: The infrastructure used by MultipathTester**.**

dles network handovers. This feature is one of the motivating cases for supporting multipath in the transport layer, as it can migrate connections from one network to another one without notifying the application. We expect the impact of the handover to be as low as possible on the applications, especially when they are latency-sensitive. The test completes when either the operating system tears down the Wi-Fi network, or the SSID of the Wi-Fi access changes.

To encourage users to perform this mobile test, we present it as a Wi-Fi reachability estimator where the smartphone computes the range of the Wi-Fi access point. The distance the device realized during a test is provided by the `Core Location` API [CoreL]. This framework takes advantage of the GPS when the user gives its consent, otherwise it estimates the localization of the device with all the remaining components such as Wi-Fi, Bluetooth, magnetometer, barometer and cellular. MultipathTester computes two reachability distances. The most naive one is the distance covered between the start of the test and the instant when the operating system changes the connectivity state of the Wi-Fi interface. However, when moving away from an access point, it is likely that the detection of the Wi-Fi loss by the system is not immediate. During a few seconds, the device might stay connected to a network losing all transmitted packets. To estimate the range of the Wi-Fi network, MultipathTester computes a "last data received on Wi-Fi" distance. At each location update by the `Core Location` API, the application checks whether some bytes have been received on the Wi-Fi. If it is the case, MultipathTester updates this second distance. As the *streaming* traffic pattern involves two data streams generating packets every 100 ms, we expect to regularly receive data. When the test completes, MultipathTester presents both computed distances to the user.

2.3.1.2 Measurement Infrastructure

Our measurement infrastructure involves three different nodes as shown in Figure 2.10. On one side, there is the smartphone running MultipathTester. On the other side, we use two different servers. The test server is contacted by the smartphone to perform the experiments. We deploy three test servers located on different continents: Europe (France), North America (Canada) and Asia (Japan). The collect server gathers the measurement results.

Each user-triggered experience is carried out as follows. First, the smartphone contacts the collect server to fetch metadata, such as the URL of available test servers and the list of experiments to launch. Right after, the smartphone initiates a *ping* traffic pattern to each of the available test servers to estimate the closest test server. Notice that these pings are synchronized, i.e., for each of the five runs, it waits for the reply of the previous ping by all the test servers before launching the next ping. Then, once the user requests it, MultipathTester interacts with the closest test server to perform experiments. At the end of each test runs, the smartphone sends the test results to the collect server. These outcomes include primary traffic-specific metrics (delays for *streaming*, download completion time and fetched file for *bulk*,...), device and network information (name and type of the network accesses, version of the application,...) and dumps of the transport protocol states. These dumps are periodically collected. Multipath TCP logging data comes from both the `TCP_INFO` and the `ioctl SIOCGCONNINFO` interfaces. To access them, we redefine Darwin kernel structures which have not been included in the header files provided by iOS, such as `struct tcp_info` and `struct mptcp_itf_stats`.

The test servers use the Multipath TCP implementation in the Linux kernel 4.14 (v0.94) [MPTCPlk] with the default multipath-specific algorithms (`default` lowest-RTT packet scheduler, `full-mesh` path manager and the OLIA congestion control scheme [Kha+13]). The smartphones use the native implementation of (Multipath) TCP provided by the Darwin kernel [Darwin].

Overall, MultipathTester contains ~9250 lines of code (without comments), whose ~7000 are Swift, 1000 are Objective-C and ~1250 are Go code.

2.3.1.3 Usage Statistics

Since the first public release of MultipathTester on March 8th, 2018 until April 30th, 2019, we collected 1098 test runs coming from 264 unique devices. 43% of the runs are mobile tests. The distribution of the test loads between Europe, Asia and America is 65%, 17% and 18%, respectively. MultipathTester has been used in 72 different mobile carriers and 288 different Wi-Fi SSIDs.

2.3.2 The Interactive iOS Multipath TCP Mode

On iOS, applications can explicitly request the usage of Multipath TCP using the iOS API [MPiOS]. Apple provides three modes of operation for Multipath TCP, each with different objectives: *handover*, *aggregate* and *interactive*. The *handover* mode aims to provide seamless handover from Wi-Fi to cellular networks for long-lived or persistent connections. The *aggregate* mode uses all network connectivities to increase the throughput of the connection. The *interactive* mode attempts to use the lowest-latency connectivity and is advised for latency-sensitive, low-volume connections. Nonetheless, the ability to use the LTE network while the Wi-Fi might still be available raises concerns about cellular data consumption. Users typically expect the device to use the Wi-Fi network when available and usable, even if it sometimes provides lower throughput and/or larger latency than the cellular one. This is why Apple restricts the *aggregate* mode to developer phones only. Since we want our application to be as accessible as possible, we do not explore the *aggregate* mode.

MultipathTester uses the *interactive* Multipath TCP mode. We focus on this mode rather than the *handover* one as it is advised for latency-sensitive applications, which matches our *streaming* traffic pattern. To understand its behavior, we analyzed its source code [Darwin] and inferred its operations. The *interactive* mode prioritizes the Wi-Fi network over the cellular one, marking the latter one as a backup subflow. As the server runs its own packet scheduler (in our case, the lowest-latency one), the use of the backup subflow ensures that the peer will not use the cellular unless it encounters connectivity issues with the Wi-Fi, such as retransmission timeouts (RTOs). The iOS packet scheduler sends data only on the Wi-Fi subflow, unless one of the following conditions holds.

1. The smoothed RTT of the Wi-Fi subflow is above a threshold initially set to 600 ms, while the cellular path is not over this threshold.

2. The Wi-Fi path is experiencing RTO — i.e., the timer has fired and no acknowledgment was received since that event — and the phone wants to push new data.

3. The Wi-Fi RTO value is over a threshold initially set to 1500 ms, while the RTO of the cellular path is lower.

The packet scheduler code checks those previous conditions in the presented order. Notice that the threshold values of the RTT and the RTO can be decreased by the Apple's WifiAssist application when it considers the Wi-Fi network as "bad". However, this system is closed-source, making it difficult to understand its behavior.

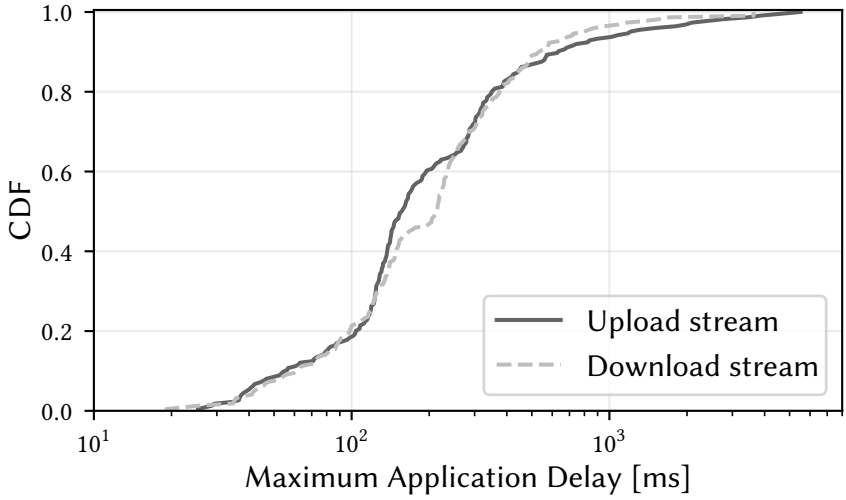


Figure 2.11: Multipath TCP maximum observed application delays under stable conditions with the streaming traffic pattern.

2.3.3 Stable Network Runs

Given that the studied *interactive* mode of iOS Multipath TCP prioritizes the Wi-Fi network and uses the cellular only if it starts experiencing connectivity issues, we do not expect Multipath TCP to behave differently from plain TCP under stable conditions. Yet, these experiments allow us to get an idea of the diversity of the networks where MultipathTester ran. In particular, our *bulk* traffic pattern shows that downloading a 10 MB already provides very diverse results. The 25$^{\text{th}}$ and 75$^{\text{th}}$ percentiles of the transfer completion time are 2.73 s and 9.69 s. The fastest run completed in 0.63 s (average throughput of 127 Mbps) while the slowest one took 1232 s (mean rate of 65 Kbps). We also observe such heterogeneous throughput distribution in the uplink direction with our *iperf* traffic. The tests experienced 25$^{\text{th}}$ and 75$^{\text{th}}$ percentiles of upload throughput of 2.94 Mbps and 16.95 Mbps, respectively. Again, we also observe quite extreme values, such as 10 Kbps or 257 Mbps.

Besides the throughput, MultipathTester also observes instabilities in some networks. Figure 2.11 shows that with the *streaming* traffic pattern, most of the experiments observed maximum application delays of a few hundreds of milliseconds. However, despite the low bandwidth usage and the absence of network changes, 6% of these static *streaming* runs experienced delays over 1 s. In particular, we focus our attention on the extreme run which experienced maximum application delays of 5.5 s and 1.6 s in the upload and download streams, respectively. The phone started the connection over the

cellular interface, but could not establishh the Wi-Fi subflow, i.e., the Wi-Fi network was reachable but not connected to the Internet. However, the cellular network seemed to be bufferbloated. The smoothed RTT estimated by the smartphone for the upload connection gradually increased from 100 ms at the connection establishment to about 4.5 s at the end of the transfer, while the smartphone did not retransmit any data. This shows the high heterogeneity of networks that MultipathTester faced, even without considering mobile situations.

2.3.4 Mobile Experiments

We now focus on our findings from our experiments where the user moves. Our mobile tests only generates the *streaming* traffic pattern. This section is split into two parts. We first report some indicative results about the Wi-Fi reachability distance estimations. We then concentrate on the performance of the *interactive* mode of iOS Multipath TCP and evaluate whether it achieves its low-latency goal.

2.3.4.1 Wi-Fi Reachability Distance

As suggested previously, there might be some delay, and therefore some distance between the point where the Wi-Fi becomes unusable and the system effectively detects it. For this, we consider 235 experiments performed between March 8th, 2018 and April 30th, 2019 where the Wi-Fi network was lost due to the user motion.

Figure 2.12 confirms our intuition that the time a Wi-Fi network is kept while being unusable is not negligible. Only 5% of the experiments observed less than 1 s between the last received packet on the Wi-Fi and the system withdrawing it. Actually, in most of these experiments, the smartphone switched from one access point to another one — different BSSIDs — within a same network — same SSID. Rather, 78% of the runs observed delays over 10 s, and 12% saw reactions lasting more than 30 s. These results illustrate the interest of using Multipath TCP in such scenarios. If the smartphone was using plain TCP, the *streaming* traffic would have been trapped on the Wi-Fi network, waiting for a while before getting notified of the loss of the connectivity and trying to establish a new connection over the cellular network.

Since the user is moving away of the Wi-Fi access point, we observe different estimated Wi-Fi reachability distances taking either last received packet or system detection criterias. Figure 2.13 shows that in the median run, the Wi-Fi is usable for 23.5 m, while the system leaves the network after 41 m. However, these values are very dependent on both the environment and the user. Indeed, the Wi-Fi reachability distance would be very dependent on the environment in which the experiment is performed (e.g., walls, trees,…). In

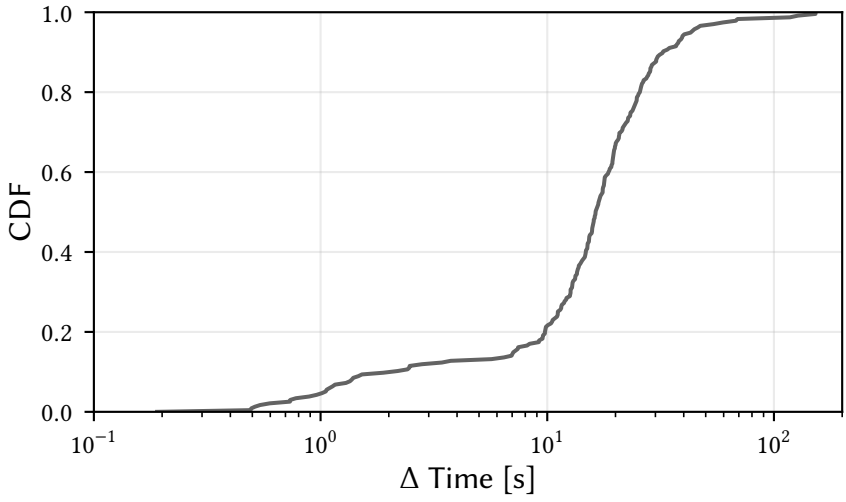


Figure 2.12: The time delta between the last received packet over the Wi-Fi and the tear down by the iOS system.

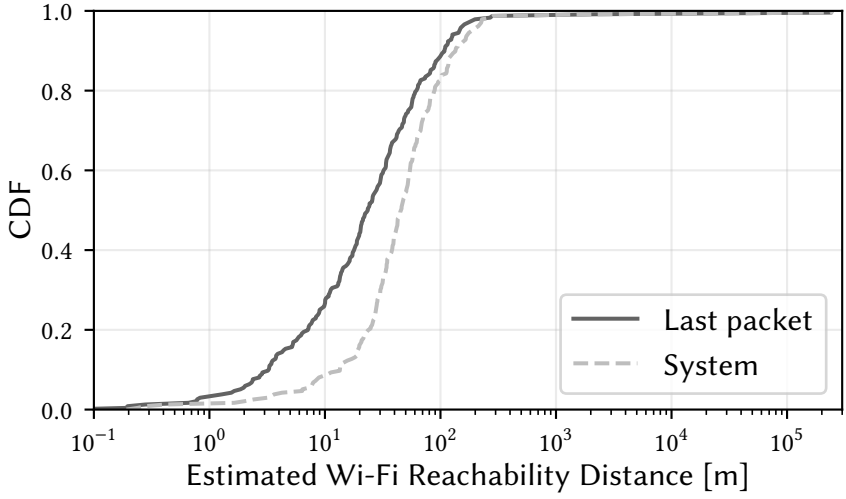


Figure 2.13: Estimated Wi-Fi reachability distance depending on the criteria used, either last received packet or system tear down.

Reason	RTT Threshold	Under RTO	RTO Threshold	Other
Test (%)	2.2%	67.0%	11.0%	19.8%

Table 2.4: Multipath TCP reason to start using the cellular.

addition, while MultipathTester advises the user to start the test close to its Wi-Fi access point, nothing enforces it to do so. Intriguingly, we found a run where the estimated Wi-Fi reachability distance was more than 241 km. The reason behind this outlier comes from an implicit hypothesis we made about the Wi-Fi access point. We assume it remains static during the test, allowing considering the distance traveled is due to the user motion. However, that particular run was connected to a high-speed train Wi-Fi access point. As both the network and the user were moving together, this value actually shows the distance covered by the train while running the experiment.

2.3.4.2 Multipath TCP and its Interactive Mode

We now consider here a dataset of 231 experiments performed between April 23rd, 2018 and April 30th, 2019. Our mobile dataset includes 44 distinct cellular networks and 84 different Wi-Fi ones.

Multipath TCP often waits for Wi-Fi RTO before switching to the cellular network. The Multipath TCP interactive mode uses the algorithm described in §2.3.2 to decide when the smartphone should start using the cellular backup path. Thanks to the periodic collection of the internal state of Multipath TCP, we can infer which condition triggered the usage of the cellular path by the smartphone. Table 2.4 shows that two-third of the tests started to use the cellular because new data arrived while the Wi-Fi subflow was experiencing a RTO. This might be related to our *streaming* traffic pattern that generates data every 100 ms. In comparison, handovers caused by high smoothed RTTs are rare. This might be linked with the high initial threshold of 600 ms. Notice that the cause for 20% of the cellular switches cannot be determined using the three first conditions. We suspect that WifiAssist declared the Wi-Fi network as "bad" and decreased the RTT and RTO thresholds. However, these thresholds are not exposed by the Darwin kernel, making it impossible to confirm this hypothesis.

A Multipath TCP handover is not an abrupt process. As the smartphone moves away from the Wi-Fi access point, its performance will eventually decrease, leading to a network handover to the cellular connectivity. Nevertheless, this switch is not necessarily instantaneous. Consider the situation shown in Figure 2.14. The connection starts over the Wi-Fi network.

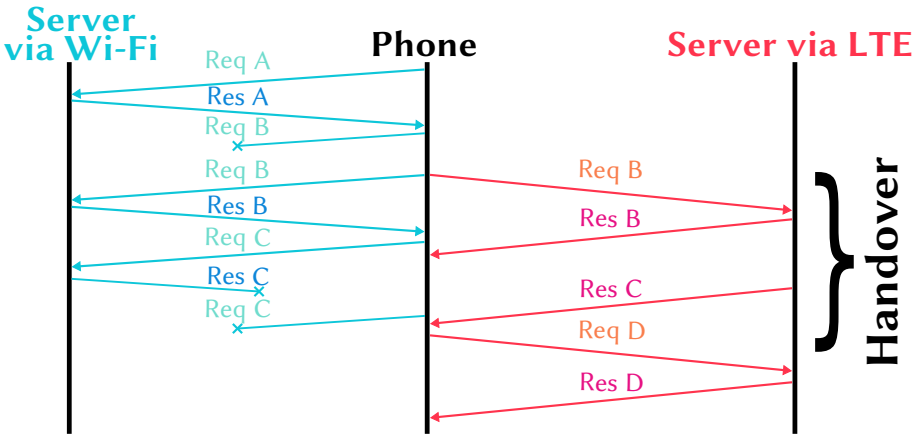


Figure 2.14: Example of a possible network handover.

After some time, it experiences retransmissions due to weaker signal. The phone then decides to use the cellular path to retransmit the lost request as it experienced an RTO on the Wi-Fi. Because a Multipath TCP subflow still remains a TCP connection, the smartphone still retransmits the packet over the Wi-Fi path too. These retransmissions might eventually succeed, leading to the reuse of the Wi-Fi path. Therefore, there is a time interval during which both Wi-Fi and cellular networks are functional and used by the connection. We call this transient state the *handover duration*.

The ability of using both networks concurrently enables the smooth handover that users expect. To quantify its duration, we measure the delay between the first data packet sent on the cellular network and the last activity observed on the Wi-Fi one. The end of the Wi-Fi liveliness can be measured as either the transmission time of the last packet (data or TCP acknowledgment) sent by the phone or the reception time of the last packet received. Figure 2.15 shows that when smartphones move, they simultaneously use the Wi-Fi and cellular networks at some point. Indeed, only 10% of the test runs experienced an abrupt switch from the Wi-Fi to the cellular network, i.e., the Wi-Fi stopped working before the smartphone started using the cellular path. This corresponds to the negative values in Figure 2.15. On the other hand, 58% of the experiments observed a handover duration of at least 10 seconds. This illustrates that in mobile scenarios, the network handover is not an abrupt process.

Multipath TCP network handovers can affect the application. The main objective of the *interactive* mode of iOS Multipath TCP is to enable the application to automatically use the lowest-lantecy interface while keeping the usage of the cellular one as low as possible. To this end, we focus on

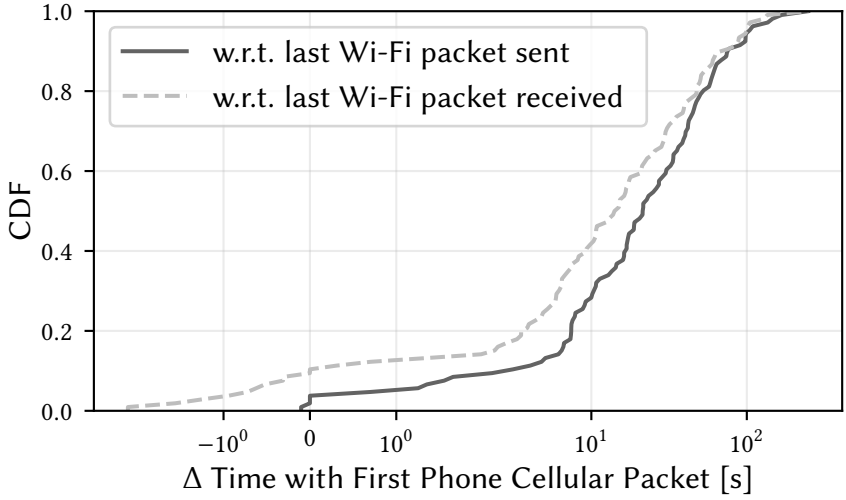


Figure 2.15: Duration of the Wi-Fi to cellular handover.

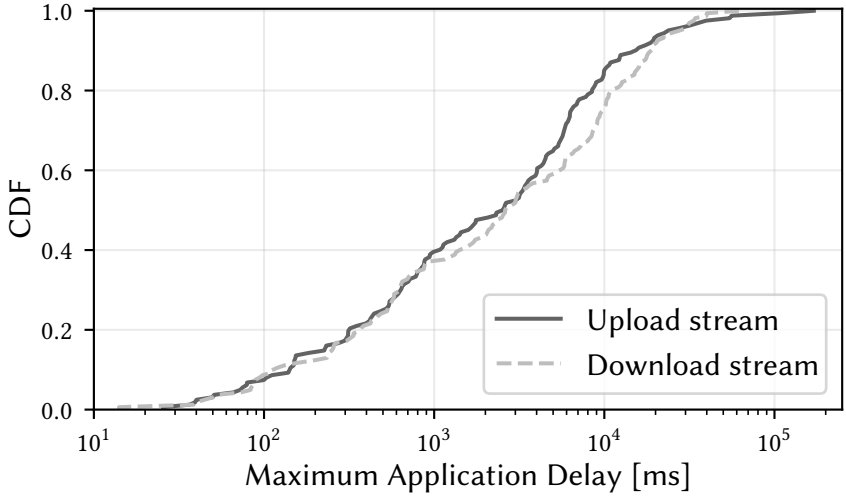


Figure 2.16: Multipath TCP maximum observed application delays under mobile conditions.

the maximum delay observed by the sending host for both upload and download streams of the *streaming* traffic pattern, i.e., two maximum delays are collected per test. Despite the *interactive* mode, Figure 2.16 shows that more than 60% of the test cases have a maximum application delay that is longer than a second for both data streams. To exemplify our results, imagine that the download stream represents an Internet radio while the upload one simulates a voice-activated application such as Siri. With the median value at 2.6 s, this means that the user would have experienced a music stall in 50% of the experiments if the playback buffer was not at least 2.5-second long. Similarly, with a 70^{th} percentile of 6 s, the Siri application would have been unresponsive for at least 6 s, possibly raising a connectivity error to the user.

These long application delays have several causes. One of them is that the detection of the working interfaces by both peers is not synchronous. Typically, when the smartphone is at the border of its Wi-Fi reachability, the network starts losing packets and exhibiting latency variability, therefore increasing the RTO value. Still, the network might still be considered as usable by the hosts. In particular, the server always chooses first the Wi-Fi path when usable, and relies on the backup cellular one otherwise. When the smartphone is out of Wi-Fi reachability, the server needs to figure out that the path is not usable anymore. In practice, this notification comes when an RTO occurs. However, the timer might have reached values over one second. Furthermore, if a single valid Multipath TCP packet reaches the server from the — potentially failed — Wi-Fi path, this subflow will be considered usable again and hence preferred over the cellular backup one [Pin15]. The server will be stuck over the Wi-Fi path until a new RTO enables it to use the cellular one. This situation may last for a while, as suggested by Figure 2.15. This delay between the loss of the Wi-Fi network by the smartphone and its detection by the server often explains the high observed application latencies. While the server needs to rely on retransmissions, the phone has a better view of the network, as it is easier for it to detect a weak Wi-Fi signal than the server. From our results, we observe that upload connections tend to have lower application delays than download ones, especially between the 50^{th} and 90^{th} percentiles. As in the upload connection, the phone is sending data, it can adapt its packet scheduling thanks to local information.

Another cause of large application latencies is because despite having network connectivity on the cellular interface, the smartphone does not establish any Multipath TCP subflow over it. It is possible that iOS keeps track of the networks exhibiting middlebox interference [Paa16] such that it does not retry using Multipath TCP for a while. Therefore, the experienced application delay is the time between the transmission of the first unanswered request and the notification that the connection has been torn down.

We also observe a tail reaching hundreds of seconds. These experiences

suffer from the lack of support of the Multipath TCP ADD ADDR option [RFC6824] by the iOS implementation. This is probably due to privacy reasons [Ear13], as the ADD ADDR option could potentially disclose in clear-text all the addresses of the smartphone. Our test servers have both IPv4 and IPv6 addresses. With the happy eyeballs process [RFC6555], if the connection starts on the Wi-Fi using IPv6 while the cellular is IPv4-only, the transfer remains stuck on the Wi-Fi. Note that this issue does not occur in the converse situation — IPv4 Wi-Fi with IPv6-only cellular — as the iPhone includes a NAT64 daemon.

2.4 Conclusion

While many researchers conducted measurements with bulk transfers, the actual benefits of Multipath TCP in the smartphone use case were unclear. This chapter addressed this gap by performing two measurement studies.

The first one passively collected network trace generated by actual Android applications running on smartphones using the Linux implementation of Multipath TCP. On one hand, we confirmed previous findings about the smartphone environment, such as download-driven network traffic and path heterogeneity. On the other hand, we showed that when they are created, additional paths are often unused. These form the main overhead of Multipath TCP on smartphones. Yet, we identified connections that benefited from the Multipath TCP seamless network handover.

The second one actively spawned different traffic patterns to evaluate the behavior of the iOS Multipath TCP implementation. Our stable tests emphasized the diversity of network performance in the current Internet. The mobile runs pointed out the time required to assess the failure of a wireless network. Although Multipath TCP enables the connection to switch from the Wi-Fi connectivity to the cellular one, the application can suffer from this potentially long process.

Both our measurement campaigns showed that Multipath TCP works in the smartphone use case, yet there remains room for improvements. We will address most of the issues raised here in the Chapter 3.

Tuning Multipath TCP for Interactive Applications

3

The previous Chapter demonstrated that smartphones can take advantage of multiple wireless networks without modifying applications thanks to Multipath TCP. On smartphones, user experience is always a compromise between network performance and energy consumption. Devices especially benefit from the network handover ability of Multipath TCP. However, its reference implementation in the Linux kernel is mainly bandwidth aggregation driven and often wakes up the cellular interface by creating a path without sending data on it. Furthermore, for most smartphone users, the Wi-Fi and cellular networks are not equivalent. Wi-Fi has two major advantages compared to cellular networks. First, using Wi-Fi consumes less energy [Hua+12; Nik+15]. Second, most service providers charge for cellular data while most Wi-Fi networks are free or charged on a flat-rate basis. For these reasons, many smartphone users only use their cellular interface for voice calls and when there is no Wi-Fi network available [CSP17].

In this Chapter, we first review the related works to motivate the need to tune Multipath TCP for the smartphone use case (§3.1). Next, we look at the Apple's Siri traffic and characterize the behavior of an interactive application (§3.2). We then propose MultiMob, a series of improvements that adapt Multipath TCP to the requirements of today's smartphone applications (§3.3). More precisely, MultiMob provides the following compromise between the experienced application latency and the utilization of the cellular network.

- **A MultiMob server replies on the last network used by the smartphone (§3.3.1).** If a smartphone sends a request over a cellular subflow because its Wi-Fi performs badly, the server should reply over the same subflow since the server has no information about the client's wireless conditions.

- MultiMob **minimizes cellular usage and unused subflows (§3.3.2).** Like iOS [CSP17], MultiMob prefers to use the Wi-Fi interface over the cellular one. MultiMob replaces the *make-before-break* strategy of the Multipath TCP implementation in the Linux kernel by *break-before-make*. With this strategy, the cellular network is only used after a failure of the Wi-Fi one.

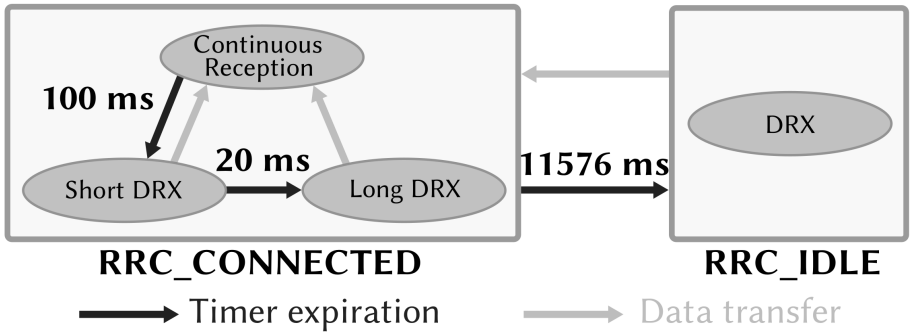


Figure 3.1: The RRC state transitions in LTE network [Hua+12].

- MultiMob **limits handover delays (§3.3.3).** The *break-before-make* strategy minimizes energy consumption at the expense of increased handover delays. MultiMob reduces those delays by extending the Multipath TCP protocol to carry data during the subflow handshake.

We evaluate MultiMob in both emulated Mininet environment (§3.4) and real wireless networks using Android devices (§3.5). We finally qualify the applicability of MultiMob (§3.6) and conclude this Chapter (§3.7).

3.1 Motivations and Related Works

Before tuning Multipath TCP, it is important to understand how smartphones interact with the wireless networks. We first provide some background about the cellular power characteristics (§3.1.1). We then describe our arguments calling for an adaptation of Multipath TCP for the smartphone use case (§3.1.2).

3.1.1 LTE State Machine

The Long Term Evolution (LTE), also known as 4G, is the most advanced deployed cellular technology. It has three main design goals: high throughput, low latency and limited power consumption. To achieve this last objective, the LTE has different operation modes. We describe here the basic knowledge needed for the remaining of this Chapter. Here, we consider the model proposed by Huang et al. [Hua+12] and refer the interested reader to that paper for further detail.

The Radio Resource Control (RRC) defines the power state of the cellular interface. Figure 3.1 illustrates the two RRC states of LTE: RRC_CONNECTED and RRC_IDLE. When the cellular interface is on but unused for a while, the

	Power (mW)	Duration (ms)
Screen off (base)	11.4	N/A
Screen 100% on	847.2	N/A
LTE promotion	1210.7	260.1
LTE RRC_CONNECTED *Short DRX*	1680.2	1 (period: 20)
LTE RRC_CONNECTED *Long DRX*	1680.1	1 (period: 40)
LTE RRC_CONNECTED tail base	1060	11576
LTE RRC_IDLE *DRX*	594.3	43.2 (period: 1280)
Wi-Fi promotion	124.4	79.1
Wi-Fi tail base	119.3	238.1
Wi-Fi beacon (idle)	77.2	7.6 (period: 308.2)

Table 3.1: LTE and Wi-Fi power model [Hua+12].

LTE is in the RRC_IDLE mode. The antenna is then in Discontinuous Reception (*DRX*). This mechanism enables the interface to periodically wake up for short instants — 43 ms in RRC_IDLE — while staying asleep for longer periods of time — 1.28 s in this case. This limits the power consumption compared to a continuously turned on antenna. Once the cellular interface needs to send or receive a packet, the antenna promotes from the RRC_IDLE state to the RRC_CONNECTED one. This promotion typically lasts for a few hundreds of milliseconds — 260 ms in our model — before the interface becomes ready for transmission. Therefore, if the smartphone wants to send a packet while being initially in RRC_IDLE state, there will be a delay between the request to send and the actual packet transmission by the cellular antenna. Upon promotion completion, the LTE interface is in *Continuous Reception* mode where packets can be sent and received. The antenna stays in this mode until no data has been sent or received for 100 ms. The LTE then enters in Discontinuous Reception. Two *DRX* sub-modes are present in the RRC_CONNECTED state. Both monitor the LTE network for a short time — 1 ms — and then put the antenna in a lower power mode for a longer period — 20 ms for *Short DRX*, 40 ms for *Long DRX*. On one hand, if data needs to be transmitted or received while being in one of the *DRX* modes, the antenna directly goes to the *Continuous Reception* mode. On the other hand, if the antenna observes no network activity in RRC_CONNECTED for 11576 ms, it switches to the RRC_IDLE state.

Each LTE mode has different power consumption as shown in Table 3.1. The cellular interface consumes more energy when interacting with the network than when idle. Yet, the RRC_CONNECTED tail base, i.e., the "sleep" state of *DRX*, has a non-negligible energy impact. It consumes more than a smartphone screen with maximum luminosity. Also, an idle device stays for more than 11 seconds in the RRC_CONNECTED state. For the transmission of

a single packet, starting from the idle mode, the energy consumption from the interface promotion to the end of the tail for LTE and Wi-Fi are respectively 12.8 J (for 11.836 s) and 0.04 J (during 0.317 s). The energy consumed per bit ratio can be very high for LTE, especially when only a few packets are sent. The attentive reader will notice that Table 3.1 does not characterize the power consumption of the Continuous Reception mode. It actually depends on several factors, such as the throughput of the network and the fraction of uplink traffic. As a lower bound, we assume that the *Continuous Reception* mode has the same power consumption as the one of the *On* mode of the RRC_CONNECTED *Short DRX*, i.e., 1680 mW.

3.1.2 Multipath TCP on Smartphones

We outline in this section some lessons we learned based on discussions with network operators, measurements with friendly users and previous works.

Smartphone applications rarely perform bulk transfers. Multipath TCP was designed to aggregate bandwidth and many articles evaluate whether it reaches that objective [Rai+11a; Che+13; PKB13; Paa+14; Den+14]. However, both previous works [Fal+10] and our measurements (§2.2) show that smartphones rarely exchange very large files. Most of the connections carry a few KB. We also notice that many connections experience large idle times. To illustrate this, we performed passive measurements by following the methodology described in Section 2.2.1, except that we collect the network traffic on the client to get the smartphone's viewpoint. Figure 3.2 shows the maximum idle time observed by each of the 183 383 collected Multipath TCP connections carrying data. Most of them experience idle times of less than one second. Nonetheless, 15.8% of the connections face more than 10 seconds of idle time between two consecutive data packets. From TCP's point of view, an idle connection is not a problem. Yet, from a battery consumption viewpoint, an idle connection can consume energy if the radio needs to remain active to support it.

Many subflows do not carry data. The `full-mesh` path manager immediately creates subflows on all active interfaces. Our measurements in Section 2.2.3 show that 76% (resp. 86%) of the first additional subflows are established within the first 100 ms (resp. 1 s) of the connection. Unfortunately, most of these subflows are useless. Around 75% of the Multipath TCP connections do not send *any* data over the additional subflows, i.e., all data is exchanged over the initial path. With the `default` packet scheduler, if the first subflow exhibits a lower round-trip-time than the additional ones, Multipath TCP will only use the initial one. Previous works also indicate that

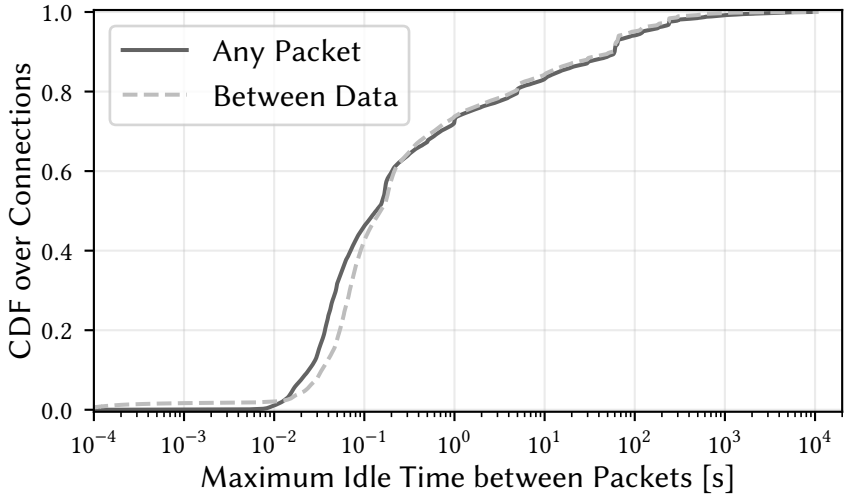


Figure 3.2: On smartphones, it is frequent to observe connections experiencing idle times of tens of seconds.

Multipath TCP can even perform worse than TCP with short flows in heterogeneous networks [Han+15; Nik+16].

Mismatch with user expectations. As previously emphasized, most users favor Wi-Fi over cellular for both monetary and power consumption reasons [Hua+12; Nik+15; CSP17]. They expect that their smartphone will use Wi-Fi whenever it works well and will switch to cellular only if it brings some benefits. However, the packet scheduling decision is taken by the sender of the packet. In practice, our measurements (§2.2.2) and previous works [Fal+10] indicate that smartphones mainly receive data, meaning that most of the scheduling decisions are taken by remote servers. The `default` packet scheduler considers the path with the lowest estimated round-trip-time as the best one to send the next data. With the decreasing latency of the LTE connectivity and network operators providing Wi-Fi access, the cellular path can sometimes exhibit a lower latency than the Wi-Fi one [SB12]. The server scheduling decisions can therefore go against the user expectations.

Backup subflows consume too much energy. According to the Multipath TCP specification [RFC6824], one way to minimize the utilization of the LTE connectivity is to always establish the cellular path as a backup subflow [Lim+14a]. While being useful in mobility scenarios, there is no point in creating backup subflows if the primary one does not face any connec-

tivity issue. Indeed, energy consumption is a major concern for mobile devices [BBV09; CH10; CSP17], and many works explore various solutions to limit the battery usage on smartphones [Ra+10; Che+15]. However, as suggested by Section 3.1.1 and previous works [Den+14], opening a subflow on the cellular interface without using it is expensive from an energy consumption viewpoint, the Wi-Fi interface consuming at least five times less than the LTE one [Nik+15]. In the remaining of this Chapter, we consider the LTE model proposed by Huang et al. [Hua+12] described in Section 3.1.1 to estimate the cellular power consumption. We expect similar results with other models [Nik+15]. Based on our model, we identify user-initiated connections in Multipath TCP Backup mode that create cellular subflows without sending data on them. In our passive measurements, we observe 12 988 connections that created 628 RRC_CONNECTED periods on cellular and consumed more than 4670 J just because of the SYN sent. Our LTE model [Hua+12] indicates that for a smartphone, opening a single cellular subflow is equivalent, from a energy consumption viewpoint, to lighting up the screen at 100% during the RRC_CONNECTED period, which lasts more than 11 s. Preventively opening the cellular subflow as previously proposed [Paa+12] is therefore very expensive from the battery perspective. Despite this, both Apple's Siri and the *interactive* mode of iOS Multipath TCP still create cellular subflows at the beginning of the connections.

Related works. Pluntke et al. [PEK11] is the first work addressing the energy concern with Multipath TCP by proposing a packet scheduler aiming to minimize the power consumption. It does not take into account mobile situations. Peng et al. [Pen+14] introduce various algorithms to optimize both energy consumption and throughput for file transfer and video streaming. However, they only rely on simulations. Lim et al. [Lim+14a] propose eMPTCP, an energy-efficient Multipath TCP version for mobile devices. In practice, it proposes a path manager delaying the establishment of the cellular subflow until a given threshold of transferred bytes is crossed. In addition, it also ships a packet scheduler that selects the cellular path only if the Wi-Fi throughput is not sufficient. While working with bulk transfers, interactive applications can transmit very few bytes and do not require large bandwidth. Furthermore, eMPTCP does not consider device mobility. Sinky et al. [SHG16] propose to rely on the signal strength of the Wi-Fi network to tune the congestion window to trigger seamless Wi-Fi handover with bulk transfer. However, it was only tested under NS3-DCE environment and not with actual devices. Han et al. [Han+16] propose a packet scheduler that stops using the cellular network when the Wi-Fi is sufficient to support delay-tolerant traffic. However, interactive applications are typically delay-sensitive and the solution does not discuss the unused cellular subflows. Frömmgen et al. [Fro+16] suggest

a packet scheduler duplicating data packets on all available network paths. While it ensures low-latency delivery, such strategy does neither take energy consumption into account nor consider user policy who prefers Wi-Fi over cellular networks.

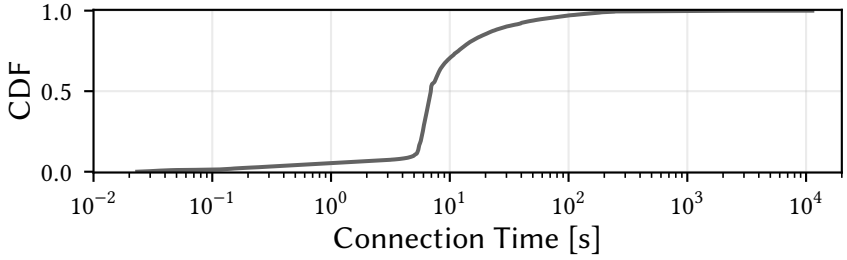
3.2 Mimicking Apple's Siri Traffic

Voice-activated applications will likely play a growing role on smartphones in coming years [Hem16; Kni16]. One of them is Apple's Siri assistant. Despite being the first application using Multipath TCP at large scale [BS16], little is known about the Siri traffic. Assefi et al. [Ass+16] analyze the reaction of Siri to losses and jitter but do not provide any model of the traffic produced by this application.

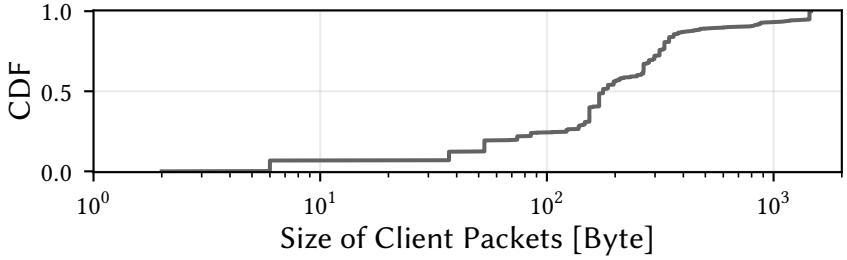
Siri interacts with a few well-known servers. To characterize its traffic, we captured all the packets exchanged with these servers from a university campus network serving several thousands of wireless devices during a week. We focus our analysis on the 18 166 Multipath TCP connections that did not perform any handover to a cellular network since we only captured packets over the Wi-Fi one. This can be assessed with TCP's sequence number and Multipath TCP's Data Sequence Number.

Siri runs over HTTPS, probably to ensure privacy and also to reduce the risk of middlebox interference. The client sends requests that are immediately answered by the server. Figure 3.3 presents some statistics extracted from the Siri trace. 60% of the Siri connections last between 5 and 10 seconds. This time is likely correlated to the time needed to ask a question to the assistant and getting back the reply. Notice that 15% of the connections last more than 20 seconds, perhaps because of successive user interactions with the Siri application. Figure 3.3b shows that 77% of the data packets sent by Siri clients have a size between 50 and 500 bytes. Only 7% of client-generated packets are larger than 1000 bytes. Looking at the connection behavior, several small packets are sent within a few microseconds to form bursts of a few thousands of bytes. This traffic pattern is probably related to the voice sampling process. In comparison, MSS-sized packets represent only 20% of the packets sent by the server. The large fraction of short packets also suggests that the Siri application disables the Nagle algorithm to avoid delaying packet transmissions.

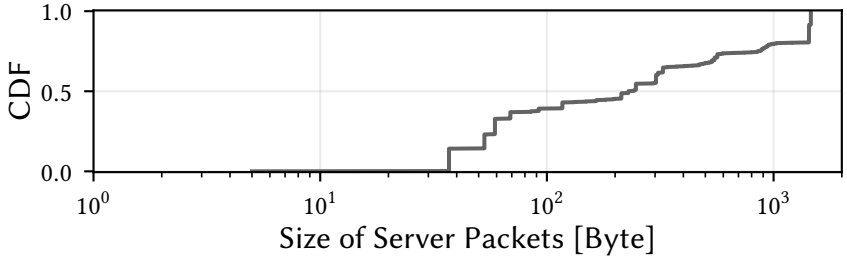
To evaluate the interplays between a simple interactive application and Multipath TCP, we use a simplified model of the observed Siri behavior. Our model is a client-oriented process with three states as shown in Figure 3.4. The client maintains the `sent` counter. In the *Sending burst* state, the client sends a burst of 2500 bytes using packets carrying between 50 and 500 bytes. Then `sent` is incremented and the client waits in the *Inter-burst wait* state before going back to *Sending burst*. Once `sent` reaches `sent_thres`, the



(a) Duration of the Multipath TCP connections created by Apple's Siri.



(b) Size of the data packets sent by Siri clients.



(c) Size of the data packets returned by Siri servers.

Figure 3.3: Analysis of the Siri traces collected on the campus Wi-Fi network.

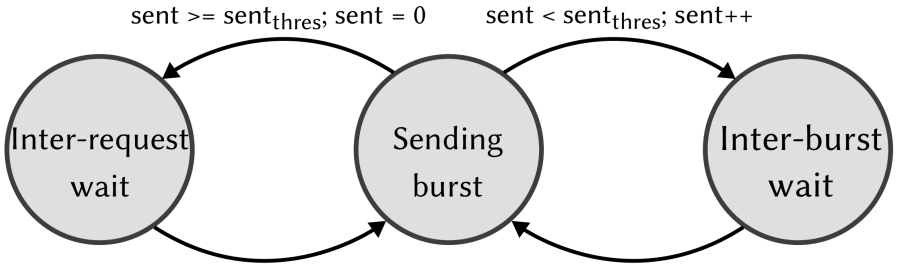


Figure 3.4: Client-side state machine of a simple voice activated application.

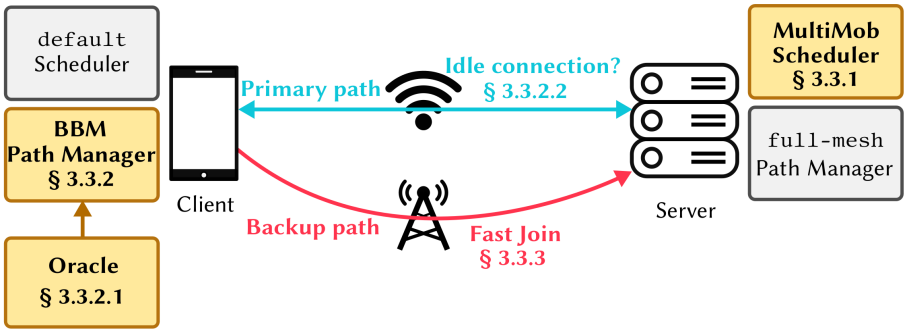


Figure 3.5: MultiMob **high-level architecture.**

client switches to the *Inter-request wait* state representing the delay between successive user interactions. We empirically set `sent_thres` to 9, and the durations of the *Inter-burst wait* and *Inter-user interaction wait* states are 1/3 s and 5 s, respectively. We model the server as a process returning a 750 bytes response after each burst. Our simple client application then collects the delay between each request and its associated server response. Both our client and server applications disable both Nagle's and Cork's algorithms. Unless explicitly stated, all the measurements presented in the following of this Chapter are based on this traffic pattern.

3.3 Tuning Multipath TCP

We now explain how MultiMob improves the Linux Multipath TCP implementation for the smartphone use case. Figure 3.5 illustrates the high-level architecture of MultiMob. We first add to the server's packet scheduler a heuristic that enables it to infer the wireless conditions affecting the client subflows (§3.3.1). Second, we adopt a *break-before-make* path management by implementing an oracle that monitors the network and opens cellular subflows only when needed (§3.3.2). Third, we extend the Multipath TCP protocol so that the client can retransmit data inside the Syn used to create an additional subflow during a network handover (§3.3.3).

3.3.1 Towards Global Packet Scheduling

When a Multipath TCP connection has 2 or more subflows, each of the communicating hosts independently selects its best subflow to transmit each data packet. The Linux implementation relies on the `default` packet scheduler selecting the available subflow having the lowest estimated round-trip-time (RTT). This packet scheduler works well in a variety of environments [Paa+14]. However, selecting subflows only on the basis of their es-

Algorithm 1 isNewerThan(sf, osf)

Input: sf a subflow, osf another subflow or None
Output: Return true if sf

```
1: if osf is None then return true
2: if sf has newer original reception than osf then
3:     return true                          ▷ See text for definition
4: return false
```

Algorithm 2 MultiMob's server-side packet scheduler tracks the subflow that was most recently used by the client.

Output: $bestsf$ next subflow to use

```
1: bestsf ←None
2: for sf in working subflows where data was not sent yet do
3:     if isNewerThan(sf, bestsf) then          ▷ See Algorithm 1
4:         if sf is available then
5:             bestsf ← sf
6: return bestsf
```

timated RTT is not always the best solution. Consider a smartphone user that moves while using the Siri application. It regularly sends small bursts of data and the server returns responses. In an environment where the Wi-Fi network features a lower latency than the cellular network, the packet scheduler will prefer the Wi-Fi path. As the generated traffic is low, its congestion window will remain open as long as the Wi-Fi network behaves well. If the smartphone detects that the Wi-Fi starts being lossy, it will shift its traffic over the cellular subflow. Nevertheless, the server is not aware of the smartphone's motion and its packet scheduler still sends responses over the Wi-Fi subflow because it exhibits the lowest latency. The server will only switch to the cellular subflow after the expiration of its retransmission timer, which potentially wastes from hundreds of milliseconds to several seconds.

To solve this problem, MultiMob includes a server-side packet scheduler that uses the most recent data sent by the smartphone as a hint to select the most suitable subflow. On the smartphone, MultiMob uses the `default` packet scheduler while setting the cellular interface as a backup one. This enables the smartphone to favor the Wi-Fi path and only use the (backup) cellular subflow when the Wi-Fi one experiences retransmissions. Algorithm 2 describes the operations of the server-side MultiMob packet scheduler. It maintains for each subflow the timestamp of the *last original packet* received over it. A packet is considered to be *original* if it contains new data (based on its Multipath TCP Data Sequence Number) or if it successfully concludes a

subflow establishment. Similarly, an acknowledgment is viewed as *original* if its Data ACK advances the lower edge of the sending window. The MultiMob scheduler removes from consideration the potentially failed subflows and the ones where this data has already been transmitted. Then it iterates over all remaining subflows to identify the one having the most recent original reception. If the congestion window of this subflow is not full, the scheduler selects it.

Thanks to the MultiMob packet scheduler, the server can quickly detect the most suitable subflow while taking into account subflow backup preferences. For an interactive application like Siri that sends small requests, the traffic will not completely fill the congestion window and the server will always reply on the subflow that was last used by the client.

3.3.2 Break-Before-Make Path Management

In the Linux kernel implementation, when a Multipath TCP connection starts, the `full-mesh` path manager initiates the exchange over the primary network interface and then additional subflows over the other ones. If we configure the cellular interface as a backup one, data packets will only be sent over that network once the Wi-Fi network fails. This *make-before-break* approach minimizes the amount of data sent over the cellular interface. Unfortunately, this strategy does not minimize energy consumption. Between an exchange carrying a few KB of payload and another one containing only Syn and Fin packets, there is little difference from a power consumption viewpoint. Recall that in Section 3.1.2, we highlighted a set of Multipath TCP connections keeping the cellular interface in the power-hungry RRC_CONNECTED mode by establishing subflows without using them.

MultiMob opts for a *break-before-make* approach and creates subflows over the backup interfaces after having detected failures on the primary ones. With *break-before-make*, the key issue from a performance viewpoint becomes how quickly can the smartphone detect that a wireless interface works badly in order to establish backup subflows. MultiMob includes two mechanisms signaling when the backup interface must be used. First, the client embeds an oracle that monitors the connections' state to create subflows when it experiences sending issues (§3.3.2.1). Second, MultiMob extends the Multipath TCP protocol to inform the peer about additional data to be received (§3.3.2.2).

3.3.2.1 Monitoring Connection Status

When sending packets over a wireless network, the smartphone may observe delivery failures. In particular, we can assume that if a network interface experiences connectivity issues, subflows using it will experience retransmis-

IP_{src}	IP_{dst}	Network interface	TCP subflows	TCP stats
1.2.3.4	4.5.6.7	Wi-Fi	$[tp_1, tp_3]$	`sloss` 2%,...
2.3.4.5	5.6.7.8	Cellular	$[tp_2]$	`sloss` 0%,...
2.3.4.5	4.5.6.7	Cellular	$[tp_4, tp_5]$	`sloss` 15%,...

Table 3.2: Oracle monitoring table example.

sions and losses, even if they belong to different connections. MultiMob relies on this assumption to spot network connectivity problems through a *Multipath TCP oracle* implemented as a kernel module. To track those events, our oracle maintains a monitoring table of netpaths as shown in Table 3.2. A *netpath* is a tuple (IP_{src}, IP_{dst}, network interface). By aggregating this information on a per network flow basis we reduce the size of the monitoring table compared to a per transport flow one. Such structure is well adapted to real deployments using proxies [KT; Tes19] where all Multipath TCP connections are terminated on a proxy.

When a new subflow is created, the oracle checks if its corresponding netpaths is already in the table. If so, the subflow is added to the list of TCP subflows matching this entry. Otherwise, a new entry is created. When a subflow stops, it is removed from the subflow list it belongs to. If a netpath does not have any associated subflow anymore, its entry is removed from the monitoring table.

Our oracle computes every T_s seconds statistics based on the subflows associated to a given netpath. Our implementation collects three[1] metrics: smoothed loss rate (`sloss`), smoothed retransmission rate (`sretrans`) and maximum RTO. Those statistics are computed based on the per-subflow state maintained by the Linux kernel. It also takes into account Tail Loss Probes (TLP) [Duk+13]. When the TLP timer fires, we enter Forward Acknowledgment mode and the packet at the head of the write queue is marked as lost.

The oracle computes the smoothed rates — `sloss` and `sretrans` — by using Volume-weighted Exponential Moving Averages (V-EMA), similar to what Android uses to estimate the loss rate of Wi-Fi beacons. These V-EMAs reduce to the three following equations

$$val_{i+1} = \frac{prod_{i+1}}{vol_{i+1}} \tag{3.1}$$

[1]The perceptive reader will notice that we collect information on an interface basis without considering link-layer metrics such as Wi-Fi beacons. Actually, we wanted to compile such information but we faced practical issues. Nexus 5 smartphones use a Broadcom Wi-Fi card and a Qualcomm GSM one both requiring proprietary drivers and we did not found a way to collect their metrics in the kernel. As these data are made available to the Android OS, it should be possible to make them available for the Linux kernel too. Yet, we leave this for future work.

Parameter	Meaning
$\texttt{sloss}_{\texttt{thres}}$	Threshold value of `sloss` that triggers backup subflow creations on the netpath
$\texttt{sretrans}_{\texttt{thres}}$	Threshold value of `sretrans` that triggers backup subflow creations on the netpath
T_s	Period between each V-EMA computation
α	Sensitivity parameter of the V-EMA equations

Table 3.3: The different parameters of the oracle.

$$prod_{i+1} = \alpha(val_{new} \cdot vol_{new}) + (1 - \alpha)prod_i \tag{3.2}$$

$$vol_{i+1} = \alpha \cdot vol_{new} + (1 - \alpha)vol_i \tag{3.3}$$

where val_{new} is the new value of the studied metric (e.g., number of lost sent packets during the last T_s period), vol_{new} is the volume of this new value (e.g., total number of packets sent during the last T_s period), $prod_i$ is the product at iteration i, vol_i the volume at iteration i and val_i the value of the V-EMA at iteration i. Note that $prod_0 = vol_0 = val_0 = 0$ and no value is computed if $vol_{new} = 0$. $\alpha \in [0, 1]$ is a sensitivity parameter experimentally set to 0.5, as Android.

The MultiMob oracle sets thresholds to detect under-performing netpaths. Once a threshold is crossed, MultiMob triggers the creation of backup subflows for all connections associated to the under-performing netpath. It also marks the subflows associated with that netpath as potentially failed. Since the oracle is part of the Linux kernel, it can query the state of all established Multipath TCP connections. It can then prompt backup subflows creation once any connectivity issue is detected. All the oracle parameters are summarized in Table 3.3.

3.3.2.2 Signaling Idle Connections

As it is based on indications of sending issues, our Multipath TCP oracle alone is not sufficient to trigger the backup interface usage under all possible traffic patterns. In particular, consider a smartphone-initiated download. During such transfer, the server pushes data towards the client. The smartphone only sends Acks to the server. However, TCP does not ensure the reliable delivery of the Ack packets once they do not contain any payload. If the primary subflow fails, the smartphone will stop receiving data, but it is difficult for the Multipath TCP stack to distinguish between losses in the network and the server application becoming idle for any reason, especially when no other network traffic is running concurrently.

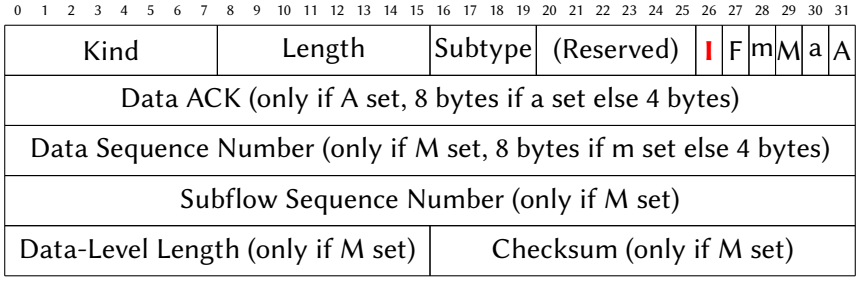


Figure 3.6: The modified Multipath TCP Data Sequence Signal option.

A first naive approach would be to let the client regularly probe the server with keep-alives. Such active probing would consume energy and could degrade wireless performance [Hu+15]. With HTTP, another solution would be that a Multipath TCP aware client uses the `Content-Length` header sent by the server to detect idle times in the middle of a transfer to create an additional subflow, e.g., via the enhanced socket API [HB16].

To let all applications benefit from such information without modifying them, MultiMob extends Multipath TCP so that the server can indicate to the client that a data transfer is not finished yet. We define two signals. The first one is sent in the Multipath TCP Data Sequence Signal (DSS) option. We take one of the unused bit in the reserved field of the DSS and call it the **I**dle bit, as shown in Figure 3.6. A host sets the Idle bit when it sends a data packet that empties its send buffer. Otherwise, the Idle bit is reset. Since this bit is included in the DSS option, it is sent reliably to the peer. Then, a receiver does not expect a connection to be idle unless it has received a DSS option with the Idle bit set. Notice that the Multipath TCP's Idle bit differs from the TCP's Push one used to signal that the carried data must be promptly delivered to the receiver. Although the Push bit is typically set when the carried data empties the sending buffer (like with SSH), our experiments revealed that it is often set in many MSS-sized packets during bulk exchanges. The second signal indicates after how much idle time the receiving host should trigger the creation of backup subflows. For this, we rely on the Experimental Multipath TCP option [RFC6824B] that carries the current value of the sender's RTO. The client sends an request RTO option from time to time and the server returns the same option containing its current retransmission timer $\text{RTO}_{\text{server}}$. The receiving host uses that value to set its idle timer at $\max(500\text{ ms}, \text{RTO}_{\text{server}})$. This timer is restarted each time it receives a data packet having its Idle bit reset, and stopped if the last data packet had the Idle bit set. If the timer expires, the stack informs the oracle that triggers the creation of backup subflows.

3.3.3 Immediate Reinjections

The *break-before-make* approach described in the previous section is beneficial from an energy viewpoint. However, a mobile typically detects the failure of a wireless interface by the expiration of its retransmission timer or using Tail Loss Probe (TLP). This unacknowledged data can only be retransmitted over another interface once a subflow has been established over this network. Such establishment requires a four-way handshake, as described in Section 1.3.2. While this ensures complete authentication of both connection participants, this handshake unfortunately delays the reinjection of the lost data, since the client needs to wait for two network round-trip-times on the backup interface.

To reduce this delay, we extend Multipath TCP to support the transmission of data inside Syn packets and immediately after Syn/Ack ones. For this, we define two new Multipath TCP options: Fast Join In and Fast Join Out. These are different from TCP Fast Open [Rad+11; rfc7413] because a new subflow is established between hosts that already share state for one Multipath TCP connection.

A naive approach would be to simply place data inside the Syn and require the server to accept it immediately. Unfortunately, this solution would induce obvious security problems because this Syn is not authenticated. With Multipath TCP, the Syn initiating an additional subflow carries the Mp Join option containing a random number used to authenticate the server in addition to the 32-bit token that identifies the connection. This token alone is not sufficient to authenticate the client because a passive listener might have observed it during the establishment of a previous subflow for the same connection [rfc6181], e.g., on an open Wi-Fi network.

The Fast Join Out option described in Figure 3.7 addresses this issue and allows the client to carry data in the initial Syn without causing any security issue. Note that as there is only 40 bytes for TCP options and current popular ones found in Syn packets are Maximum Segment Size (4 bytes), Selective Ack (2 bytes), Timestamps (10 bytes) and Window Scale (3 bytes + 1 padding byte), the Fast Join Out option must be engineered to fit into 20 bytes. As highlighted by Figure 3.8, the Fast Join Out option in the initial Syn is 20-byte long and contains three important fields. First, the token Token_S identifies the Multipath TCP connection as in the Mp Join option. Second, the Data Sequence Number (DSN) indicates the Multipath TCP sequence number of the data contained in the Syn payload. The associated DSN_C must be the first unacknowledged data byte. Third, the 32-bit truncated HMAC (HMAC_C) authenticates the client. Its computation is the following

$$\text{HMAC}_C = \text{HMAC}_{K_S \,\|\, K_C}(\text{Token}_S \,\|\, \text{DSN}_C) \tag{3.4}$$

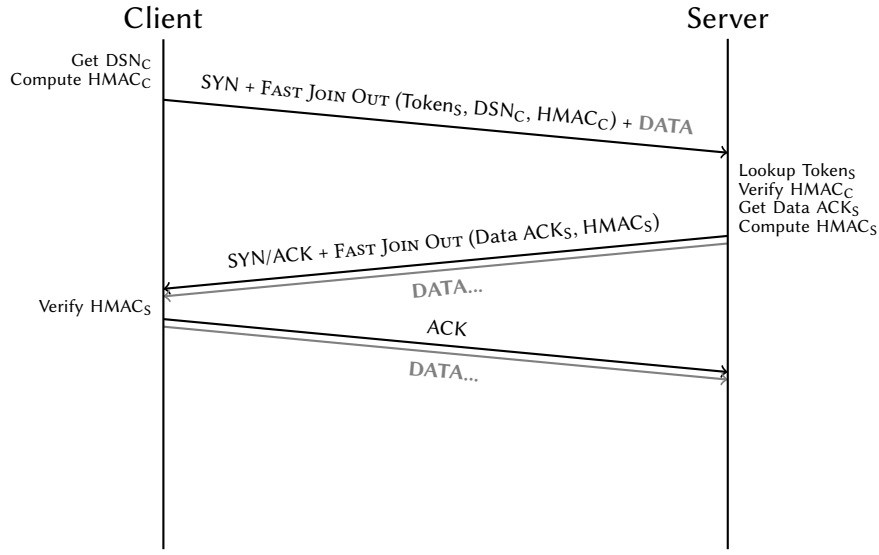


Figure 3.7: With Fast Join Out, the client can directly send data inside the Syn packet.

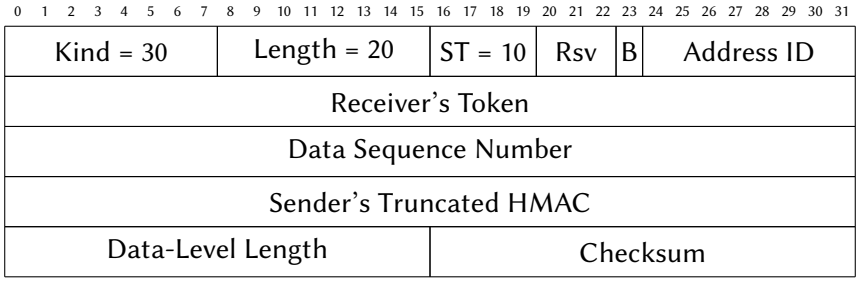


Figure 3.8: The Fast Join Out option in the initial Syn.

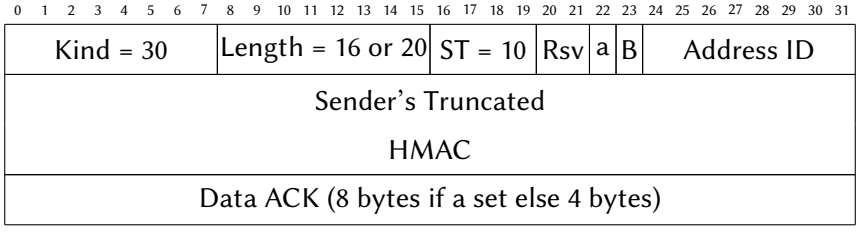


Figure 3.9: The Fast Join Out option in the Syn/Ack.

where K_C and K_S are respectively the connection keys of the client and server that were exchanged during the Multipath TCP connection handshake. To prevent replay attacks, our implementation accepts only one SYN containing the FAST JOIN OUT option for a given DSN_C. To be eligible, the DSN_C present in the FAST JOIN OUT option should match the one expected by the server. However, ACKS packets might not have reached the client. To cope with this situation, we define EDSN as the next Expected DSN by the server and DSN_C as the DSN contained in the FAST JOIN OUT option of the SYN packet. The server accepts the establishment of the subflow if the following inequality holds

$$EDSN - wnd_{rcv} \leq DSN_C \leq EDSN \tag{3.5}$$

where wnd_{rcv} is the receive window of the server. Notice that the FAST JOIN OUT option also carries both the Data-Level Length and Checksum fields. Similar to the Multipath TCP DSS option, this information enables the receiver to detect middlebox interference affecting the data. The B bit indicates whether the client wants to consider this new subflow as a backup one.

To authenticate the server, it computes the following $HMAC_S$

$$HMAC_S = HMAC_{K_C \,||\, K_S}(DSN_C \,||\, Data\ ACK_S) \tag{3.6}$$

where DSN_C is the Data Sequence Number contained in the SYN packet and Data ACK_S is the value of the acknowledged Multipath TCP sequence number, possibly updated by the data received. The server then sends a SYN/ACK packet with the FAST JOIN OUT option carrying both the computed $HMAC_S$ and the updated Data ACK_S, as described in Figure 3.9. Once it sent the SYN/ACK packet, the server can start sending data. At the reception of the SYN/ACK packet, the client first checks that the Data ACK_S value is in the expected range, i.e.,

$$DSN_C + Data\text{-}Level\ Length_C \leq Data\ ACK_S \leq LUSB_C + 1 \tag{3.7}$$

where Data-Level $Length_C$ is the Data-Level Length value sent in the SYN packet and $LSUB_C$ is the largest unacknowledged sent byte by the client over the Multipath TCP connection. If the inequality holds, the client can check the computed $HMAC_B$ using Equation 3.6. If it matches, it replies with an ACK packet and the additional subflow is established.

The FAST JOIN OUT option is useful when the mobile client sends data to a remote host. However, there are situations where only the server pushes data towards the client. A first typical example are streaming applications where the server pushes data at a regular rate. Another example is a long client-initiated file download. When the oracle running on the mobile client detects losses or the absence of data, it may want to quickly establish a subflow without having data to send to the remote host. Such path creation can

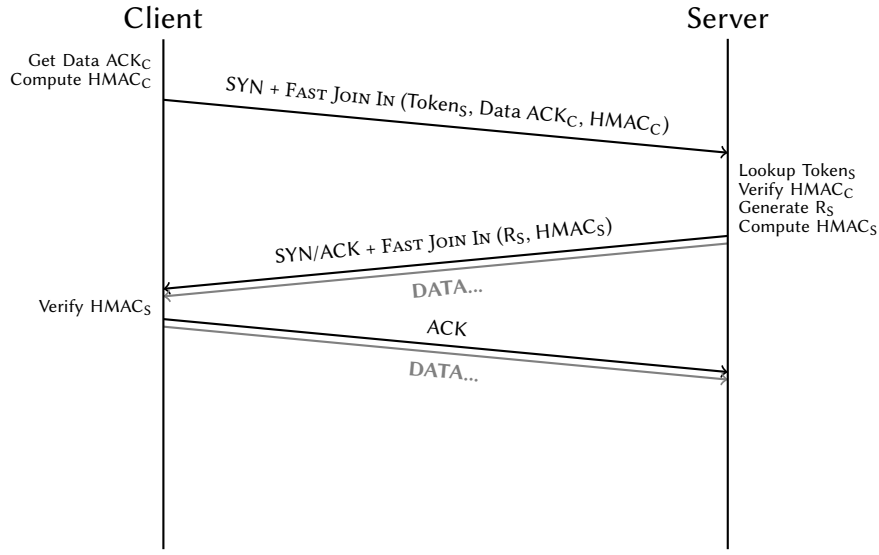


Figure 3.10: With Fast Join In, the client can recover a data transfer in only one round-trip-time instead of two.

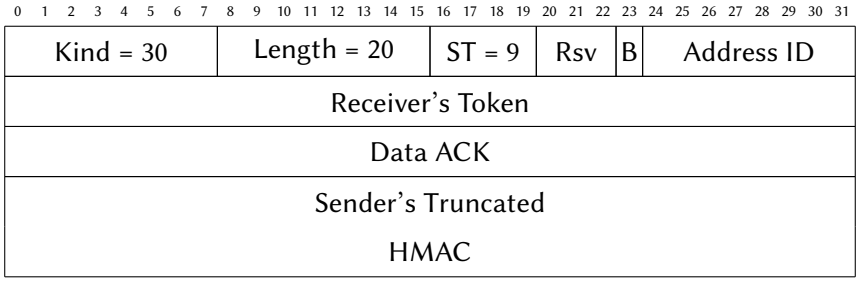


Figure 3.11: The Fast Join In option in the initial Syn.

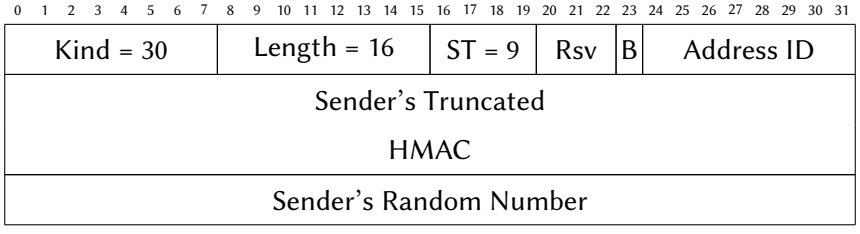


Figure 3.12: The Fast Join In option in the Syn/Ack.

then enable the server to continue its data transfer. This case is covered by the Fast Join In option whose operations are described in Figure 3.10. Actually, this option is very similar to the Fast Join Out one. Instead of computing $HMAC_C$ with the DSN, Fast Join In uses the current client-side Data ACK value.

$$HMAC_C = HMAC_{K_S \parallel K_C}(Token_S \parallel Data\ ACK_C) \tag{3.8}$$

The client then sends this value along with the associated Data ACK_C in the Fast Join In option in the Syn packet as shown in Figure 3.11. The Syn packet is only considered if the carried Data ACK_C fits in the expected server range, i.e.,

$$SUSD_S \leq Data\ ACK_C \leq LUSD_S + 1 \tag{3.9}$$

where $SUSD_S$ and $LUSD_S$ are respectively the smallest and the largest unacknowledged sent bytes by the server. If the inequality is true, the receiving host can verify the $HMAC_A$ computation using Equation 3.8. In order to prevent replaying attacks, the server only accepts a given Data ACK_C value once for a given connection. Note that compared to the Fast Join Out one, the Fast Join In option in the Syn packet does not carry Data-Level Length nor Checksum fields, as there is no payload in such packets. This leaves more space to support 8-byte long HMAC.

The main difference between Fast Join Out and Fast Join In resides in the Syn/Ack packet. While the Data ACK_S provided in the Fast Join Out Syn/Ack can possibly acknowledges previously sent client data, the next DSN to be sent by the server is necessarily Data ACK_C. To avoid deterministic value, the server instead generates a random value R_B as in classical Mp Join and communicates it to the client as shown in Figure 3.12. Therefore, the $HMAC_S$ is computed as follows.

$$HMAC_S = HMAC_{K_C \parallel K_S}(Data\ ACK_C \parallel R_S) \tag{3.10}$$

Once the Syn/Ack is sent, the server can immediately transfer data over this new subflow. At the other side, once the Syn/Ack has been received and the $HMAC_S$ verified, the client considers the subflow as established. With Fast Join In, a download-only data transfer can be resumed within one round-trip-time, instead of two with normal Mp Join.

3.4 Emulation Results

In this Section, we evaluate each of the MultiMob components in Mininet environment [Han+12]. We consider a scenario with two disjoint paths between the client and the server as illustrated in Figure 3.13. Our emulations use Multipath TCP v0.91 in the Linux kernel 4.1. Unless explicitly stated, the primary path exhibits a round-trip-time of 15 ms and the additional one

Figure 3.13: The studied network topology in our emulations.

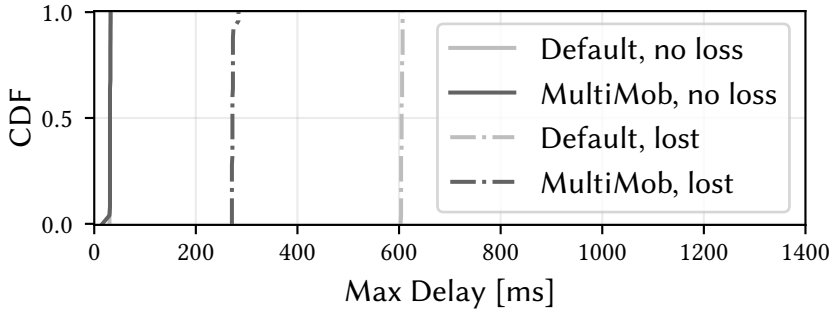


Figure 3.14: Maximum delay to return an answer to simplified interactive requests. Each scenario ran 25 times. 100% losses generated by a `netem` command begin to occur on the primary path when the client is in *Inter-request wait* state, i.e., between request bursts.

25 ms. Both paths have a bandwidth of 10 Mbps and the router queue sizes are equal to the bandwidth-delay product. When the simplified interactive traffic is tested, it runs for 20 s.

We first evaluate the impact of the MultiMob server-side scheduler (§3.4.1). We then show that the packet scheduler alone is not sufficient and discuss how the operations of the oracle are affected by the threshold values (§3.4.2) and the periodicity of the V-EMAs computations (§3.4.3). Next, we demonstrate the benefits of the Idle bit (§3.4.4) and finally assess the time saved by using Fast Join Out compared to the normal Mp Join (§3.4.5).

3.4.1 MultiMob Server-Side Packet Scheduler

In this simple scenario, the client opens the connection over the primary path and then creates a backup subflow over the additional one. We test different packet schedulers on the server. Figure 3.14 shows that when there is no loss, both the `default` and the MultiMob server-side packet schedulers exhibit quite similar performances. Notice that since the Linux sysctl `tcp_slow_start_after_idle` is set to 1, a request can be answered within two RTTs

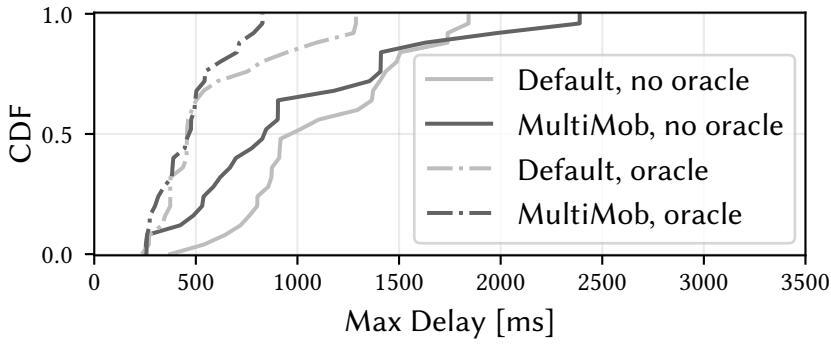


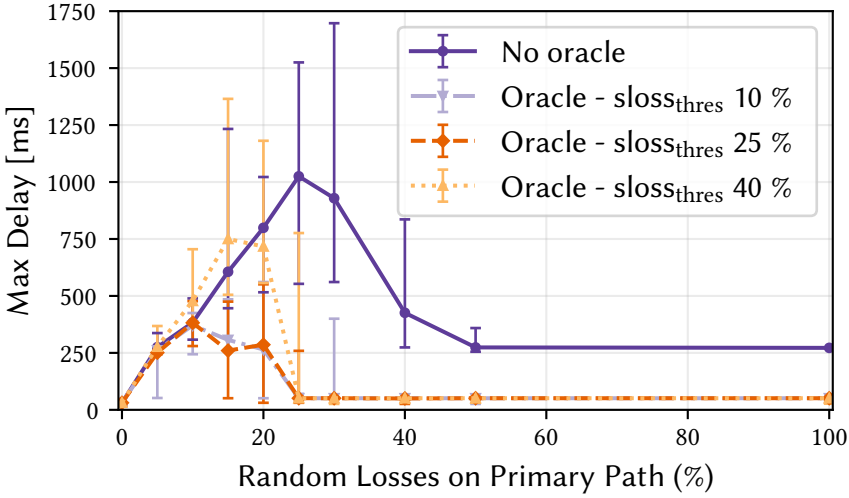
Figure 3.15: Maximum delay to return an answer to simplified interactive requests. Each scenario ran 25 times. 25% losses generated by a `netem` command begin to occur on the primary path when the client is in *Inter-request wait* state, i.e., between request bursts.

if its sending phase generates more packets than the initial congestion window (10 packets). However, when the primary subflow fails between two requests, the MultiMob packet scheduler reduces the maximal experienced delay by a factor of two compared to the `default` one. When the client sends its first request after a loss, it experiences a RTO before reinjecting the packets on the additional path. Nonetheless, the server is not aware that the primary subflow failed. With the `default` packet scheduler it sends the reply on the primary lossy path. It therefore also experiences a RTO before reinjecting the response on the additional subflow. Since the MultiMob server-side scheduler follows the last client decision, it does not experience RTO and hence reduces the delay perceived by the application.

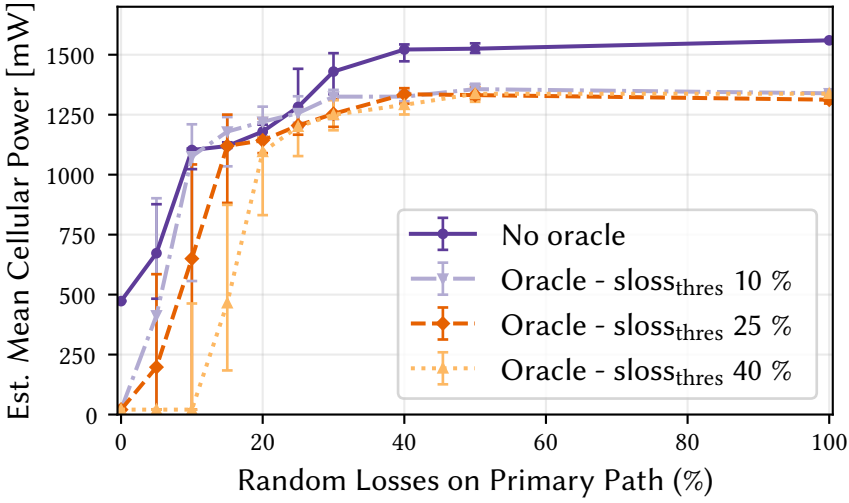
Notice that while the MultiMob scheduler works well when the primary path completely fails, it is not sufficient to handle all loss patterns. Figure 3.15 illustrates that when the primary path exhibits 25% random losses, the performance of both the `default` and the MultiMob packet schedulers are less predictable. At client side, the `default` packet scheduler still prefers the lowest latency subflow, even if lossy. The path selected by the server-side MultiMob packet scheduler is function of the last received packet. While the use of the oracle still lets the application experience delays of several hundreds of milliseconds, it helps to decrease their variability.

3.4.2 Influence of the Threshold Values of the Oracle

One of the foundations of our oracle is to consider currently active connections as connectivity probes. Continuous and bulk transfers typically have a



(a) Maximal delay to answer a simplified interactive request.



(b) Estimated mean power consumption of the additional path, assuming it is a cellular interface, based on our model [Hua+12].

Figure 3.16: Simplified interactive requests with light continuous background traffic. The second interface is set as backup. If any, the loss event begins when the client is in *Inter-request wait* state, i.e., between request bursts. The oracle periodicity T_s is 100 ms. Markers show medians and error bars 25th and 75th percentiles over 25 runs.

better network view than on/off and idle ones. When they suffer from losses, such transfers make the oracle trigger the creation of backup subflows for all related connections, even idle ones. This enables limiting the number of connections experiencing RTOs before using the backup path.

To assess this, we set up a light background request/response flow (12 KB/s) and generate simplified interactive request traffic. To observe how the threshold of V-EMA values affects the operations of the oracle, we focus here on the smoothed loss rate one ($\texttt{sloss}_{\texttt{thres}}$). Figure 3.16a shows the maximum perceived application latency to answer a request and Figure 3.16b estimates the energy consumption of the additional backup path, considering that it is an always-on cellular interface. Without any loss, we observe similar request delays between all configurations. Notice that when the oracle is present, it prevents the creation of the backup path. This decreases the estimated power consumption compared to the default `full-mesh` behavior directly creating the backup subflow after the connection establishment. When losses occur on the primary path, our oracle detects that the background traffic experiences connectivity issues, and therefore creates backup subflows for all connections using that netpath. Then, the simulated interactive client can directly use the additional path and does not experience any RTO. Since the server relies on the MultiMob packet scheduler, it replies on the subflow used for the last request. On the contrary, if there is no oracle, the interactive connection client must experience a RTO before using the additional path, even if the additional subflow is always established at the beginning of the connection. Furthermore, when the link is very flappy (20 to 30% of random losses), without the oracle the client tries to reuse the lossy primary path once some packets manage to use it. Our oracle prevents this behavior by considering the primary path as potentially failed, discouraging the smartphone's packet scheduler to reuse the primary path.

When using the oracle, the creation of additional subflows depends on the network conditions and the $\texttt{sloss}_{\texttt{thres}}$ parameter. When set to a low value like 10%, the interactive traffic client experiences low delays but the backup path is quickly created, as a few losses suffice. With higher threshold values like 40%, the additional path remains closed in the median case when the primary path experiences 10% of random losses. However, the interactive traffic can experience high latencies, possibly larger than without the oracle, as it cannot shift the traffic on the non-existent backup path. Based on those emulation results, we experimentally set $\texttt{sloss}_{\texttt{thres}}$ to 25% for real mobile devices as a reasonable trade-off between low-latency and low additional path use. The $\texttt{sretrans}_{\texttt{thres}}$ is set to 50% and the maximum RTO threshold is empirically set to 1.5 s to avoid using a subflow that might hurt interactivity because of the lack of retransmission responsiveness.

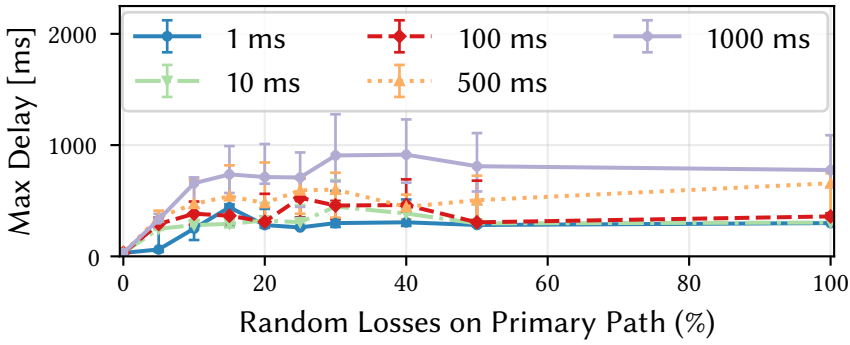


Figure 3.17: Varying T_s with simplified interactive traffic only, with $\texttt{sloss}_{\texttt{thres}}$ and $\texttt{sretrans}_{\texttt{thres}}$ respectively set to 25% and 50%. Markers show medians and error bars 25$^{\text{th}}$ and 75$^{\text{th}}$ percentiles over 20 runs.

3.4.3 Influence of Oracle Periodicity

As suggested by the equations of the V-EMA, the reactivity of the oracle also depends on the smoothed computation periodicity T_s. Indeed, as Figure 3.17 shows, the lower T_s, the quicker the reaction of the oracle to losses and the lower the variability of the detection. A value of 1 ms allows a very fast reaction, but the oracle might spend a lot of CPU time to update its monitoring table. On the contrary, a value of 1 s is not responsive enough. In the remaining of our experiments, we empirically set T_s to 500 ms to match sub-second reactivity while keeping low CPU usage on mobile devices.

3.4.4 Bulk Download and Primary Subflow Loss

In Section 3.3.2.2, we motivated that monitoring the connections sending data is not sufficient when the smartphone only receives data. To illustrate this situation, Figure 3.18 plots the time-sequence graph of file download from the receiver's viewpoint. After about half a second, the client detected that it did not receive data anymore while the Idle bit was not set in the DSS option. Therefore, it triggers the creation of a subflow on the backup path. Once it experiences a RTO, the server starts using the newly created subflow and the data transfer succeeds. Notice that in our experience, the retransmissions at 6 s were caused by a burst of duplicate Acks.

3.4.5 Fast Join Benefits

The primary goal of our proposed Fast Join options is to decrease the establishment time of backup subflows. To assess this, we evaluate the benefits of

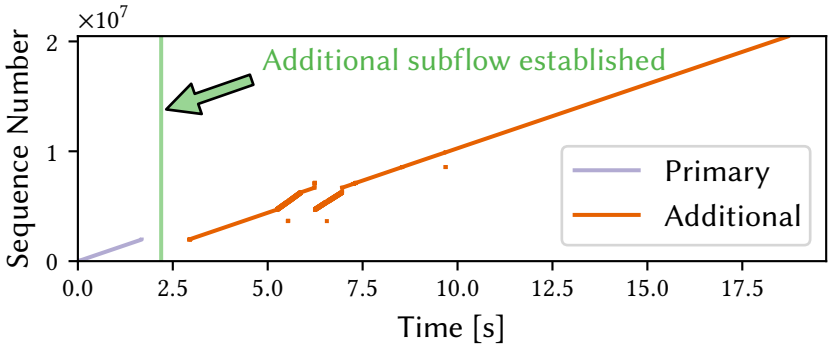


Figure 3.18: Time-sequence graph of the packets received by a client during a 20 MB HTTP GET. The primary subflow suffers from 100% losses at 1.5 s.

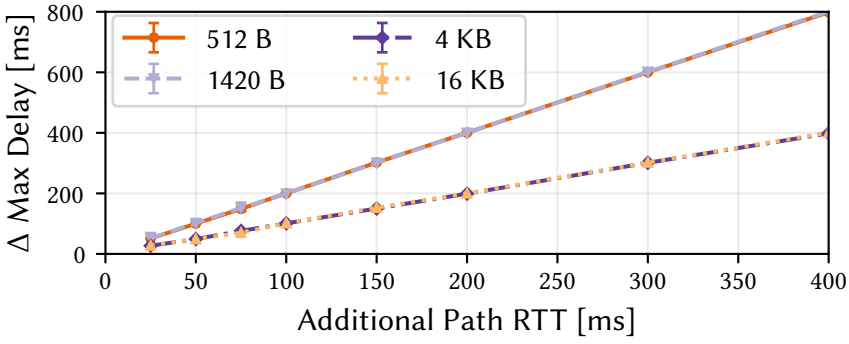


Figure 3.19: Difference of maximum application perceived delays between regular MP JOIN and FAST JOIN OUT depending on the request size. To limit the detection variability, we set the oracle periodicity T_s to 1 ms. Markers are medians over 6×2 runs, bars show minimum and maximum.

the Fast Join Out option with regular request/response traffic where the primary path experiences 100% losses after five seconds. Figure 3.19 shows the delta of the application-perceived delays of the first request following the loss event using either Mp Join or Fast Join Out. When the request fits inside a single TCP packet, as for the popular Siri application, the Fast Join Out provides immediate client data reinjections and the responses can be received after one RTT instead of three RTTs with Mp Join. If the request requires several TCP packets to be sent, Fast Join Out still saves one RTT compared to the regular Mp Join. These results confirm our intuitions provided by Figures 1.9 and 3.7.

3.5 Performance Evaluation

This section presents the evaluation of MultiMob on Nexus 5 smartphones running a slightly patched Android 6.0.1. To perform our evaluation, we backported Multipath TCP v0.89 to the Linux 3.4 kernel on Nexus 5 phones. Four client configurations are studied.

1. No Backup: the regular Multipath TCP behavior with the `full-mesh` path manager, without considering the cellular as a backup network;

2. Backup: same as No Backup, but considering the cellular as a backup and using it once the primary Wi-Fi path is potentially failed, i.e., it experiences RTOs — this is the default backup behavior since Multipath TCP v0.90;

3. IETF Backup: same as Backup, except that the backup path is used only if there is no more primary Wi-Fi subflow, i.e., all the Wi-Fi subflows have been torn down — this was the backup behavior until Multipath TCP v0.89;

4. MultiMob: our proposed Multipath TCP tuning described in Section 3.3.

In the remaining of this section, we often call the three first configurations as the *vanilla Multipath TCP configurations*. We use two different servers. The first one, using the `default` packet scheduler, is used by vanilla Multipath TCP configurations. The second one, leveraging the MultiMob server-side packet scheduler, is used by the MultiMob configuration.

We first explore particular use cases with micro-benchmarks to understand the benefits of MultiMob (§3.5.1). We then compare at a larger scale vanilla Multipath TCP with MultiMob through active measurements performed on a set of modified Android 6 devices used by real users (§3.5.2).

3.5.1 Micro-Benchmarks

We first compare the standard Android 6.0.1 network stack with MultiMob (§3.5.1.1). We then study actual user mobility with both simplified interactive and streaming applications (§3.5.1.2).

3.5.1.1 Android Network Handover

Android 6.0.1 includes a network interface controller. Like iOS, Android favors the Wi-Fi connectivity over the cellular one and sets the Wi-Fi interface as the default one. By default, once connected to a Wi-Fi network, Android switches off the cellular interface. We patch Android so that the cellular antenna always remains on, even when getting Wi-Fi connectivity.

To avoid staying on underperforming Wi-Fi networks, Android 6 monitors the Wi-Fi interface and computes layer-2 transmission and reception success rates with a V-EMA [AOSP]. Android then computes a network score that combines these success rates, the Received Signal Strength Indication (RSSI) and the throughput of the Wi-Fi network. If the Wi-Fi score drops below the cellular one, Android selects the cellular network as the default one and tears down the Wi-Fi connectivity.

When an Android smartphone associates to a Wi-Fi access point (AP), it first sends a HTTP request to a Google server. This enables the device to detect if the AP is behind a captive portal. If the request succeeds, the Wi-Fi connectivity is considered as functional. Nevertheless, Android does not seem to monitor the performance of the selected Wi-Fi network besides tracking the Wi-Fi beacons. This can cause application degradations when the device is connected to overloaded shared Wi-Fi APs [AEK09]. In such networks, priority is given to the network subscriber, the leftover capacity being shared among the other users. If the network subscriber makes intensive usage of its network, the community users might experience consequent losses.

To mimic such situation, we connected two smartphones to the same Wi-Fi AP providing Internet access with a distance of about a meter. Both run our simplified interactive application without any *Inter-request wait* time — i.e., `sent_thres` $= \infty$ — for 80 seconds. One smartphone uses plain TCP while the other one includes MultiMob. After 5 s, the WAN interface of the Wi-Fi AP starts experiencing random losses on outgoing packets. This lossy period lasts 50 s, after which the random loss is removed. Figure 3.20 shows the maximum application perceived delay by the simulated interactive client depending on the percentage of random losses on the WAN interface. With low loss percentages ($\leq$ 10%), both phones experience similar delays. This is because losses can be recovered with fast retransmissions, and the experienced loss rate is not sufficient for the MultiMob oracle to trigger the cellular subflow creation. However, with higher loss rates, while MultiMob switches

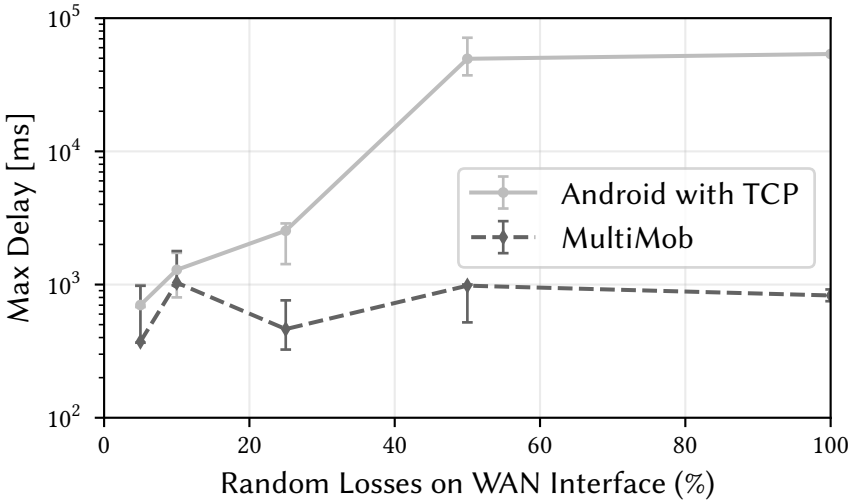


Figure 3.20: Maximal application perceived delays for simplified interactive traffic without any *Inter-request wait* time over 3 runs, showing minimal, median and maximal values.

to the cellular network while keeping sub-second maximum application delays, Android persists in using the Wi-Fi network and does not break the TCP connection. From Android's viewpoint, the Wi-Fi performance remains good because Wi-Fi beacons are still successfully received, although Internet connectivity itself is severely altered. This experiment shows that MultiMob, initially being single-path, can react to such connectivity failures while plain Android continues to use underperforming networks.

3.5.1.2 Mobility Scenarios

To evaluate how MultiMob performs in changing wireless conditions, we go for the short walk presented in Figure 3.21 with two smartphones. The first one uses the MultiMob configuration while the other one has a vanilla Multipath TCP one. Our walk begins at **A**, close the Wi-Fi AP. Starting from **C**, the Wi-Fi signal becomes weaker given the distance and the presence of trees and buildings. Android usually detects the loss of the Wi-Fi signal and tears down the Wi-Fi network at location **D**. From spot **F**, the Wi-Fi signal becomes available again.

Simulated Interactive Traffic. Each of our test phones sends 100 requests during our 80 s walk from **A** to **D**. Figure 3.22 shows the instantaneous mean over the test duration of the fraction of packets that are carried by the cellular

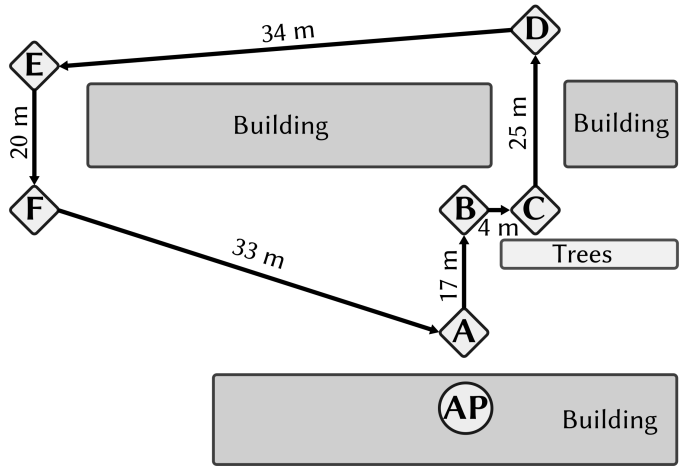
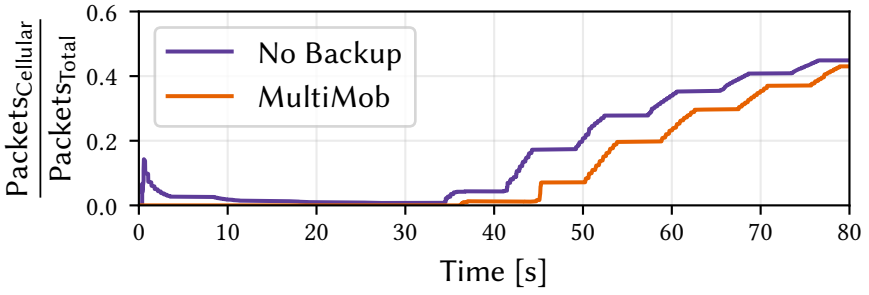
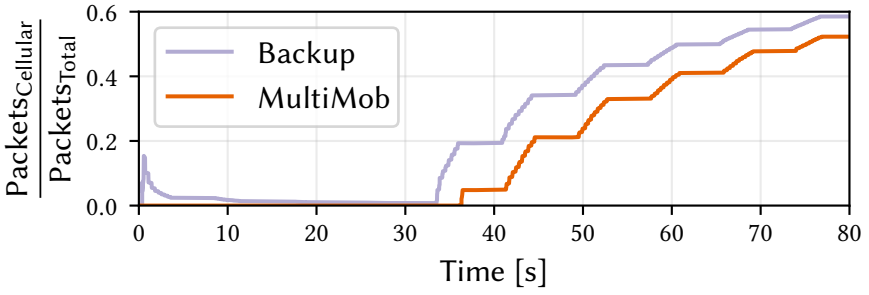


Figure 3.21: Walk map for mobile micro-benchmarks.



(a) No Backup **vs.** MultiMob.



(b) Backup **vs.** MultiMob.

Figure 3.22: Evolution of the mean fraction of total packets carried by the cellular network for the simplified interactive traffic.

Configuration	Max Delay (ms)	Requests Answered	Mean Estimated Cellular Power (mW)
No Backup	1112	100	884
Backup	780	100	885
IETF Backup	21938	84	558
MultiMob	1187	100	657

Table 3.4: Aggregated results from simulated interactive micro-benchmarks. MultiMob shows the median value over the three runs.

interface for both runs. In addition, Table 3.4 shows aggregated results related to these tests. In the vanilla Multipath TCP configurations, the cellular subflow is always created at the beginning of the connection, but no data packet is sent on that network while the Wi-Fi connectivity remains functional. This is expected for both Backup and IETF Backup cases, and the larger observed RTT on the cellular network combined to the low induced network load explain the No Backup result. When the client requests start being lost between locations **C** and **D**, the cellular network is used to recover the connectivity, except for the IETF Backup case which waits for the system to tear down the Wi-Fi subflow before using the cellular one. This leads to missed requests (as the sending buffer fills up and gets full) and very large experienced application delays. On the contrary, both the No Backup and Backup cases leverage the already established cellular subflow to reinject requests over it as soon as an RTO occurs on the Wi-Fi path. MultiMob needs first to detect the connectivity loss with its oracle before establishing the cellular subflow. Yet, its experienced maximum application delay remains similar to those of No Backup and Backup cases. On the opposite, MultiMob consumes less cellular energy since it delays the utilization of the LTE network.

Fixed Rate Streaming Traffic. For this test, we configure the smartphone to stream a web radio over HTTP while performing twice the walk presented in Figure 3.21. Our servers relay the same web radio at a constant bit rate using Icecast. Since all the data flows from the server to the client, all the packet scheduling decisions are made by the server. We compare each vanilla Multipath TCP configuration with MultiMob over a dozen of runs. Notice that since the IETF Backup configuration clearly underperforms the other ones, we do not discuss it in the remaining of this section.

Figure 3.23 shows the time-sequence graph for the No Backup vs. MultiMob test. We did not observe any stall during our dozen of experiments, which is important for a streaming application. However, the No Backup smartphone receives data nearly exclusively on the cellular interface, even

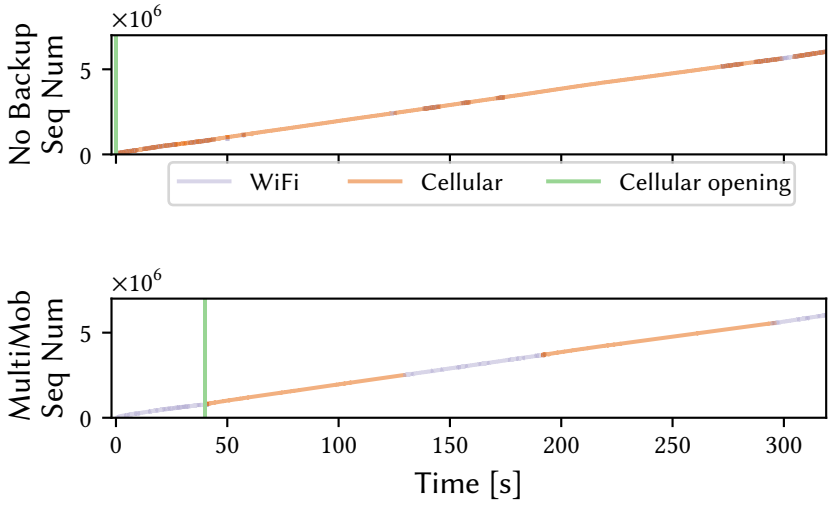


Figure 3.23: Time-sequence graph of the server streaming flow as perceived by the client for the run comparing No Backup **vs.** MultiMob**. Color indicates on which interface the packet was received.**

when the Wi-Fi network is available. From the server perspective, the cellular subflow appears to be more stable with often a lower estimated smoothed RTT than the Wi-Fi one. This explains why the `default` packet scheduler prefers using the cellular network. On the other hand, with its associated packet scheduler, MultiMob forces the server to use the Wi-Fi path when still being used by the smartphone. The Wi-Fi to cellular (between **C** and **D**) and LTE to Wi-Fi (between **F** and **A**) handovers are distinguishable on the MultiMob graph. Furthermore, notice that the MultiMob phone waits for 40 s before opening the cellular subflow. This corresponds to the instant the smartphone's receive timer detects that no more data is received after some time without having received the Idle bit. Based on our cellular power consumption model, the No Backup smartphone would have consumed 444 J (1386 mW) while the MultiMob one spent 329 J (1028 mW).

In the Backup vs. MultiMob test, the usages of the network interfaces are similar, i.e., the Wi-Fi network is used when available. Over a dozen of runs we did not observe any stall, except for a specific test that impacted both Backup and MultiMob. Figure 3.24 shows the buffer playing time on the client for that particular run. At 50 s (first **C** to **D** pass), the MultiMob smartphone faces a half-second stall time. This is due to the reception of a retransmitted packet on the Wi-Fi network while the cellular subflow was already established. Since packets are acknowledged on the subflow they came

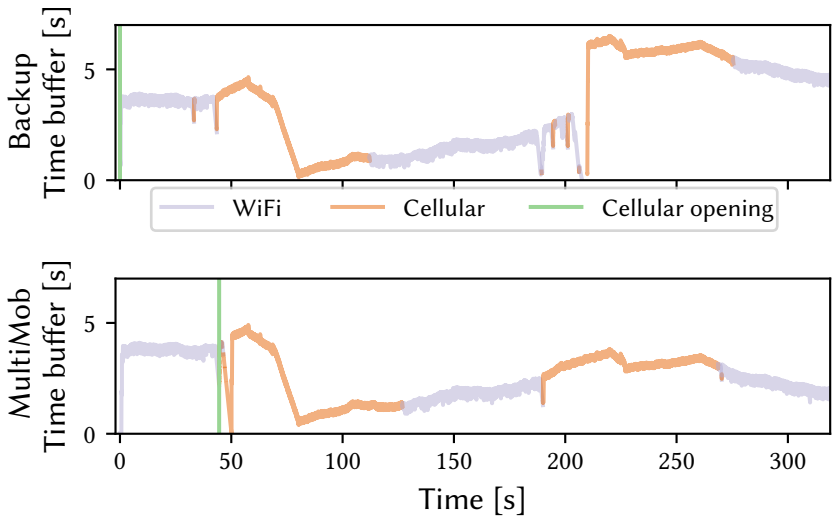


Figure 3.24: Playing time of the client buffer for the worst case test in Backup **vs.** MultiMob**. Color indicates on which interface the last packet was received.**

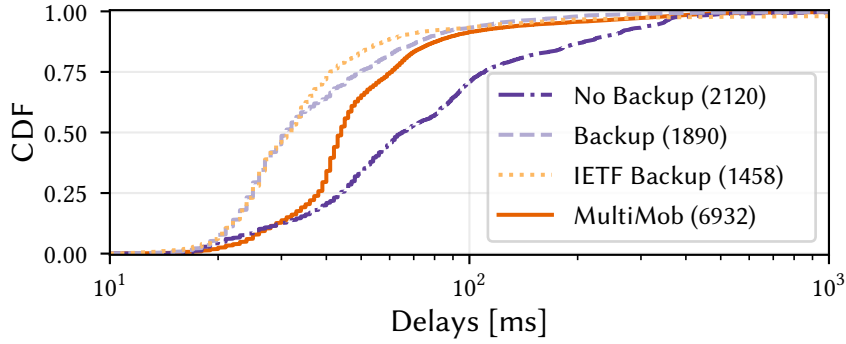
from, the MultiMob server-side scheduler then tries to reuse the Wi-Fi path to send new data, but it was meanwhile lost. After experiencing a RTO, the server finally reinjects these new data packets on the cellular subflow and the connection continues. In its case, the Backup smartphone radio experienced a 3 s stall at time 205 s (second **C** to **D** pass). This stall is actually caused by the `default` packet scheduler always favoring the primary Wi-Fi subflow over the cellular one. Indeed, around 200 s, the Wi-Fi was underperforming, so the server experienced RTOs and reinjected data over the cellular path. However, some Wi-Fi retransmissions eventually got acknowledged, causing the server to reuse the Wi-Fi path. These packet losses increased the server estimations of both the smoothed RTT and the variability of the Wi-Fi path, hence rising the value of its RTO. When the smartphone eventually went out of Wi-Fi reachability, it took seconds for the server to fire its retransmission timer. In comparison, when the MultiMob server-side packet scheduler received the acknowledgment for the packet retransmitted on the Wi-Fi path, it figured out that no original data was acknowledged. It therefore did not consider the Wi-Fi subflow to transmit the next packet. Again, the Backup configuration opened the cellular path at the beginning of the connection, while MultiMob did it at 45 s. This induces a smaller energy consumption by MultiMob with 319 J (994 mW), though the Backup one remains close with 347 J (1083 mW).

3.5.2 Measurements with Real Users

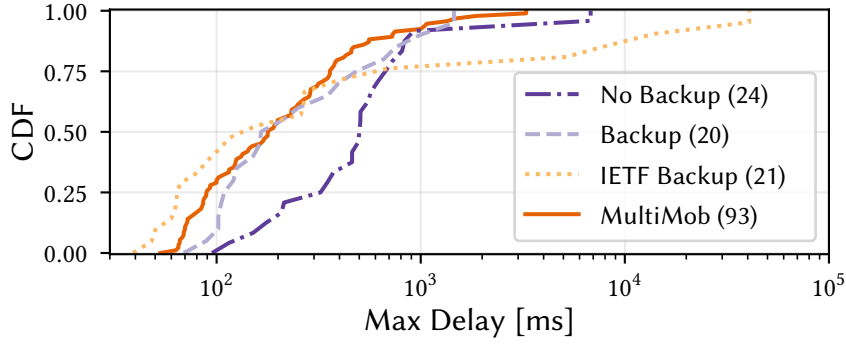
We now summarize active measurements performed on Nexus 5 Android devices distributed to a few students and academics over a period of seven weeks (from 28th January to 22nd March 2017). We installed on each smartphone an application that periodically changes the network configuration, either once during night or after a device reboot. Our measurement application monitors the smartphone accelerometer. Once it detects user motion, it starts sending data following the simplified interactive request pattern described in Section 3.2 during 80 s. The test network conditions depend on the presence of Wi-Fi and/or LTE networks. To observe the performance of all Multipath TCP configurations to switch from the Wi-Fi connectivity to the cellular one, we only consider here tests where both Wi-Fi and LTE interfaces are online at the beginning of the measurements. Notice that the Wi-Fi may be lost during some tests, though it is not always the case.

Figure 3.25 shows our in-the-wild measurements with the simplified interactive request traffic. First, Figure 3.25a reveals that nearly all the requests are answered within one second. Notice that with the No Backup configuration, 29% of the requests require 100 ms or more to be answered. This shows that the simultaneous usage of both networks with such traffic can lead to increased perceived delays because of the network heterogeneity between paths. Next, we focus on the maximal application perceived delays over a run shown in Figure 3.25b. We confirm that waiting for the Wi-Fi interface being torn down by the OS before using the cellular one, as done by the IETF Backup configuration, is not suitable for mobile situations. For all the remaining configurations, the largest application perceived latency often remains within one second, with some outliers of a few seconds. This shows that MultiMob does not impact too much interactive applications by delaying the creation of cellular subflows.

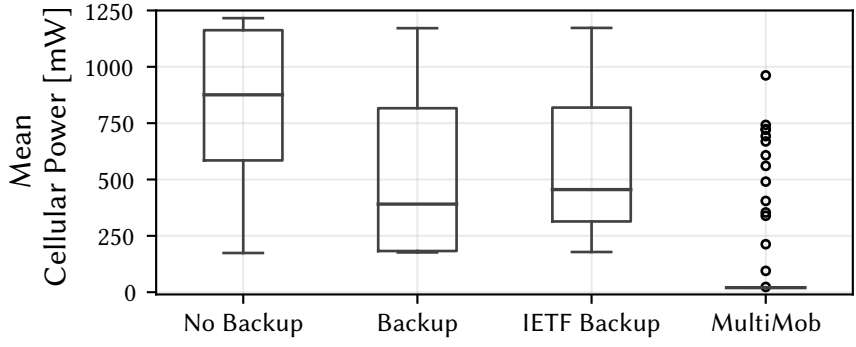
The main difference between MultiMob and vanilla Multipath TCP configurations appears in scenarios where the Wi-Fi connectivity remains alive during the entire test. Figure 3.25c presents the estimated cellular energy consumption based on the test network trace collected on the clients. Since vanilla Multipath TCP configurations always open additional subflows at the beginning of the connection, they consume cellular energy by sending Syn and Fin control packets, even if no actual data is sent on that interface. Background connections initiated by our real users can sometimes increase the energy consumption by using the cellular network. In contrast, since most of the time MultiMob does not create additional subflows, its cellular energy consumption remains very low. MultiMob therefore achieves a reasonable trade-off between low latency for delay-sensitive applications and limited energy impact.



(a) All request delays. The legend indicates the number of requests considered for each configuration.



(b) Maximal application perceived delays. The legend shows the number of tests for each configuration.



(c) Estimated mean cellular power consumption during tests where the primary path stays alive.

Figure 3.25: Simplified interactive traffic with real users.

3.6 Limitations

Both our emulation and real network experiments show that MultiMob works in the smartphone use case. However, it is important to keep things in perspective and highlight some weaknesses of our solution.

Multipath TCP remains hard to implement and deploy at a large scale. Besides the iOS one, the main Multipath TCP implementation resides in the Linux kernel. Such integration enables any TCP-based application, without requiring any modification, to benefit from the multiple network accesses the device has. At the time of writing, despite being initiated ten years ago, Multipath TCP has not been integrated in the mainstream Linux kernel yet, although this upstream effort is now close to succeed. Its acceptance has been hindered by the initial implementation design requiring lot of multipath awareness in the plain TCP workflow. Meanwhile, the mainstream kernel is moving on with new features, while a Multipath TCP version is specific to a given kernel one. This requires additional effort to port Multipath TCP to a specific device-required kernel version.

Limited set of real users. As suggested by the legend of Figure 3.25b, the dataset generated by our half-dozen of users is quite small. Actually, this is a consequence of the previously mentioned point. Smartphone vendors often fork a particular version of the Linux kernel and then apply specific code patches for their devices. This implies that the Linux version run by a smartphone depends on both the device and the Android version. In our case, we based the smartphone code on a backport of Multipath TCP v0.89 (based on Linux 3.14) to the Nexus 5 specific kernel based on Linux 3.4 realized by Grégory Detal. Except a few flagship devices [Bon18], it is unlikely that smartphone vendors will officially support non-mainstream Linux kernel features on a large panel of devices. Our potential user base is therefore limited to our particular device (the Nexus 5 device released in 2013) and to their willingness to install our custom Android ROM.

MultiMob **hacks Multipath TCP for a specific use case.** In this Chapter, we made the assumption that all users want to limit their cellular usage while keeping low latency for interactive traffic. In some countries, operators often sell LTE data quotas for a period and surcharge users if they exceed their limit. However, there are people who pay little attention to their cellular consumption and just want to always get the largest bandwidth and/or the lowest latency. For these users, the classical `full-mesh` path manager and either the `default` packet scheduler without backup interface or the `redundant` one [Fro+16] would better suit their needs. MultiMob proposes a trade-off

between low latency and limited cellular usage. If it does not fit the user's expectations, the client device can still fallback to the default `full-mesh` path manager. Yet, with the Linux implementation, it remains hard for a given Multipath TCP server to select the most suitable packet scheduler on-the-fly, especially if the client has no way to express its requirements. Similarly, MultiMob adds a few protocol mechanisms (the fast join and the idle bit) which are not standardized. These would not work if the peer do not support them. A recent work propose to extend the behavior of a Multipath TCP Linux stack with eBPF code [Tra19] but only a few well-known parts of the Multipath TCP stack can be currently tuned.

MultiMob **is constrained by TCP itself.** The Multipath TCP control plane relies on TCP options. However, this is a scarce resource. Because of the 4-bit encoding of the Data Offset field in the TCP header, there remains only 40 bytes left for TCP options. As a consequence, the fields of the fast join options were tweaked to fit in the byte limit and the Fast Join Out option in the Syn packet does not support 8 bytes DSN. All these engineering efforts inhibit innovation at transport protocol level. Another limitation comes from the Multipath TCP design goals to avoid deployment issues. Multipath TCP subflows must behave like regular TCP flows. While this seems legitimate, it induces sub-optimal strategies. Consider the case where a smartphone sends a full congestion window of data on a Wi-Fi network, and all these packets are lost. The smartphone then reinjects all these data on the cellular connectivity which get acknowledged. The transfer hence continues on the cellular. At some point, the smartphone notices that the Wi-Fi is functional again and wants to reuse it. Nonetheless, it has first to retransmit all the previously lost packets on the Wi-Fi network, even if their data was finally transmitted over the other subflow, before starting sending useful new data. By honoring their TCP requirements, Multipath TCP subflows face overhead which may impact applications. An alternative would be to reset such retransmitting sublows and re-establish a new subflow on the same network directly. Yet, this approach is not the cleanest one and still induces additional control traffic.

3.7 Conclusion

Given that smartphones have both cellular and Wi-Fi interfaces, user expect them to be able to perform seamless handovers between wireless networks. Multipath TCP enables such seamless handovers since it can use both cellular and Wi-Fi networks for a single connection. Yet, using both interfaces simultaneously is very expensive from both monetary and energy viewpoints. After having characterized how an interactive application looks like, we pro-

posed, implemented and evaluated MultiMob, a set of improvements to the Multipath TCP Linux implementation and protocol. MultiMob uses *break-before-make* to minimize energy consumption and includes an oracle monitoring the status of ongoing connections. It extends the Multipath TCP protocol to indicate idle transfers and support immediate retransmissions over a different network path. Furthermore, thanks to its server-side packet scheduler, a server automatically selects the best performing subflow to respond to requests from a smartphone. We performed both emulation and real network experiments showing that MultiMob achieves a lower cellular energy consumption while keeping good latency performance on smartphones.

Despite achieving its trade-off goal, we also highlighted the limitations of our MultiMob work. They mainly come from both the Multipath TCP kernel-space implementation and the constraints of TCP itself. Therefore, one could ask if TCP is still suitable for the modern Internet. In the coming Chapters, we explore a newer, more flexible alternative than TCP: QUIC.

QUIC 4

While TCP is the dominant Internet transport protocol, applications do not always require reliable delivery. For instance, real-time services often set a transmission deadline where overdue data is ignored by the application. If the packet arrives too late, it is somewhat equivalent to a non-delivery or a packet loss. To address such needs, the User Datagram Protocol (UDP) [RFC768] is a transport protocol providing packet-based, connectionless service. It does not ensure in-order nor actual data delivery but prevents corrupted data from being delivered to the application. Figure 4.1 illustrates the UDP packet header. Compared to the TCP one (Figure 1.3), it is much simpler as it only contains the ports (multiplexing different applications running on a given host), the length of the payload (up to 65535 bytes) and the checksum. Hence, UDP is a minimal-overhead transport protocol for applications that do not require TCP services.

Innovation in the network and transport layers of the TCP/IP protocol suite has been rather slow during the last decades. It took roughly twenty years to design, implement and deploy IPv6 [Dha+12]. TCP continues to be incrementally improved [RFC7414] with recent extensions including Multipath TCP [RFC6824] and TCPCrypt [Bit+10], despite the proliferation of middleboxes slowing their deployment [Hon+11]. New transport protocols such as SCTP [RFC4960] generated interest [Bud+12], but are still not widely deployed on the Internet.

The remaining of this Chapter introduces QUIC [QUIC-T; Lan+17], a new transport protocol built atop UDP aiming at replacing TCP while addressing large deployment concerns. In particular, its design objectives are the following.

- Built-in secured exchanges with both authenticated and encrypted

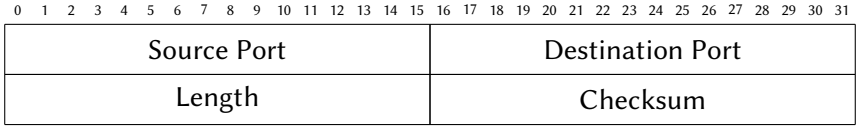


Figure 4.1: The UDP header [RFC768].

header and payload;

- Reliable, in-order data stream multiplexing to support multiple distinct transfers over a single connection;
- Low-latency connection establishment;
- Connection migration and resilience to NAT rebinding.

The built-in encryption enables QUIC to address both privacy [RFC7258] and ossification issues that middleboxes cause on transport protocols such as TCP [Hon+11]. QUIC addresses the need of application protocols such as the Hypertext Transfer Protocol (HTTP) [RFC7540] to efficiently exchange different objects (text, pictures, videos,...) over a single connection. For the web in particular, reducing the latency is key for user experience [Fla+13]. A fast QUIC connection establishment contributes to this latency reduction compared to the 2 or 3-RTT secure TCP/TLS connection establishment. QUIC is a connection-oriented protocol above UDP which does not rely on the classical 4-tuple (IP_{src}, IP_{dst}, $port_{src}$, $post_{dst}$) during connectivity failures events such as NAT rebinding.

Google first proposed in 2016 an initial version of QUIC [gQUIC] embedded in its Chrome browser. We refer to this particular version as gQUIC. Langley et al. [Lan+17] reported that gQUIC represented more than 30% of the egress Google traffic. Its features attracted so much interest that the IETF chartered a working group to standardize it [QUIC-T] — refered as iQUIC. As of January 2020, there are about twenty implementations taking part in its process. While basic principles are similar, iQUIC is now quite different from gQUIC.

In the remaining of this Chapter, we focus on the design of iQUIC. Before diving into its operations, we first introduce the main QUIC building blocks: packets and frames (§4.1). Then, we describe the lifetime of a QUIC connection, starting from its establishment (§4.2), continuing to the data exchange (§4.3), discussing the handling of possible network changes (§4.4) and finishing by its closure (§4.5). We conclude this Chapter by discussing notable differences between iQUIC design and the original gQUIC to fully understand this thesis work (§4.6).

4.1 Container Packets and Core Message Frames

From the network viewpoint, a QUIC packet is composed of two parts as exemplified in Figure 4.2. First, the cleartext header contains a few flags and the Destination Connection ID. This identifier enables the receiving host to map the packet to the connection it belongs to. This makes the packet, and

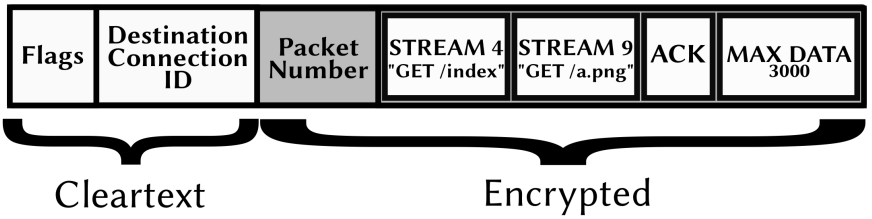


Figure 4.2: An example of Short Header QUIC packet.

Frame Type	Type-dependent Fields...

Figure 4.3: Format of a QUIC frame. The Frame Type field is a variable-length integer.

hence the connection, independent of the 4-tuple used to carry it. The second part is a large encrypted payload divided into several parts. Its first element is the packet number. Each host maintains distinct packet number spaces for sending and receiving data. These sequence numbers are monotonically increasing. Within one flow direction, *all* packets have different packet numbers. In cases where data needs to be retransmitted, it is put in another packet with a higher number than the original one. This simplifies several transport functions by removing the ambiguity of having multiple retransmitted packets with the same packet number. These ambiguities affect round-trip-time estimation and loss recovery in TCP.

The remaining of the encrypted payload consists in a sequence of *frames*. They carry data and control information while QUIC packets act as their container. In other words, when a packet is lost, frames are the messages that are retransmitted, if needed. On the wire, frames follow the Type-Value pattern as illustrated in Figure 4.3. This flexible format combined to encryption makes it easy to define new frames while mitigating deployment problems due to network ossification. Notice that the Frame Type field is encoded as a variable-length integer called *varint*. Such encoding uses the two most significant bits to indicate the length of the field ranging from 1 to 8 bytes. Currently, frames always format their Frame Type using one byte. Yet, this potentially allows QUIC to define up to 2^{62} different frames.

Currently, there are about twenty different frames covering about thirty values — some of them reserving a range acting as flags for their operations. Figure 4.2 shows the most common ones. QUIC supports several data streams like the Stream Control Transmission Protocol (SCTP) [RFC4960]. The STREAM frame contains the Stream ID (identifying the data flow), a Byte Offset and the payload associated to that stream. In Figure 4.2, the

Frame Type	Largest Acknowledged
ACK Delay	ACK Range Count
First ACK Range	ACK Ranges (# ACK Range Count)

Figure 4.4: Format of an ACK frame. All its fields are variable-length integers, the visual size of fields is for reader's convenience.

packet carries two STREAM frames belonging to different data streams with their own identifier. The ACK frame is a kind of generalization of TCP and SCTP's SACK blocks. As illustrated in Figure 4.4, ACK frames acknowledge a first block of packets whose numbers are contained in the range [Largest Acknowledged − First ACK Range; Largest Acknowledged]. Additional ACK Range Count ranges can be contained in the ACK Ranges field. The number of blocks an ACK frame can contain is only limited by the maximum size of a QUIC packet, while TCP's SACK option cannot exceed the 40-byte limit of TCP options. Notice that the ACK frame also includes an ACK delay field enabling the peer to accurately estimate the path latency, even if the host delays the sending of ACK frames. Our illustrating packet also includes a MAX DATA frame advertising the receive window of the sender. It indicates the amount of data that can be sent over the entire connection. The control of the receive buffer can also be performed at stream-level using the MAX STREAM DATA frame including the related Stream ID. Note that those MAX DATA and MAX STREAM DATA frames only appear in some QUIC packets, unlike TCP that embeds the Window field in TCP headers.

All the current QUIC frames — including the ones previously presented — are idempotent. Receiving a valid frame more than once does not introduce any ambiguity at the receiver's side. The Byte Offset field of the STREAM frame is absolute and does not suffer from the wrap-around ambiguity that affects TCP's sequence number. Similarly, since the packet numbers are monotonically increasing, ACK frames explicitly acknowledge specific packets, enabling the sender to figure out possible spurious frame retransmissions. Both the MAX DATA and MAX STREAM DATA frames advertise the absolute amount of data that could be sent, the receiver considering the highest observed value. Such absolute offset prevents the uncertainty present in TCP when several packets with the same acknowledgment number advertise different relative Window values. While QUIC does not enforce idempotent frames, future extensions should ensure this property.

4.2 Establishing a QUIC Connection

As of January 2020, the QUIC protocol standardization is not yet finished. Nonetheless, this process benefits from deployment experiences that contribute to its specification. This is possible thanks to the built-in QUIC version negotiation that takes place prior to any exchange between two peers. Such process enabled Google to update its gQUIC stack bundled in its Chrome browser every few months [Rüt+18]. It is also useful for the standardization process where iQUIC implementations indicate which version of the specification they follow. With distinct version numbers, the QUIC protocol can be very diverse: changes in the encryption and authentication schemes, different formats or even type values for a given frame, new messages,...

To ensure that implementations can negotiate current and future versions, the QUIC specification defines protocol invariants [QUIC-INV] that must be respected by all its versions. In practice, the first packets of a QUIC exchange use a Long Header containing a 4-byte Version field. If the server supports the proposed value, the communication goes on as explained next. Otherwise, it replies with a Version Negotiation packet listing all its supported versions. With this mechanism, the client is aware of the server capabilities and can then select one mutually supported version. While this process adds one RTT for the connection establishment, it is not the common case. When the client knows the server in advance, it is very likely that the upcoming connections will use the previously negotiated version, hence saving a RTT.

To encrypt its exchange, QUIC relies on the Transport Layer Security Version 1.3 (TLS 1.3) [RFC8446]. When a client contacts a server for the first time, it sends an Initial packet containing the Client Hello (CHLO) TLS message in a dedicated CRYPTO frame. The server similarly replies by transmitting a CRYPTO frame containing the Server Hello (SHLO) TLS message. Both CHLO and SHLO contribute to the generation of the encryption key being used by the connection. After having exchanged these TLS messages (one round-trip-time), the QUIC peers start transmitting data. During this first connection, peers can also negotiate pre-shared keys (PSKs) for future exchanges. With them, a client initiating subsequent connections can already encrypt its Initial packet with a PSK, allowing peers to directly start exchanging data without waiting for a round-trip-time.

In addition to the encryption, hosts negotiate during the connection establishment the Connection IDs (CIDs) that are used in the clear-text header of QUIC packets. Each QUIC host advertises the asymmetric, unidirectional CID that its peer must use to reach it, independently of the 4-tuple (IP $_{src}$, IP $_{dst}$, port$_{src}$, post$_{dst}$). In other words, the client (resp. the server) sends packets to its peer by setting in the Destination Connection ID field the CID chosen by the server (resp. the client). Such recipient-oriented routing enables network

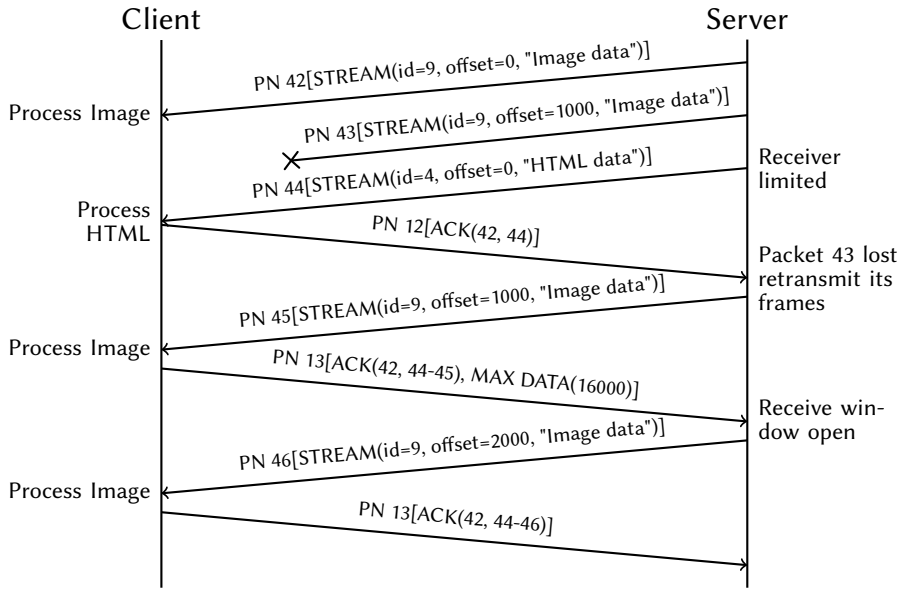


Figure 4.5: Illustrating the benefits of QUIC stream multiplexing. Only the packet number (PN) and the relevant content of frames are shown for reader's convenience.

engineers to perform load-balancing between their servers [Iye18].

During this handshake, hosts may want to advertise some internal QUIC values to their peer, such as their initial receive window. This is possible through the QUIC transport parameters carried as a TLS extension in both CHLO and SHLO messages. These parameters can be used to specify, e.g., how the ACK delay in the ACK frame should be decoded or disable mechanisms like connection migration.

4.3 Exchanging Data

Once the handshake completed, data can be exchanged over a QUIC connection using Short Header packets. To understand its operations, consider again the client packet illustrated in Figure 4.2. It requests two objects — an HTML page of 1000 bytes on stream 4 and an image of 3000 bytes on stream 9 — and advertises a connection-wide receive window of 3000 bytes. Figure 4.5 illustrates a scenario of a server answering the client's request assuming QUIC packets can contain up to 1000 bytes of data. The server uses the two first packets to send the beginning of the image, while the third packet carries the HTML page. Notice that the server cannot send the last bytes of the image yet, as the client restricts the transmission of at most 3000 bytes over the connec-

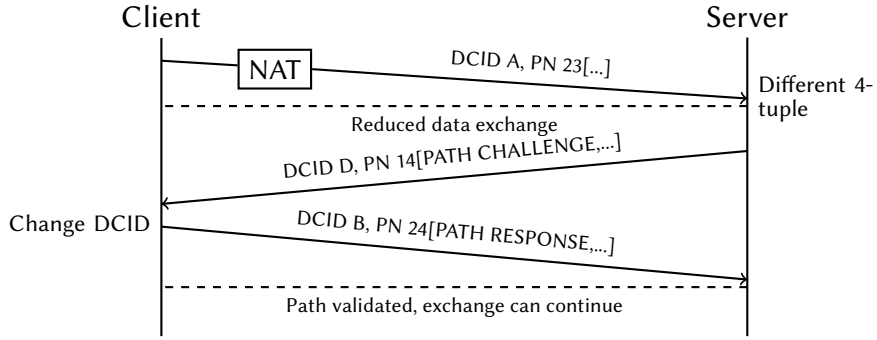


Figure 4.6: An example of NAT rebinding and how the QUIC connection migrates. Only the Destination Connection ID (DCID), the packet number (PN) and the relevant frames are illustrated.

tion. Assume now that the second packet is lost. In that case, the client sends a packet containing an ACK frame acknowledging only the first and the third ones. Upon its reception, the server can notice that the second packet was not received, and can retransmit the lost STREAM frame in another packet with a higher number. Once the missing data is received, the client decides to authorize the server to send up to 16000 bytes over the connection with the MAX DATA frame, hence allowing the transmission of 13000 additional data bytes over the connection. This receive window increase enables the server to send the last image data bytes completing the client's request.

Despite the loss of the second packet that delayed the transmission of the image file, the HTML one was directly processed by the client thanks to the QUIC stream multiplexing. In comparison, TCP only offers a single in-order byte stream over a given connection. If the image and HTML objects were interleaved over a single TCP connection, like HTTP/2 does [RFC7540], the HTML file could not have been processed before the reception of the retransmission due to the in-order TCP delivery.

4.4 Handling Network Changes with Connection Migration

Because it relies on UDP, a QUIC exchange may experience different 4-tuples during its lifetime. Such changes can be caused either by hosts themselves — like a client switching from a Wi-Fi network to a cellular one — or by the network — NAT rebinding is the typical example. Thanks to its labeling by CIDs, QUIC connections can handle these events by design. Figure 4.6 shows that a server can map an incoming packet to its connection even if a NAT modified its source IP address and port.

Frame Type	Sequence Number
Retire Prior To	Length (1 byte)
Connection ID (Length bytes)	
Stateless Reset Token (16 bytes)	

Figure 4.7: Format of a NEW CONNECTION ID frame. The visual size of fields does not reflect their actual length, annotated to each field. Non annotated fields are variable-length integers.

This connection migration ability however raises two main concerns. The first one resides in the identity of the new remote IP address and port. Despite QUIC packets are authenticated by TLS 1.3 — hence causing less problems than with TCP— a malicious client might forge a QUIC packet by faking the source IP address and port of a victim device, hoping that the server will flood it. To address this concern, consider again the case where the 4-tuple change comes from a NAT rebinding as illustrated in Figure 4.6. When the server notices a change in the 4-tuple, it reduces its sending rate and starts the *path validation* process. It consists in sending a specific frame, the PATH CHALLENGE one, containing some random data. The process completes when the host initiating the address validation receives a PATH RESPONSE frame echoing the content of the original PATH CHALLENGE one. This process ensures that, although the server perceives a different 4-tuple, it still communicates with the same client.

The second main concern is the privacy implications of moving a CID over a different 4-tuple. This could enable a passive observer to correlate the activity of the different paths. To mitigate this issue, QUIC includes the NEW CONNECTION ID frame illustrated in Figure 4.7. The sender advertises an additional Connection ID that its peer can use in its sent packets to reach it. In Figure 4.6, the server previously provided the additional CID B to the client. When noticing the reception of a PATH CHALLENGE frame, it decided to change the Destination Connection ID set in its sent packets. This behavior is not mandatory but enables concerned hosts to limit the impact of connection migration on privacy. Each Connection ID comes with a monotonically increasing Sequence Number — the one of the initial CID being 0. This label allows the frame's receiver to announce to its peer that it will not use CIDs with a Sequence Number inferior to Retire Prior To. Such signaling is also possible without communicating any CID. In that case, hosts instead use the shorter RETIRE CONNECTION ID frame.

Characteristic	iQUIC	gQUIC
Encryption	TLS 1.3 [RFC8446]	QUIC Crypto [QUIC-C]
Packet Number	Encrypted	In clear-text
Connection ID	Adaptable length, unidirectional	8-byte long, bidirectional
Connection Migration	Path validation + Connection ID change	Nothing special

Table 4.1: Summary of the main differences between iQUIC and gQUIC.

4.5 Closing the Exchange

QUIC bundles two levels of exchange: the data one with STREAM frames and the connection one. Each of these has its own termination mechanisms.

Similar to TCP, data streams can be closed either gracefully or abruptly. The graceful method uses a FIN bit carried in the STREAM frame. For the abrupt one, a sending (resp. receiving) host can reset a stream by sending a RESET STREAM (resp. STOP SENDING) frame.

Hosts can terminate a connection in three ways. First, the exchange remains idle for a period larger than a host-specific timeout — advertised to the peer using QUIC transport parameters — implicitly closing the connection. Second, a peer can explicitly request the connection termination by sending a CONNECTION CLOSE frame, optionally mentioning the reason of this event. Third, a host may also send a Stateless Reset packet when it cannot send a CONNECTION CLOSE frame for any reason. In this last case, the Stateless Reset packet carries the Stateless Reset Token — carried by NEW CONNECTION ID frames — to avoid denial of service attacks due to forged packets. Regardless of the method used, the connection closure resets all the open data streams.

4.6 Differences between iQUIC and gQUIC

This Chapter summarized the main elements of the current iQUIC protocol. Its design is mainly based on the experience of the gQUIC one. Although both provide the same services from the application viewpoint, there are a few important differences between iQUIC and gQUIC that are worth to mention. These are summarized in Table 4.1.

Encryption. One of the key motivations for QUIC is its low-latency connection establishment enabling encrypted data exchange in the first client packet. However, the design of gQUIC started in 2012 [Ros13], i.e., two years before the first draft of TLS 1.3. At that time, the available state-of-the-art encryption protocol was TLS 1.2 [RFC5246]. It requires a round-trip-time to

negotiate the encryption material for the connection, which mismatches this low-latency objective. To bridge this gap, gQUIC embeds the quite informal QUIC Crypto protocol [QUIC-C]. Lychel et al. [Lyc+15] formalized it and identified several practical attacks against the QUIC Crypto protocol. Still, it served as a prototype for the current version of TLS 1.3 [RFC8446] now bundled in iQUIC.

Packet Number. When using symmetric encryption keys, all encrypted packets must use a different nonce that needs to be communicated to the peer to decrypt the payloads. Because the packet number is monotonically increasing, it is a reasonable candidate for being the nonce. Therefore, the gQUIC packet headers contain the packet number in clear-text. However, leaving this monotonically increasing packet number unencrypted raised privacy concerns about the linkability of a connection by pervasive monitoring [RFC7258]. In addition, some actors feared that network vendors might design middleboxes "optimizing" a connection by looking at its sequence numbers [Not18]. The consensus within the QUIC WG was to keep the packet number as the nonce for the encryption of the payload. This encrypted payload is then sampled to produce a second nonce used for the encryption of the packet number. Such process enables iQUIC to leave only a few flags and the Destination Connection ID in clear-text to narrow the possible unwanted interactions with in-network agents.

Connection ID. gQUIC's 8-byte Connection IDs are randomly generated by the client at the beginning of a connection. Then, packets sent by both hosts contain the same bidirectional CID. Both the symmetric property and the fixed 8-byte length of CIDs are problematic. Because the server does not influence the client-chosen CID, it raises two main issues. First, there is a risk that two clients contacting the same server select the same CID. Assuming that the client's Connection ID selection is truly random, with the birthday paradox, when there are $\sqrt{2^{64}} = 2^{32}$ concurrent connections at the server, there is a probability of 50% that two of them share the same CID. In that case, both affected connections might be corrupted. Second, as Connection IDs are the sole indication for packet routing, their symmetry makes it difficult for network engineers to balance the load among servers. Both issues motivated the choice of unidirectional Connection IDs for iQUIC. As each host determines the CIDs for the packets it receives, there is no more identifier clash and servers can design self-balancing Connection IDs [Iye18]. In addition, the length of 8 bytes is both too short (like datacenter-chosen CIDs) and too long (for Internet-of-Thing use cases). To adapt to various situations, iQUIC supports variable host-defined Connection ID lengths.

Connection Migration. Both gQUIC and iQUIC connections can seamlessly handle 4-tuple changes. Still, gQUIC connections continue their operations without particular migration management. The issues discussed in

Section 4.4 were raised during the standardization of IQUIC. Hence, GQUIC does not verify that the new path is usable and does not use NEW CONNECTION ID frames.

Multipath gQUIC

5

One missing feature of QUIC is the ability to simultaneously exploit the different paths that exist between a client and a server. Today's mobile devices such as smartphones have several wireless interfaces and users expect to be able to combine them easily. Furthermore, a growing fraction of hosts are dual-stacked and the IPv4 and IPv6 paths between them often differ, providing different performance [Bev+13; Liv+15; LED16]. Previous Chapters demonstrate that Multipath TCP addresses this gap, both for bandwidth aggregation and network handover use cases. The current version of the QUIC protocol uses only a single UDP flow between the client and the server, preventing the simultaneous usage of several paths. While the QUIC connection migration mechanism allows moving a flow from one 4-tuple to another one, this can be seen as a form of hard handover. Experience with Multipath TCP on smartphones [BS16] and our previous Chapters show that multipath transport provides seamless network handovers.

In this Chapter, we build upon the lessons learned with Multipath TCP and first propose extensions to gQUIC to enable it to simultaneously operate over multiple paths (§5.1). We implement Multipath QUIC inside the `quic-go` open-source implementation written in Go [Cle+17] and compare its performance with Multipath TCP in a wide range of scenarios using Mininet [Han+12] (§5.2). Next, we extend this analysis to real wireless networks by including Multipath QUIC into our MultipathTester iOS application described in Section 2.3 (§5.3). After reviewing the related works (§5.4), we conclude this Chapter (§5.5).

5.1 Adding Multipath to gQUIC

As for Multipath TCP, there are two main motivations for adding multipath capabilities to a transport protocol like QUIC. The first one is to pool resources of different network paths to carry the data over a single connection [WHB08]. Such pooling is important for multi-homed devices when transferring long files, but it can also help dual-stacked hosts to automatically select the best network when the quality of the IPv4 and IPv6 paths differ. Another motivation is the resilience to connectivity failures. On dual-homed devices with wireless interfaces, such as smartphones, one network

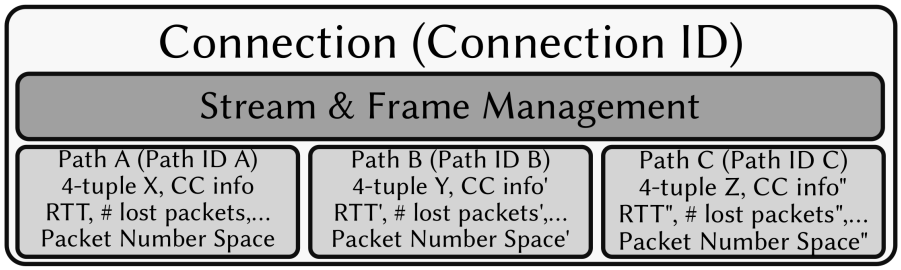


Figure 5.1: High-level architecture of Multipath gQUIC, here illustrating a connection with three paths.

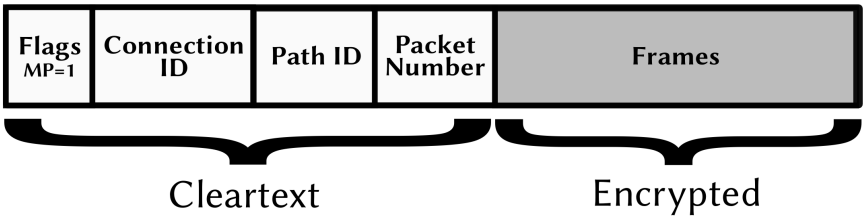


Figure 5.2: The format of a Multipath gQUIC packet.

can fail at any time and users expect that their applications will immediately switch to the other one without any visible impact [Paa+12; CSP17]. These use cases are covered by Multipath TCP [BS16; CSP17].

In this Section, we focus on the initial design of the multipath extensions for the gQUIC protocol [gQUIC]. The overall architecture of Multipath QUIC is shown in Figure 5.1. We first discuss how hosts can distinguish the usage of multiple paths (§5.1.1). We then explain how this multipath usage keeps the data transfer reliable (§5.1.2). Next, we discuss the inclusion of multipath-specific algorithms into QUIC, i.e., path management (§5.1.3), packet scheduling (§5.1.4) and congestion control (§5.1.5). Finally, we summarize the benefits of our design (§5.1.6).

5.1.1 Path Identification

Hosts need to agree on a way to identify the different paths used. A first solution would be to make those paths implicit by sending ranges of packet numbers over a particular path. However, paths can exhibit heterogeneous characteristics with very different delays. Because the packet number in the clear-text header of gQUIC is not encrypted, middleboxes in the network can see it and might decide to drop packets with smaller packet numbers than the highest seen on the connection. Such a device placed in front of a server using multiple paths might break the slowest one.

As illustrated in Figure 5.2, Multipath gQUIC takes the explicit approach by including in the clear-text header the Path ID on which it was sent. The presence of this field is determined by a previously unused bit in the Flags field of gQUIC packets. Putting the Path ID in the header also enables Multipath gQUIC to collect some metrics — such as round-trip-time or lost packets — for a given path. It also allows hosts to detect a remote address change over a specific path, e.g., due to NAT rebinding. When such events occur, hosts can update the 4-tuple associated to the path accordingly.

5.1.2 Reliable Data Transmission

To send data, QUIC uses encrypted STREAM frames. These contain a stream identifier and an absolute byte offset. This information is sufficient to enable a receiver to reorder the data contained in STREAM frames that it receives over different paths. However, the QUIC acknowledgment is per-packet based and reordering could affect packets sent on different paths due to network heterogeneity. With a single packet number space, this could lead to huge ACK frames containing many ACK blocks. To cope with this, each path maintains its own packet number space, as emphasized in Figure 5.1. By combining the Path ID and the Packet Number in the public header, Multipath gQUIC exposes the paths to middleboxes. Because packet numbers are now relative to paths, Multipath gQUIC also adds a Path ID field in the ACK frame. This enables a receiver to acknowledge QUIC packets that have been received on different paths. Our `mp-quic` implementation returns the ACK frame for a given path on the path where the data was received to avoid affecting the latency measurements of a given path. However, since it contains the Path ID, it is possible to send ACK frames over other paths.

Notice that reusing the same sequence number over different paths might have a detrimental impact on security, as the cryptographic nonce would be reused. To mitigate this issue, two approaches are applicable. The naive one consists in restricting each sequence number to be used only once over all paths. Nevertheless, this solution might complicate implementations as the packet number spaces require some coordination to prevent reusing the same value over different paths. Instead, our approach resides in involving the Path ID with the packet number in the nonce computation. This ensures that it is not possible to get the same nonce twice across paths. Furthermore, it increases the theoretical maximum number of packets over a QUIC connection to $2^{62} \times \text{nb}_{\text{paths}}$, where nb_{paths} is the number of paths used over the connection.

5.1.3 Path Management

A QUIC connection starts with a secure handshake. Like gQUIC, Multipath gQUIC performs the cryptographic handshake over the initial path. There

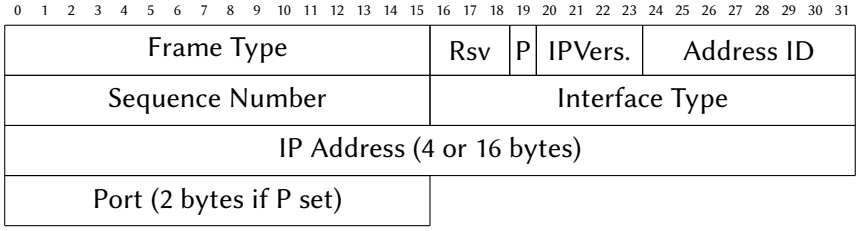


Figure 5.3: The ADD ADDR frame advertising sender's addresses.

are probably opportunities to leverage the availability of several paths for the secure handshake [Cos+18], but we leave this for future works. Similar to Multipath TCP, Multipath gQUIC uses a *path manager* that controls the creation and the deletion of paths. Our mp-quic implementation embeds a `full-mesh` path manager. Upon handshake completion, it makes the client open one path over each of its interfaces.

An important difference compared to Multipath TCP is that Multipath gQUIC can directly use a new path by placing data in the first packet. Recall from Section 1.3.2 that Multipath TCP requires a three-way handshake before being able to use any path. Since Multipath gQUIC allows both hosts to create bidirectional paths, it must ensure that the chosen identifiers do not clash. Hence, bidirectional paths created by the client (resp. the server) have an even (resp. odd) Path ID. However, our implementation does not currently use server-initiated paths because clients are often behind NATs of firewalls that would block them.

Often, a host is not aware of all the reachable addresses of its peer. To enable, e.g., a dual-stacked server to advertise its IPv6 address to the client over a connection initiated on IPv4 addresses, Multipath gQUIC includes an ADD ADDR frame shown in Figure 5.3 carrying the addresses of the sending host. This frame can be reliably exchanged at the beginning of a connection or when addresses change. Given that frames are encrypted, the ADD ADDR frame does not suffer from both the security concerns of the Add Addr option in Multipath TCP [Ear13] and its unreliable delivery. Each advertised address contains an Interface Type (indicating if the address correspond to, e.g., a Wi-Fi or a cellular network) and an Address ID. Similarly, a reliable REMOVE ADDR frame advertises the loss of a previously announced address to the peer. This frame contains the Address ID of the removed address. Notice that in order to be idempotent, those frames need to be ordered, as a very late ADD ADDR frame might arrive after a REMOVE ADDR one actually scheduled after. Therefore, both ADD ADDR and REMOVE ADDR frames contain an Address ID specific Sequence Number ordering the events concerning that Address ID. In addition, Multipath gQUIC enables hosts to get a

global view about the active paths' performance thanks to the PATHS frame. It contains the Path ID of the paths considered as active by the sending host and statistics such as the estimated path round-trip-time. It can also communicate underperforming or broken paths and thus speed up the handover process in mobility scenarios.

5.1.4 Packet Scheduling

As soon as several paths are active, a Multipath gQUIC sender needs to select over which path each packet will be transmitted. This selection is performed by the *packet scheduler*. Previous works proposed and analyzed several packets schedulers for Multipath TCP [Paa+14; OL15; Fer+16]. In Multipath gQUIC, our starting point is the `default` scheduler used by the Multipath TCP implementation in the Linux kernel [MPTCPlk]. Recall that it relies on the smoothed measured round-trip-time (RTT) and prefers the path with the lowest RTT provided that its congestion window is not already full. Multipath gQUIC uses the same heuristic, but with two major differences.

First, while Multipath TCP has to decide which data — either new or reinjected — will be sent on which path, the Multipath gQUIC scheduler also determines which control frames (ACK, MAX DATA, PATHS,...) will be included in a given packet. Since frames are independent of the packets containing them, they are not constrained to a particular path. Therefore, when a packet is marked as lost, its frames are not necessarily retransmitted over the same path. In comparison, Multipath TCP is forced to (re)transmit data in sequence over each path to cope with middleboxes. The retransmission strategy is thus more flexible in Multipath QUIC than in Multipath TCP. Notice that a sub-optimal scheduling of the MAX DATA frames might cause head-of-line blocking. To prevent such receive buffer limitations, the scheduler bundled in our `mp-quic` implementation ensures proper delivery of the MAX DATA frame by sending them on all paths when they are needed.

Second, when a new path starts in Multipath gQUIC, the host does not have an estimation of the path's RTT yet. A first solution would be to ping the new path and wait one RTT to obtain this measurement, but Multipath gQUIC would then lose its ability to directly send data on new paths. Another approach would be to use round-robin at the start of the connection and automatically switch to the lowest latency path once RTT estimations are available. However, this approach is fragile when paths exhibit very different delays and hosts could then possibly face head-of-line blocking. Our implemented scheduler duplicates frames over paths when the path's characteristics are still unknown. While this induces some overhead, it enables faster usage of the new paths without facing head-of-line issues.

5.1.5 Congestion Control

To achieve a fair distribution of the network resources, transport protocols rely on congestion control algorithms. Both single-path TCP— in the Linux kernel — and gQUIC— in our base `quic-go` and the Chromium implementations — use CUBIC [HRX08]. Using CUBIC in a multipath protocol would cause unfairness [Wis+11]. Several multipath congestion control schemes have been proposed [Wis+11; Kha+12; Pen+16]. In our Multipath gQUIC implementation, we integrate the OLIA congestion control scheme [Kha+12] which provides good performance with Multipath TCP. The adaptation and the comparison of other multipath congestion control schemes to Multipath QUIC is left for further study.

5.1.6 Summary

Overall, our Multipath extensions to gQUIC are simpler and clearner than the Multipath extensions to TCP [RFC6824], while keeping them as deployable as possible. Thanks to the clean support for multiple data flows by STREAM frames, Multipath QUIC does not need to specify a new type of sequence number, in constrast to Multipath TCP's Data Sequence Number. Multipath QUIC does not need to specify mechanisms to detect or react to middlebox interference given that all the frames are encrypted and authenticated. This also reduces the possibility of attacking a QUIC connection, compared to Multipath TCP whose security relies on keys exchanged in clear during the initial handshake [RFC6824]. Furthermore, thanks to the independence between packets and frames, Multipath QUIC can spread multiple data streams over several paths by design and the packet scheduling is potentially more powerful than Multipath TCP's one. Finally, the flexibility of QUIC allows us to easily define new types of frames to enhance the protocol.

5.2 Performance Evaluation of Multipath gQUIC

The are several approaches to evaluate the performance of a transport protocol. On the one hand, many TCP extensions have been evaluated by simulations before being deployed [FF96; Rad+11; Rai+12]. On the other hand, the gQUIC designers deploy improvements on the Google servers and use the collected statistics to tune the protocol [Lan+17]. In this Section, we rely on measurements on the Mininet emulation platform [Han+12] with complete (Multipath) gQUIC and (Multipath) TCP implementations, taking (Multipath) TCP as the baseline. We first describe our methodology based on experimental design [And12] (§5.2.1). After discussing some Mininet configuration details (§5.2.2), we then explore how transport protocols behave with bulk

	Low-BDP		High-BDP	
Factor	Min.	Max.	Min.	Max.
Capacity [Mbps]	1	50	1	50
Round-Trip-Time [ms]	0	50	0	400
Queuing Delay [ms]	0	100	0	2000
Random Loss [%]	0	2.5	0	2.5

Table 5.1: Experimental design parameters (inspired by Paasch et al. [PKB13]). Notice that we limit the path's bandwidth to 50 Mbps due to the performance of the `quic-go` implementation.

transfers, either with large files (§5.2.3) or short ones (§5.2.4). Finally, we assess how multipath transport protocols handle network handover (§5.2.5).

5.2.1 Methodology

In this Section, we consider a multipath network with two multi-homed hosts over disjoint paths with different characteristics as shown in Figure 3.13 on page 64. The performance of multipath protocols is a function of the links' bandwidth, the round-trip-times, the presence of bufferbloat (i.e., the queuing delays) and the random packet losses. Instead of focusing on a few well-chosen cases, we provide a fairer comparison of the multipath benefits by leveraging an experimental design approach [And12] covering a wide range of parameters. Multipath TCP already benefited of such evaluation methodology [PKB13]. Our experimental design selects the values of these parameters using the WSP algorithm [SCS12] over the ranges listed in Table 5.1. We group our experiments into four classes.

- **Low-BDP-no-loss**: environments with a low bandwidth-delay product and no random losses;
- **Low-BDP-losses**: environments with a low bandwidth-delay product and random losses;
- **High-BDP-no-loss**: environments with a high bandwidth-delay product and no random losses;
- **High-BDP-losses**: environments with a high bandwidth-delay product and random losses.

For each class, we consider 253 network scenarios, i.e., 253 samples in our parameter space. For each scenario, we vary the path used to start the connection, leading to 506 experiments. Each experiment is repeated 9 times for

each protocol — TCP, Multipath TCP, gQUIC, Multipath gQUIC— and we analyze the median run. We use the Linux kernel version 4.1.39 patched with Multipath TCP v0.91 as our baseline. Our Multipath gQUIC implementation, based on the `quic-go` one written in Go language, implements the design and the algorithms presented in Section 5.1. Experiments run in Virtual Machines (VMs) hosted on a server powered by a Intel Xeon E5540 @ 2.53 GHz. Each VM has two dedicated cores and 4 GB of RAM.

To ensure a fair comparison given that QUIC uses encryption consuming CPU on our emulation platform, our measurements use HTTPS over (Multipath) TCP (TLS 1.2 [RD08]) or (Multipath) gQUIC (QUIC crypto [QUIC-c]). When using the single path protocols, we use the CUBIC congestion control. Since there is no multipath variant of CUBIC, we use the OLIA congestion control scheme with multipath versions of transport protocols. The maximal receive window values are set to 16 MB for both TCP and QUIC.

5.2.2 Performing Accurate Experiments with Mininet

The Mininet emulation platform [Han+12] is a powerful tool enabling fast network experiments. However, the specification of the topology, and in particular the link characteristics, should be performed with care. Our experiments aims at representing a network where the bottleneck link has the specified bandwidth with the router introducing queuing delay due to its buffer. The remaining of the network introduces propagation delay and possibly random losses. To introduce such link alterations, we rely on the `tc` tool [Alm+99].

Nevertheless, Mininet hosts use different network name spaces within the same Linux kernel. Several researchers revealed interference leading to unrealistic results when `tc` commands are directly attached to the sending hosts [Frö17] or when requesting very small propagation delays [Leb17]. Furthermore, the API offered by Mininet to specify the link characteristics generates questionable configurations. When requesting specific bandwidth, propagation delay and buffer size, Mininet sets up both `htb` and `netem` [Hem+05] on the outgoing interface of the hosts attached to the configured link. This setup can generate unexpected results when specifying small buffer sizes. Indeed, `netem` implements the link delay by keeping packets during a period lasting `latency` in a buffer of capacity `limit`. This means that in one flow direction, e.g., server to client one, there cannot be more than `limit` packets in-flight in the network. To assess our intuition, we consider a loss-free single-path network with a 50 Mbps link exhibiting a one-way delay of 20 ms — 40 ms RTT — and set the `limit` of `netem` to only 30% of the bandwidth-delay product (75 KB with 50 packets instead of 250 KB and 167 packets). We run `iperf` [iPerf] in UDP mode to generate a constant-rate traffic. When

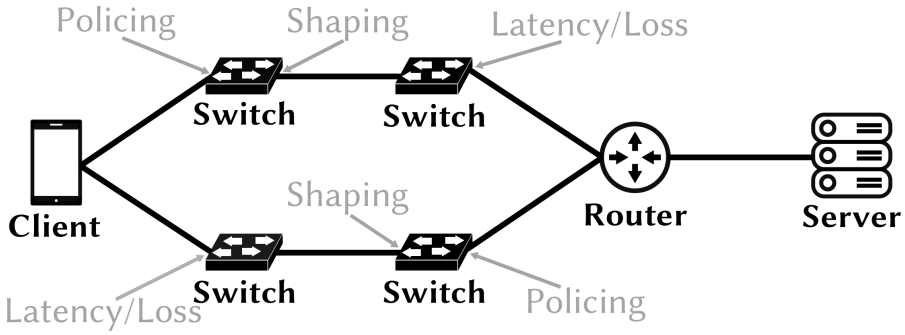


Figure 5.4: Our network topology in the Mininet platform, annotated with the traffic control operations. For clarity, we only show actions for the client to server flow (resp. the server to client flow) on the upper path (resp. the lower path).

generating a 50 Mbps traffic, the network only achieves a throughput of 28.5 Mbps. This surprising result actually matches the `netem` behavior restricting the usable bandwidth to 75KB/20ms = 3.75MB/s = 30Mbps. This finding shows that relying on the `limit` parameter of `netem` on a link to implement a router's buffer is not appropriate.

Instead of using the Mininet defaults, we set up manually our `tc` commands in our specific network topology illustrated in Figure 5.4. As suggested by Frömmgen et al. [Frö17] we do not set any command on hosts' interfaces. To implement our router's buffer, we police the traffic on the ingress interface of the first switch to implement a First-In-First-Out queue. The policing command accepts the specified link rate plus a `burst` size corresponding to the router's buffer size. The exceeding packets are dropped. We then shape the traffic at the same rate as the policing on the outgoing interface of the first switch. Finally, we set the `netem` command applying link delays and random losses on the outgoing interface of the second switch. Since the traffic was previously shaped, the `limit` parameter of the `netem` command can be set to the bandwidth-delay product, but in practice we set a much higher value to avoid interference. With our setup to reproduce the previously considered scenario, we achieve a throughput of 49 Mbps with UDP packets and observe with TCP traffic a latency variation matching the queue size. These results convince us about our proposed emulation methodology.

5.2.3 Large File Download

Our first explored traffic pattern is the download of a 20 MB file over a single stream. Here, a multi-homed host wants to minimize the download time and thus maximize the bandwidth aggregation of the available paths. The client

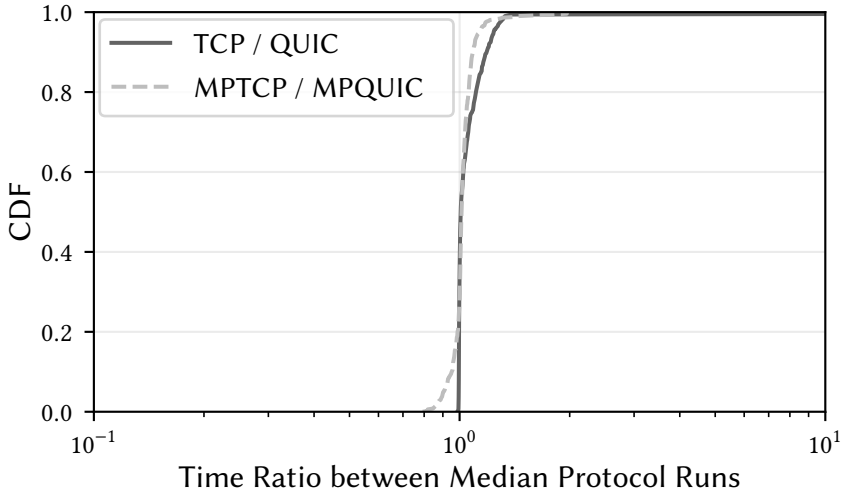


Figure 5.5: gQUIC tends to be slightly faster than TCP. On average, Multipath gQUIC and Multipath TCP exhibit similar performance.

measures the delay between the transmission of the first connection packet and the reception of the last byte of the file.

Low-BDP-no-loss: Both Protocols Achieve Similar Performance

Our first metric is the ratio between the delay to receive the file with TCP and the time required using gQUIC. If the ratio equals 1, both protocols are equivalent. If the ratio is larger than 1, TCP is slower than gQUIC. Figure 5.5 provides the Cumulative Distribution Function (CDF) of this ratio over all considered experiments with a low bandwidth-delay product and no random losses. Notice that packet losses due to router buffer overflow can still occur in these environments. When a single path is used, we do not observe much difference between TCP and gQUIC. This is expected since in this situation, the congestion control is the main influencing factor, and both use CUBIC. Notice that for some runs, there is a little advantage for gQUIC. This is because gQUIC tunes the CUBIC congestion control algorithm to emulate the gQUIC connection as two TCP ones. Such behavior was motivated by Google's YouTube use case that carries both audio and video data streams over a single connection. In practice, when a loss occurs due to router's buffer overflow, the TCP's CUBIC multiplies its congestion window by 0.7, while the gQUIC's factor is $(2 - 1 + 0.7)/2 = 0.85$.

When using multiple paths, both gQUIC and TCP achieve similar performance. This is expected as both protocols operate in bufferbloat-free net-

works and the few lost packets can be easily recovered by fast retransmissions. Since our experimental design aims at representing all possible cases within our parameter space, we expect to face quite extreme scenarios. Nevertheless, the performance gap between Multipath TCP and Multipath gQUIC in these outliers remains limited. In particular, we identify one case where Multipath TCP performs 25 % better than Multipath gQUIC and another one where Multipath gQUIC is 2× faster than Multipath TCP.

Experimental Aggregation Benefit

The completion time ratio between protocols only shows whether (Multipath) gQUIC performs better or worse than (Multipath) TCP for a given experiment. To better assess the benefits of multipath protocols over single-path ones for specific scenarios, we propose a modification of the aggregation benefit metric [Kas12; PKB13]. Instead of comparing the measured goodput with the sum of the link bandwidths, our *experimental aggregation benefit* (EABen) compares the sum of the goodputs achieved by single-path protocols over each of both links with the goodput of the multipath variant. Let C be a multipath aggregation experience with n paths. G_s^i is the mean goodput achieved by a single-path connection on path i. G_s^{max} is the maximum single-path mean goodput measured over the n paths. Given G_m the mean goodput achieved by a multipath protocol over a given experiment C, the experimental aggregation benefit EABen(C) is defined by

$$\mathsf{EABen}(C) = \begin{cases} \frac{G_m - G_s^{max}}{(\sum_{i=1}^{n} G_s^i) - G_s^{max}} & \text{if } G_m \geq G_s^{max}, \\ \frac{G_m - G_s^{max}}{G_s^{max}} & \text{otherwise.} \end{cases} \tag{5.1}$$

An EABen of 0 indicates that a multipath protocol achieves the same performance as the single-path variant over the best path. If multipath achieves a mean goodput equal to the sum of the mean goodputs over all paths, then the experimental aggregation benefit equals 1. A value of -1 for the EABen indicates that the multipath protocol failed to transfer data. Because we rely on experimental values, the EABen can be greater than 1 when the performance of multipath protocols is better than the sum of the performances of the single path variants over each path, i.e., $\mathsf{EABen} \in [-1; +\infty[$. In this Section, we compare Multipath gQUIC with single-path gQUIC and Multipath TCP with single-path TCP.

Low-BDP-no-loss: Multipath Is Beneficial to Both Protocols

Research on Multipath TCP has shown that its performance was impacted by the characteristics of the initial path [Arz+14]. We thus split the results into two categories depending on whether the connection is created on the best

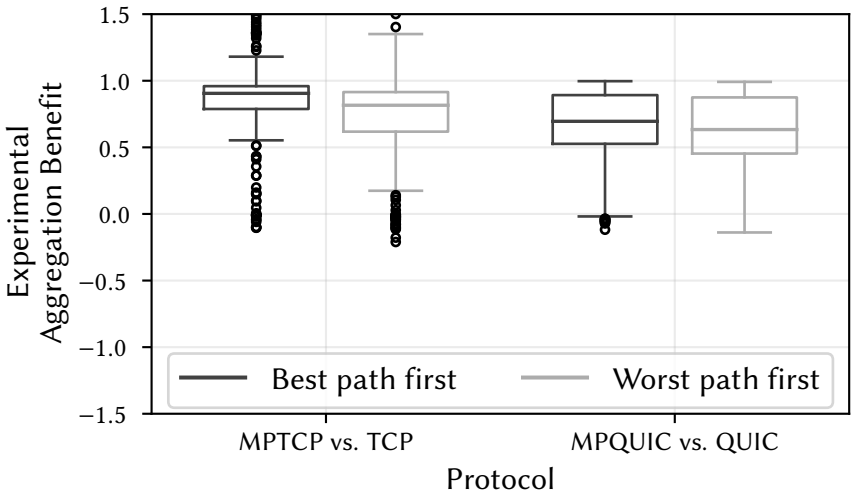


Figure 5.6: In Low-BDP scenarios without random losses, using several paths is more beneficial to gQUIC than to TCP.

or the worst performing path from an experimental viewpoint. Figure 5.6 shows the experimental aggregation benefit over the 253 Low-BDP scenarios without random losses. Our measurements show that the multipath capability benefits to both TCP and gQUIC. Indeed, independently of the initial path, both Multipath TCP and Multipath gQUIC achieve a positive experimental aggregation benefit in respectively 91% and 88.5% of our scenarios. The Multipath TCP results stay in-line with the previous findings of Paasch et al. [PKB13]. For the Multipath gQUIC ones, this is encouraging since except the duplication of the initial congestion window, there is no built-in specific optimization such as the Opportunistic Retransmission and Penalization (ORP) [Rai+12] mechanism of Multipath TCP.

Low-BDP-losses: (Multipath) gQUIC Outperforms (Multipath) TCP

Random losses — such as those that occur on wireless links — affect the performance of reliable transport protocols. We now analyze the experiments in Low-BDP environments with such losses. As mentioned in Table 5.1, we consider random losses of up to 2.5%. This range fits in the retransmission rates experienced by Google's deployment [Lan+17]. Figure 5.7 shows that in those scenarios, (Multipath) gQUIC performs better than (Multipath) TCP. (Multipath) gQUIC overall better handles losses than (Multipath) TCP thanks to its ACK frames that can acknowledge up to 256 packets number ranges. This is much larger than the 2-3 blocks than can be acknowledged with the

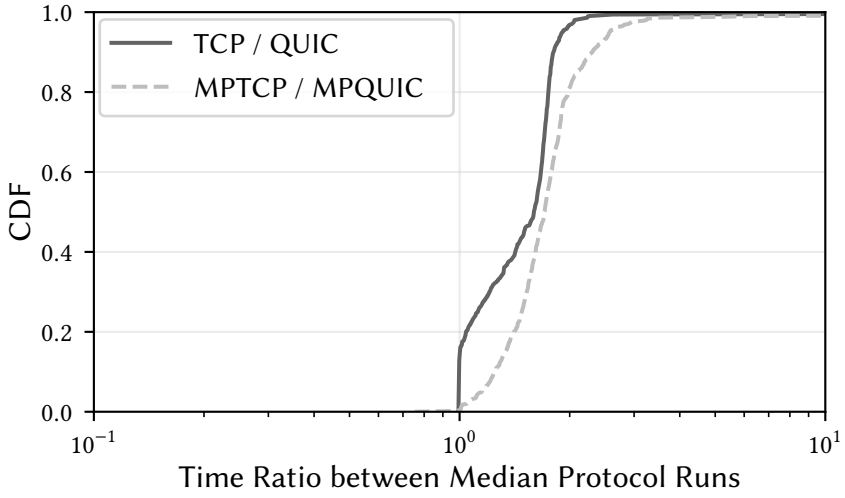


Figure 5.7: In Low-BDP scenarios, (Multipath) gQUIC reacts faster than (Multipath) TCP to random losses.

SACK TCP option depending on the space consumed by the other TCP options. Therefore, early retransmits are more effective in gQUIC and it suffers less from head-of-line blocking.

Low-BDP-losses: Multipath Is Still Beneficial to gQUIC

Figure 5.8 shows that using multiple paths can still reduce download times compared to single-path over the best path. Regardless of the initial path characteristics, Multipath gQUIC (resp. Multipath TCP) achieves better goodput than the best single-path variant in 63% (resp. 29%) of our considered scenarios. Again, the packet scheduling is much less affected by random losses given gQUIC's round-trip-time estimation. Furthermore, the content of a packet marked as lost is not automatically retransmitted by Multipath gQUIC on the same path, unlike Multipath TCP. Also, if the content is delivered over another path, Multipath gQUIC does not need to eventually deliver again the packet over the original path — addressing Multipath TCP's goal that each path behaves like a regular TCP flow. When a packet is lost, the OLIA congestion control scheme reduces the sending window over the affected path. The loss detection mechanisms can enable Multipath gQUIC to take advantage of multiple paths, even if they are lossy.

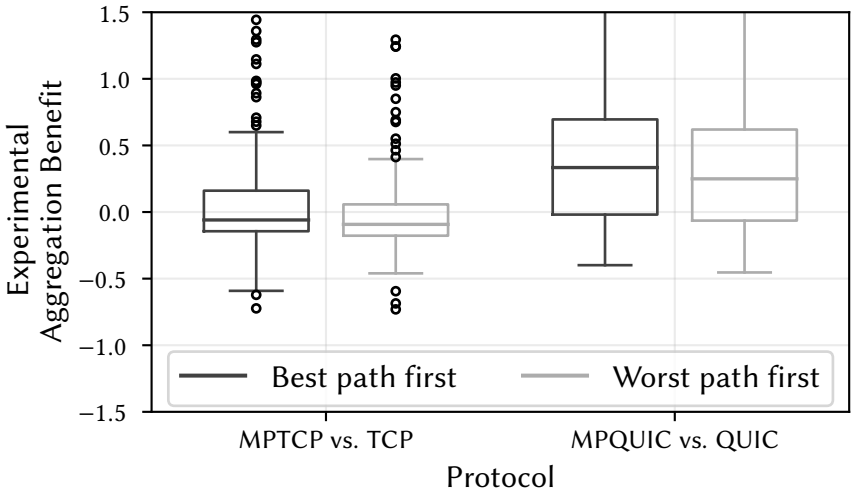


Figure 5.8: Multipath can be still advantageous for gQUIC in lossy environment, though the measured goodput varies more.

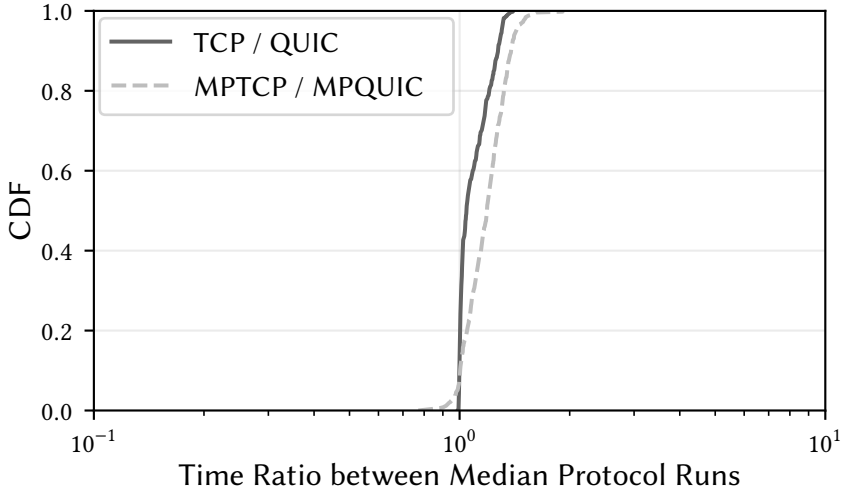


Figure 5.9: Without random losses, we observe the same trend of ratio completion time in High-BDP environments as Low-BDP ones.

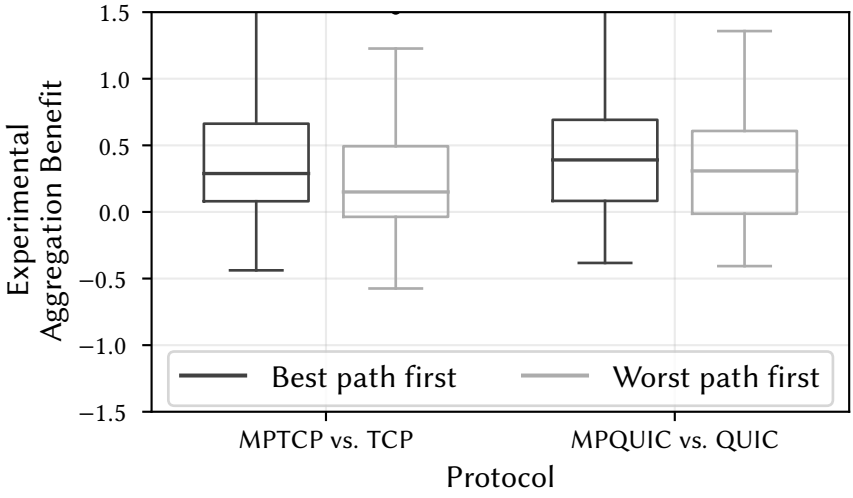


Figure 5.10: Multipath gQUIC can remain advantageous for long transfers in High-BDP environments.

High-BDP-no-loss: Multipath Protocols Still Perform Well

When the network does not exhibit random losses, Figure 5.9 confirms that the trend about download completion times in Low-BDP environments is also present in High-BDP ones. The explanation about the single-path protocol is similar. However, we observe that Multipath gQUIC outperforms Multipath TCP in 90% of the experiments, with the median run showing a ratio of 1.175. In addition, the experimental aggregation benefits of multipath protocols decrease while staying most of the time positive, as shown in Figure 5.10. When choosing the less performing path, multipath is beneficial in 73% of the scenarios with gQUIC, while this percentage reaches 67% with TCP. In such networks, Multipath TCP suffers from the capture effect of the initial subflow and most of the bytes are carried by this path. Indeed, due to the additional path establishment delay, the initial path has more time to grow its congestion window. In comparison, Multipath gQUIC starts using both paths at the same time and achieves a better load balancing across them. Because all paths can start sending data once the connection is established, congestion windows evolve more fairly. Besides the capture effect and despite the large size of the file, the impact of the connection handshake latency is sometimes visible.

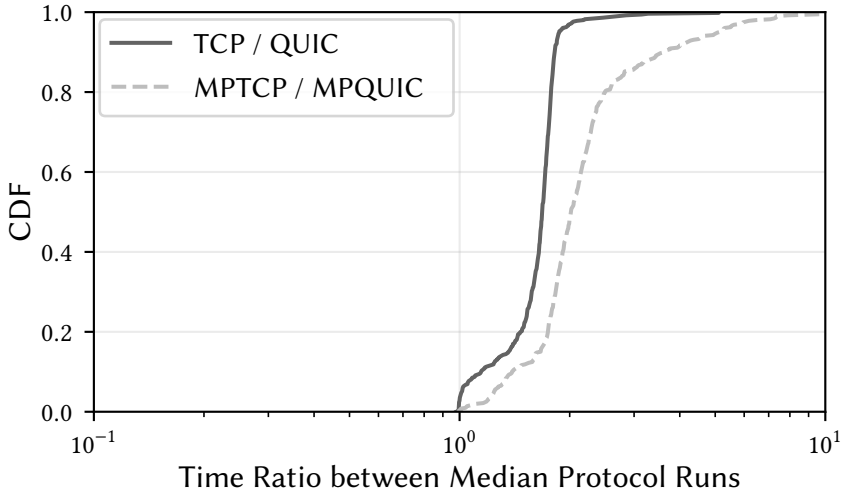


Figure 5.11: gQUIC performs better than TCP in High-BDP environments when there are random losses.

High-BDP-losses: (Multipath) gQUIC Better Copes with Losses

When adding random losses in High-BDP networks, (Multipath) gQUIC still outperforms (Multipath) TCP as shown in Figure 5.11. Based on the previous insights provided by Low-BDP-losses and High-BDP-no-loss experiments, this result is expected. The better loss signaling, more precise latency estimation, the higher scheduling flexibility and the fairness in the evolution of the paths' congestion window with (Multipath) gQUIC explain this outcome.

Despite those mechanisms, Figure 5.12 indicates that the usage of multiple paths is beneficial to gQUIC in only 41% of the studied scenarios (Multipath TCP being advantageous in 9% of the cases). In such networks, it might be relevant to explore non loss-based congestion control schemes and to take into account the loss rate in the packet scheduling.

Explaining the CoNEXT Results: Huge Buffers

The perceptive reader would note that the results presented in this Section differs from the ones presented in our CoNEXT 2017 paper. The cause of these dissimilarities comes from a network misconfiguration from our side. Link characteristics were set using `tbf` and `netem`. Instead of specifying the buffer size by setting the `limit` parameter of `netem`, we relied on the `latency` one of `tbf`. Without setting its `limit` parameter, `netem` implicitly allocates a buffer of 1000 packets. With such large values, there is no packet

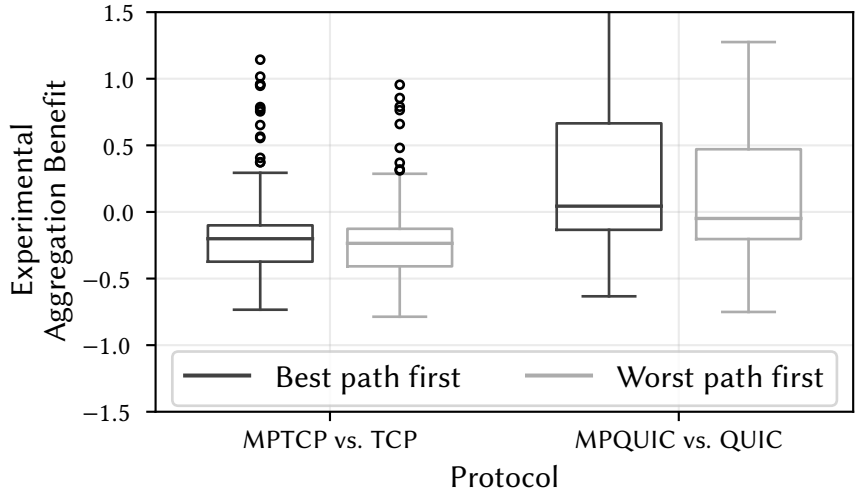


Figure 5.12: In the extreme lossy High-BDP networks, there is room for improvement for both Multipath gQUIC and Multipath TCP.

loss as the traffic is previously shaped. This makes connections suffer from severe bufferbloat.

Therefore, our previous "Low-BDP" environments actually backed huge buffers with quite small propagation delays (up to 50 ms RTT). Figure 5.13 shows that in such loss-free conditions, the performance of single-path gQUIC and TCP protocols are similar. As they both rely on the CUBIC congestion control, their sending windows increase exponentially at the same rate, achieving the same results.

However, when using multiple paths in such situations, gQUIC outperforms TCP in 86.5% of the considered experiments. To explain this difference and quantify the impact of each of the factors, we consider the median experience with a MPTCP / MPQUIC completion time ratio of 1.28. The studied scenario consists in an initial path showing a RTT of 27.6 ms, a queue size of 39 ms and a bandwidth of 13.06 Mbps, while the additional one has a RTT of 44 ms, a queue size of 98 ms and a bandwidth of 6.42 Mbps. In such a network, Multipath gQUIC completes in 9.03 s, while Multipath TCP takes 11.6 s.

This result is mainly due to the receive window increase. Except at connection beginning, we notice that the Multipath gQUIC transfer is never blocked by the receive window. In comparison, the Multipath TCP connection faces idle times taking together a few seconds due to receive buffer limitations. Especially during the beginning of the transfer, the reordering between both paths exceeds the receive window, leading to head-of-line blocking. Un-

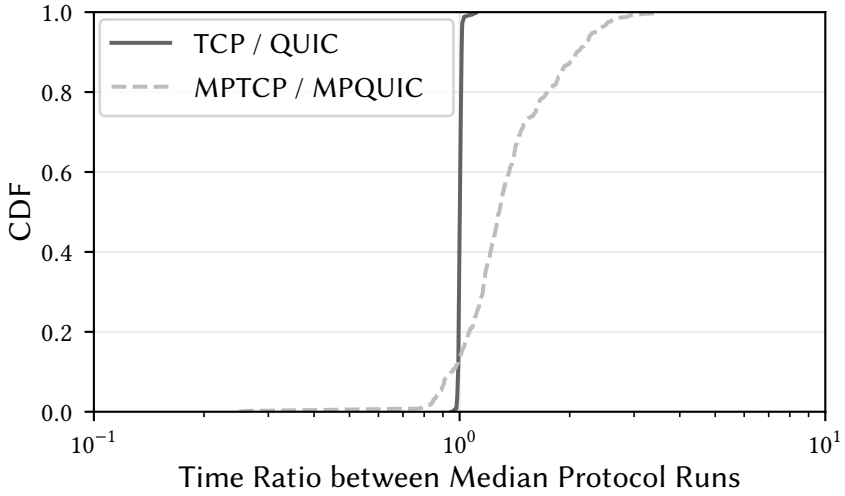


Figure 5.13: With huge buffers preventing packet losses, gQUIC and TCP exhibit similar performance. On average, Multipath gQUIC tends to be faster than Multipath TCP.

der those conditions, Multipath TCP uses the Opportunistic Retransmission and Penalization (ORP) [Rai+12] mechanism that also leads to retransmissions of packets from the slow path over the fast one, limiting the overall goodput. Indeed, the (Multipath) TCP receive buffer autotuning present in the Linux kernel relies on the estimated network RTT [Rai+12]. However, it often increases too slowly regarding the experienced reordering. In comparison, (Multipath) gQUIC updates its receive window by increments communicated by MAX DATA frames sent on all available paths. Its initial value is 32 KB. If the receiver consumes half of this increment, it sends a MAX DATA frame increasing the receive window. In addition, if two MAX DATA frames are scheduled within 2 estimated smoothed RTTs, the increment is doubled. With this strategy, in our considered scenario , Multipath gQUIC reaches a stable receive window of 2048 KB + (2048 KB + 1024 KB)/2 = 3584 KB after carrying 2.5 MB. In comparison, Multipath TCP uses a receive window of 2.3 MB after exchanging 4.6 MB. The TCP's automatic buffer tuning increases with the estimated round-trip-time, however this estimation grows too slowly compared to the actual experienced latency. To confirm our intuition in such pathological networks, we run again the experiment with Multipath TCP and force its receive window to its maximal value of 16 MB at connection initiation. In such conditions, the transfer completes in 10.4 s with a completion ratio of 1.15 compared to Multipath gQUIC. This result confirms the non-

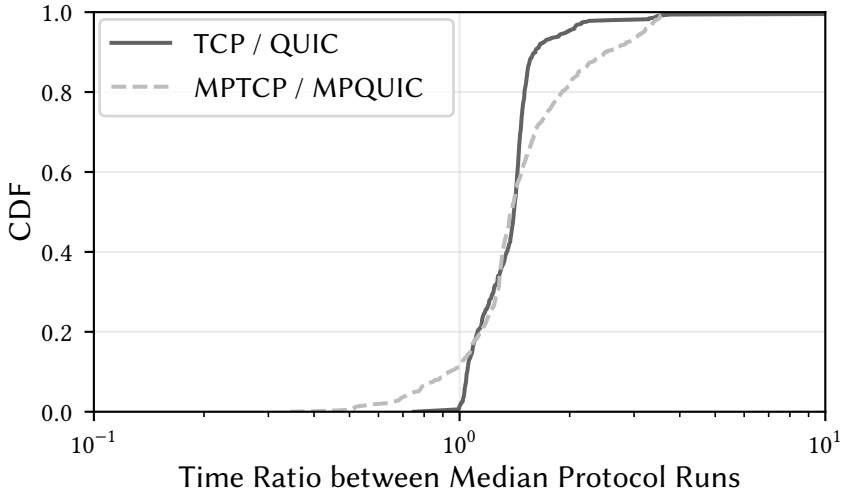


Figure 5.14: For small transfers, gQUIC is faster than TCP thanks to its shorter handshake.

negligible impact of the receive window autotuning on multipath protocols.

5.2.4 Short File Download

While large transfers are useful to illustrate the bandwidth aggregation capability of multipath transfers, they are not representative of all the possible traffic patterns on the Internet. For instance, Google reported that a single request in its search engine generates an average response of 100 KB [Lan+17]. We now focus on the download of a 256 KB file over a single stream in Low-BDP-no-loss environments.

(Multipath) gQUIC outperforms (Multipath) TCP

Figure 5.14 shows that single-path gQUIC outperforms HTTPS over TCP. For short transfers, the connection establishment is a non-negligible fraction of the total transfer time. With gQUIC, the secure handshake provided by the QUIC Crypto consumes a single round-trip-time. With TLS 1.2 with TCP, the three-way handshake combined to the TLS one consume together three round-trip-times. This delay could be reduced by using the recent TLS 1.3 [RFC8446] and the TCP Fast Open extension [Rad+11].

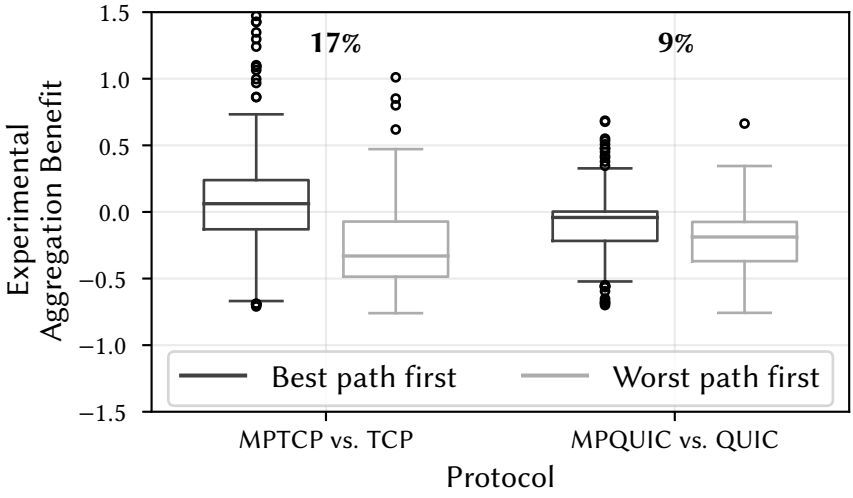


Figure 5.15: For small transfers, gQUIC should remain single-path with heterogeneous paths.

Multipath is not useful for short bulk transfers

As highlighted by Figure 5.14, Multipath gQUIC outperforms Multipath TCP in most of the situations. However, using a multipath protocol with a bandwidth aggregation objective is not really desirable for short transfers, as illustrated in Figure 5.15. When the connection is initiated on the best performing network, using multiple paths can bring some benefits, as suggested by previous results [Den+14]. However, performance mainly depends on the connection handshake latency and the data transfer might be over before taking full advantage of the multipath opportunities. In addition, Multipath gQUIC can sometimes face head-of-line blocking due to its fast usage of both paths, as it may send some last data on the less performing path. Notice that the usage of multiple paths might be relevant for short transfers when facing lossy network paths. In such cases, the scheduling algorithm should adapt to the case it performs.

5.2.5 Network Handover

We now consider a specific situation that involves a request/response traffic over a single connection and where the initial path becomes completely lossy after some time. This can correspond to a smartphone initially both connected to a bad Wi-Fi network and a good cellular one. Recall from Chapters 2 and 3 that such situation is one of the main motivations for adding Multipath

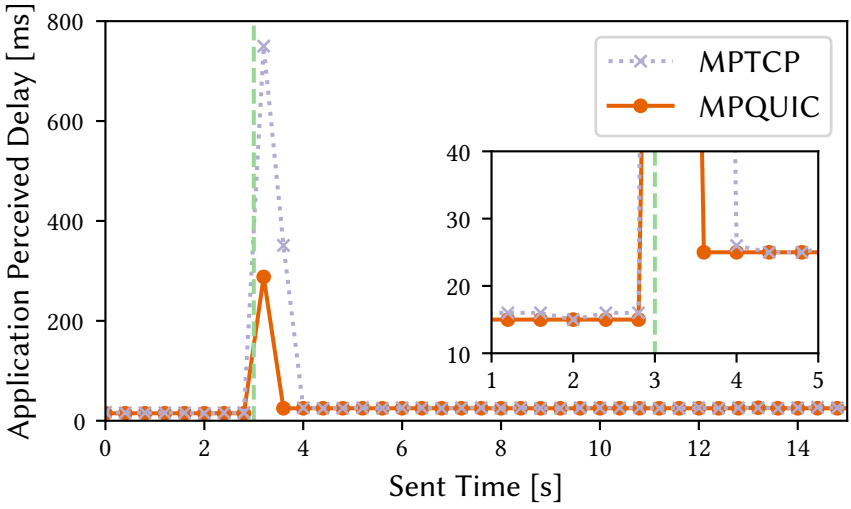


Figure 5.16: Multipath gQUIC is able to perform network handover by design while achieving lower application latency than Multipath TCP.

TCP on smartphones [CSP17]. We emulate this case by focusing on a specific network scenario where the initial path has a lower latency (15 ms RTT) than the additional one (25 ms RTT). The client sends 750-byte long requests every 400 ms and the server immediately replies to these requests with 750-byte responses. The client computes the application perceived delay between sending the request and getting the response. Initially, paths are loss-free. After 3 seconds, the initial path becomes completely lossy without notifying the OS (using `tc netem loss 100%`).

Before exploring Multipath gQUIC, let us first explain how Multipath TCP achieves network handover. Figure 5.16 shows that initially, both hosts use the initial path as it exhibits a lower latency than the other one. Since the loss event occurs after an answered request, the client is the first to notice the network failure. In practice, the Multipath TCP implementation in the Linux kernel considers a path as *potentially failed* when it experiences a RTO without observing any network activity since last packet transmission [Pin15]. The affected path remains in this state until data is acknowledged on this path. The `default` packet scheduler temporarily ignores potentially failed paths. This enables it to quickly decide to use another path when severe losses affect a particular path. Hence, after facing a RTO, the client reinjects the request over the slow but functional path. However, when the remote host receives the data that has been reinjected over the second path, it does not know that a previous copy was sent unsuccessfully over the first path. It just

follows its scheduling strategy preferring the lowest latency path, which remains the initial path from the server perspective. Therefore, it first sends the response over the initial lossy path, experiences a RTO marking the first path as a potentially failed one and then reinjects it over the additional path. This recovery time could be long for interactive applications. In our specific scenario, our client measures a maximum application delay of 750 ms when using Multipath TCP.

In comparison, Multipath gQUIC prevents the RTO at server side and achieves a maximum application delay of 288 ms. The Multipath gQUIC client still experiences an RTO when sending the first request after the initial path became lossy. However, instead of simply retransmitting the lost STREAM frame, the client host also includes in the packet a PATHS frame indicating that the initial path is potentially failed. When the server receives that frame, it can then directly mark the initial path as unusable and send back the response to the client, continuing the connection on the functional path. Notice that such approach could be applied to Multipath TCP too. Nevertheless, carrying such message requires the definition of a new clear-text Multipath TCP option which, unlike QUIC frames, is vulnerable to middlebox interference.

5.3 Real Network Comparison using MultipathTester

In this Section, we extend our performance analysis with real wireless environment. In practice, we leverage our MultipathTester iOS application presented in Section 2.3 to bundle our mp-quic implementation and compare Multipath gQUIC with the iOS one of Multipath TCP. We first describe the adaptation we made to our application to include Multipath gQUIC (§5.3.1). We then investigate the performance of Multipath gQUIC under stable network runs (§5.3.2). Finally, we compare the behavior of Multipath gQUIC with iOS Multipath TCP when running our mobile experiments (§5.3.3).

5.3.1 Adapting MultipathTester

Our MultipathTester application integrates supports for both Multipath gQUIC and Multipath TCP from day one. Hence, all the design elements presented in Section 2.3.1 still apply. We only describe here the specific aspects of Multipath gQUIC.

Streaming Traffic Pattern. Considering the *streaming* traffic pattern presented in §2.3.1.1, its implementation differs between Multipath TCP and Multipath gQUIC. Recall that with Multipath TCP, two independent connections are used to prevent head-of-line blocking when a lost response blocks

the delivery of the next request. As (Multipath) QUIC supports stream multiplexing, we employ a single connection that carries two independent data streams.

Mobile Mode. When launching the mobile mode, we simultaneously evaluate both Multipath TCP and Multipath QUIC, meaning that they compete for the network interfaces. However, due to the low traffic volume generated by the *streaming* traffic pattern — 40 KB/s uplink and 40 KB/s downlink — we believe that the impact on the observed delays should be negligible. Furthermore, running them simultaneously enables us to evaluate protocols using the same network conditions due to device mobility.

Collecting Multipath QUIC Internal States. Our Multipath gQUIC implementation itself logs its internal variables in a file using a dedicated thread. Its content, updated every 100 ms, includes path congestion window and smoothed estimated round-trip-time. These data are then sent to the collect server.

Multipath QUIC Infrastructure. On both smartphone and test servers, we use our `mp-quic` implementation to serve (Multipath) QUIC. Recall that this version is based on an old gQUIC version using a different network format than iQUIC. However, except for the gQUIC connectivity, we do not expect much difference with iQUIC in terms of (multipath) performances.

Multipath QUIC Path Management and Packet Scheduling. To avoid being unfair with regard to Multipath TCP, we configure the Multipath QUIC scheduler such as it also advertises all cellular paths as backup ones. To do so, we take a free bit in the ADD ADDR frame to indicate if the advertised address — along with its Address ID — is a backup one or not. The peer is then aware that a path is backup one, either if the remote source address is marked as such or if a received PATHS frame indicates that the path uses a backup Address ID. This prevents QUIC from using the cellular path directly. If the smartphone notices RTO on the Wi-Fi path or some data being in-flight for more than 600 ms, it starts using the cellular path.

5.3.2 Stable Network Runs

In this section, we briefly describe some interesting results obtained during stable network tests. We first provide single-path findings and then expand on multipath ones.

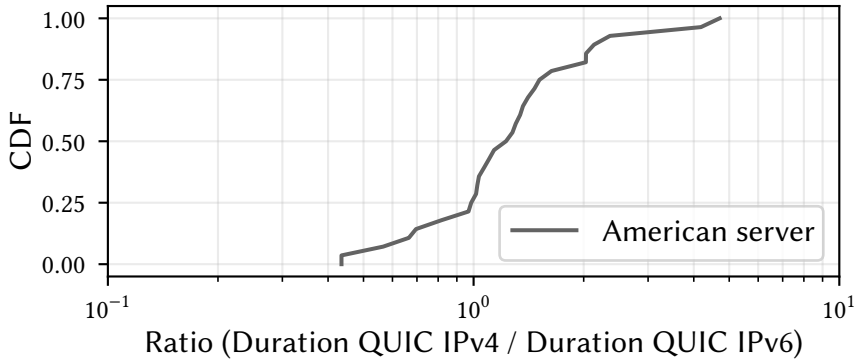


Figure 5.17: In America, we observe that IPv6 offers better results than IPv4 under our bulk traffic pattern, probably due to NAT64.

IPv6 Connectivity. Proportionally, the American server observes the largest proportion of IPv6 compatible hosts, with 65% of smartphones having an IPv6 address. In comparison, the European one only observes 43% of the devices with IPv6 addresses, and on the Asian one, this number drops to 29%. However, we also observe that having an IPv6 address does not guarantee QUIC connectivity using IPv6. On the European server, if we select the devices having both IPv4 and IPv6 addresses, we observe a gQUIC connectivity success rate of 89% using IPv4, but this rate decreases to 58% over IPv6. When digging into the tests where the IPv4 gQUIC handshake succeeded but not the IPv6 one, we notice two kinds of equally balanced errors. The first one is simply the IPv6 gQUIC handshake that timeouts. Most of the times, this happens when IPv6 is provided by the Wi-Fi network. Indeed, this issue arose in 12 different Wi-Fi networks, while only 2 distinct cellular ones suffered from such timeouts. The second error cause is the gQUIC client that encounters a "no route to host" error while trying to send a packet to an IPv6 address. This typically occurs when the smartphone selects the Wi-Fi network as its default interface while it only provides IPv4 connectivity, even though an IPv6 address is allocated on the cellular interface. Such routing issue is probably due to a bad interaction between iOS and the gQUIC implementation.

Performance of QUIC Using IPv4 vs. IPv6. When QUIC is usable over both IPv4 and IPv6, we do not observe much difference in terms of performance using the different traffic patterns on the European server. However, on the American one, we see better results with IPv6 than with IPv4. For instance, Figure 5.17 provides the ratio of the download completion times of a 10MB file between gQUIC IPv4 and gQUIC IPv6. In 75% of the measure-

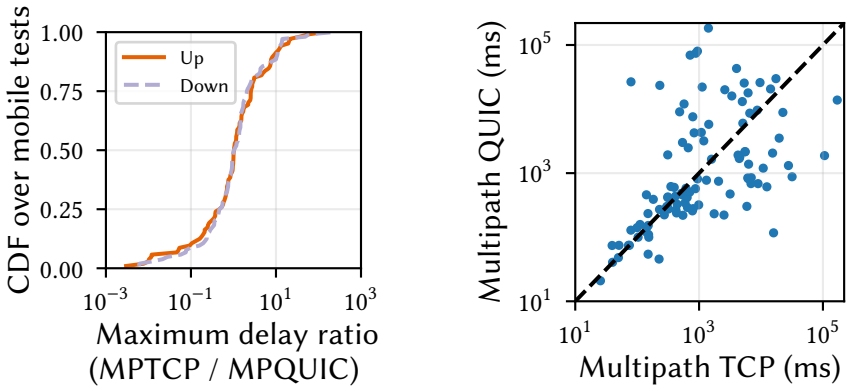
ments, using IPv6 leads to shorter completion times than using IPv4. Considering the same test set and looking at the *streaming* traffic pattern, we observe lower maximum experienced delays with gQUIC IPv6 than with gQUIC IPv4. This might be related to the high deployment of IPv6 in American networks, where the IPv4 connectivity is provided by NAT64. Other studies [Dha+12] have shown that IPv6 was faster than IPv4 in mobile networks.

Usage of gQUIC on Unofficial Ports. QUIC usually runs on port 443, but some middleboxes might expect other protocols such as DTLS on this port and could interfere with QUIC [Lan+17]. To detect the presence of such middleboxes, we also run our *ping* traffic with gQUIC on the non-standard port 6121 to observe if it experiences connectivity issues. Globally, we do not observe much difference between ports 6121 and 443.

Performance of Multipath gQUIC vs. Multipath TCP. When the smartphone can use both Multipath TCP and Multipath gQUIC, we observe similar performance for each protocol with our different traffic patterns. This is expected as we applied the same scheduling strategy preferring the Wi-Fi network with both protocols. However, especially with the *iperf* traffic pattern, we notice that when the network offers a large upload bandwidth, i.e., over 50 Mbps, Multipath TCP achieves a much higher throughput than Multipath QUIC. This is probably related to the implementation overhead. Multipath TCP in the Darwin kernel is much more optimized than the `gomobile` framework making the link between Swift code and the user-space `mp-quic` implementation written in Go.

A Note About the Extreme Multipath TCP Streaming Run In the light of the Multipath gQUIC run, we reconsider the Multipath TCP extreme *streaming* runs facing a maximum upload delay of 5.5 s shown in Figure 2.11. The Multipath gQUIC run take place 35 s before the Multipath TCP one, also without having Internet connectivity over the Wi-Fi network. Still, the maximum observed application delay was 51 ms, which is very different from the 5.5 s with Multipath TCP. This shows that smartphones might experience very different network conditions within a short period, even without user mobility.

In Real Networks, Path Validation Is Required. When performing our experiments, we identified a design issue affecting gQUIC, and therefore Multipath gQUIC. Recall that in gQUIC, there is no particular mechanism to assess whether a path is really working. This allows Multipath gQUIC to directly use new addresses when communicated. However, such approach is



(a) Ratio of max delays between Multipath TCP and Multipath QUIC.

(b) Upload stream max delays. Each point represents a test.

Figure 5.18: The performance of both protocols mainly depends on the network conditions.

unsafe when networks are either asymmetrical or subject to middlebox interference. We faced this last case with a smartphone connected to a dual-stack Wi-Fi network. While Multipath TCP managed to download a 10 MB file in about 5 seconds, Multipath gQUIC needed more than 10 minutes to complete the transfer. Digging in this particular run, the connection began with IPv4 addresses. Then, the client started to send packets over IPv6 ones. The network lets those IPv6 packets flow from client to server, but it — probably a firewall — dropped all the IPv6 packets towards the client. Such situation made the server crazy, as it received packets containing mostly control frames on the IPv6 path (so the path seemed to work), but none of its sent data was acknowledged. To address this case, we adapted Multipath gQUIC hosts to consider a path as usable only if it receives acknowledgments for previous packets. If a host has nothing useful to send, we trigger a PING frame on the required path. We limit the sending rate of a non-validated path to an initial congestion window. With such adaptation, our `mp-quic` manages to achieve similar performance as Multipath TCP for our 10 MB download.

5.3.3 Mobile Experiments

With the built-in support of Multipath TCP in iOS and Multipath QUIC provided by our application, MultipathTester can compare how both protocols handle network handovers when the user moves. Here, we study a subset of the collected dataset where both protocols were usable. This dataset contains 104 experiments, involving 40 cellular networks and 61 Wi-Fi ones.

To compare their performance, we consider the maximum delay experienced by each of the protocols over the same run. We then compute for each mobile test the ratio between the maximum delay of Multipath TCP and Multipath QUIC. Figure 5.18a shows that when Multipath QUIC uses the same scheduling strategy as Multipath TCP, we do not observe a clear trend in favor of one protocol over the other. Each of them tends to start using the cellular interface at the same time. In addition, Fig. 5.18b indicates that for a given test run, we can observe very different experienced maximum delays between Multipath TCP and Multipath QUIC. This result is both surprising and encouraging. Although Multipath TCP is included in iOS11, it does not seem to detect handovers better than our application. This indicates that smartphone applications that will include Multipath QUIC in the future could reach similar handover efficiencies as Multipath TCP.

Yet, Multipath QUIC avoids some of the issues that Multipath TCP encounters. Recall the case described in §2.3.4 where some subflows are not established, probably due to middlebox interference with the clear-text Multipath TCP options. Because it is encrypted, Multipath QUIC does not suffer from these issues, leading to much lower application delays.

5.4 Related Works

Before our Multipath gQUIC contribution, few scientific articles have analyzed QUIC or its performance. Megyesi et al. compare QUIC, SPDY and HTTP [MKM16]. They analyze page load times and report that each protocol has some advantages over the others in some environments. Carlucci et al. performed a similar study with HTTP/1.1. and a earlier version of gQUIC [CDM15]. Langley et al. reported gQUIC represents more than 30% of the egress traffic at Google, a large fraction being induced by mobile devices [Lan+17]. Kakhki et al. [Kak+17] identified some performance issues with the Chromium implementation of gQUIC.

Our Multipath gQUIC work attracted interest from other researchers. Viernickel et al. [Vie+18] concurrently developed a Multipath gQUIC prototype atop the `quic-go` implementation. They focus on specific scenarios and confirm most of the findings presented in this Chapter. Rabitsch et al. [RHB18] leverage our `mp-quic` implementation to develop a stream-aware scheduler based on a previously proposed Multipath TCP scheduler [Lim+17].

5.5 Conclusion

Multipath capabilities are important for smartphone and dual-stack hosts. In this Chapter, we proposed extensions to the gQUIC that enable this new

protocol to use several paths simultaneously. Our extensions remain simple thanks to gQUIC's flexible design. We implemented Multipath gQUIC in the open-source `quic-go` implementation in Go. We evaluated its performance over more than a thousand Mininet scenarios covering a wide range of parameters. We showed that in lossless environments, the performance of Multipath TCP and Multipath gQUIC are comparable. When losses are present, we pointed out that the usage of multiple paths is gQUIC remains beneficial, while this multipath capability is less advantageous for TCP. Then, we extended our analysis to actual wireless networks by leveraging our iOS MultipathTester application. We showed that Multipath gQUIC works in real networks, and when using the same multipath algorithms, it achieves results similar to Multipath TCP ones.

Still, QUIC is a very changing protocol and the current iQUIC version is now very different from the base gQUIC one. In addition, our in-the-wild measurements revealed some gQUIC design issues affecting Multipath gQUIC. With our gained knowledge, we address those concerns in Chapter 6.

Rethinking the Multipath Extensions for IQUIC

6

Throughout the realization of this thesis, the QUIC protocol was in constant evolution. During the IETF standardization process and considering the implementations' experience, it gradually gained in complexity. Its specification is split into several documents. Their contents contain keywords defining a specific behavior that a protocol implementation must follow. The main document about the transport features of QUIC initially contained 44 pages and less than 100 keywords (GQUIC, or IQUIC version 00). Now, the same document spans 139 pages (without appendices) mentioning 380 keywords (IQUIC version 24).

Because the base QUIC protocol changed, the design of its Multipath extensions was also affected. However, Multipath GQUIC suffers from many design issues coming from GQUIC itself that we first discuss (§6.1). The current IQUIC version fixes them, and based on our gained experience, we revisit the design of our Multipath extensions (§6.2). Multipath IQUIC leverages the current architecture to revisit how a multipath transport protocol can benefit from network asymmetries (§6.3). We also present a resilient connection establishment strategy to make Multipath IQUIC less affected by the initial network path choice (§6.4). We finally conclude with the lessons learned (§6.5).

6.1 Design Issues Affecting Multipath GQUIC

The initial version of our Multipath extensions described in Chapter 5 raises several major concerns that are critical for possible IETF standardization. In particular, Multipath GQUIC does not check that new paths are actually usable (§6.1.1), is vulnerable to pervasive monitoring (§6.1.2), poses technical challenges due to the utilization of the GQUIC Connection ID (§6.1.3) and does not enable hosts to control the number of paths used (§6.1.4).

6.1.1 Absence of Path Validation

Recall from Section 4.6 that GQUIC does not perform any particular operation when it observes a change in the 4-tuple used by a connection. Our in-the-wild experiments described in Section 5.3.2 confirm that this missing feature

is detrimental to Multipath GQUIC. We adapted our mp-quic implementation to consider a path as usable only if it received acknowledgments at some point. However, this approach is a quick fix. To reach a wide adoption, the Multipath extensions should embed a clean solution in the protocol itself.

6.1.2 Correlating Multipath Traffic

In the GQUIC design, the packet number is put in the clear-text packet header. Multipath GQUIC does the same with the Path ID to inform middleboxes that observing decreasing packet numbers is expected. Nevertheless, it also exposes the connections to passive path correlation. An attacker can easily figure out that two different IP addresses actually belong to the same host. As IQUIC focuses on mitigating pervasive monitoring [RFC7258], its Multipath extensions should not introduce new privacy concerns.

6.1.3 Immutable Symmetric Connection ID

In addition to the privacy concerns, a fixed symmetric Connection ID also causes at least three technical issues. First, since the Connection ID is chosen by the client, a server might experience several concurrent GQUIC connections using the same identifier. Second, it makes it difficult to move a GQUIC flow from one host to another one, e.g., to migrate a client to a closer server than the current one. Third, Multipath GQUIC relies on the assumption that network paths are bidirectional, hence sustaining packet delivery in both directions. However, our experience reveals that actual networks do not always let packets go through in both directions.

6.1.4 Uncontrolled Path Management

Multipath GQUIC hosts can initiate new bidirectional paths at any time without requiring the peer's consent. Creating a new path is as simple as sending a packet carrying a new Path ID. Nevertheless, this approach does not allow a host to restrict the number of paths over a given connection. For instance, a peer may not want to allocate too many resources to support a high number of concurrent paths. A malicious client could also perform a Denial-of-Service attack by requesting the usage of a huge number of paths. Note that although Multipath TCP relies on a 4-way handshake to fully establish additional subflows, it does not embed a protocol mechanism to indicate the desired number of paths over a connection.

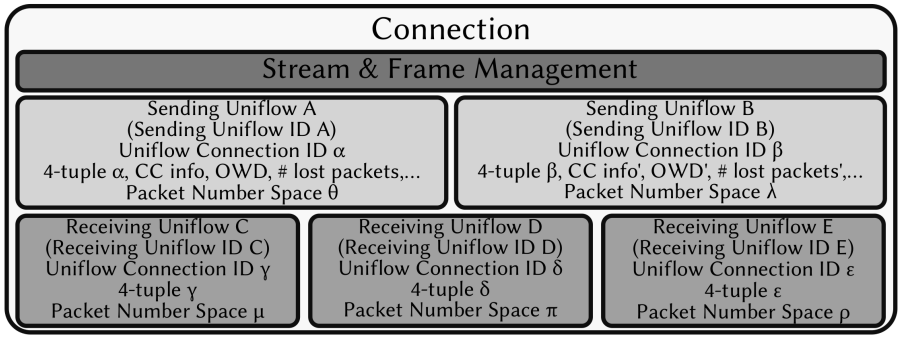


Figure 6.1: Architectural view of Multipath ɪQUIC with a connection having two sending uniflows and three receiving uniflows.

6.2 Multipath Extensions for ɪQUIC

We now present our Multipath extensions updated to ɪQUIC addressing the concerns described in the previous Section. Compared to Multipath gQUIC, "paths" are actually unidirectional flows — called uniflows — and their packets are identified using the Connection ID (§6.2.1). Such an approach allows QUIC hosts to keep the number of uniflows used by the peer under control (§6.2.2). The Multipath extensions are negotiated at connection establishment (§6.2.3). Data and control frames can then be spread over multiple network paths (§6.2.4). A Multipath ɪQUIC host can also advertise its mapping between local IP addresses and uniflows to its peer (§6.2.5). Nonetheless, this path asymmetry impacts the uniflow latency estimation (§6.2.6) and requires some adaptation of the hosts' algorithms, such as the packet scheduler (§6.2.7). Still, Multipath ɪQUIC addresses the main concerns raised by Multipath gQUIC (§6.2.8).

6.2.1 Identifying Unidirectional Flows with Connection IDs

Similar to Multipath TCP, Multipath gQUIC considers a path as a bidirectional channel where each direction exhibits similar path characteristics. However, network links such as ADSL do not provide the same bandwidth in each direction and the One-Way-Delay (OWD) often differs between the forward and the reverse directions on the Internet [Pat+08]. Multipath ɪQUIC breaks the symmetric assumption and leverages the asymmetric Connection IDs to integrate unidirectional paths called *uniflow*. In other words, packets belonging to a given connection's uniflow exhibit the same Destination Connection ID in their headers[1]. We refer to this specific Connection ID as the Uniflow

[1]The QUIC expert would — rightly — say that all packets flowing over a given uniflow do not necessarily share the same Destination Connection ID, as it can be updated with NEW

Frame Type	Uniflow ID	Sequence Number
Retire Prior To		Length (1 byte)
Connection ID (Length bytes)		
Stateless Reset Token (16 bytes)		

Figure 6.2: Multipath version of the NEW CONNECTION ID frame. The visual size of fields does not reflect their actual length, annotated to each field. Non annotated fields are variable-length integers.

Connection ID (UCID). We distinguish two kinds of uniflows, as illustrated in Figure 6.1. On the one hand, the *sending uniflow* can transmit packets to the remote host by including the corresponding Uniflow Connection ID in the headers. On the other hand, the *receiving uniflow* collects incoming packing carrying the related Uniflow Connection ID in their headers. Notice that each uniflow has its internal Uniflow ID.

This notion of uniflow is seamlessly included in single-path IQUIC. Each packets' flow uses a different Connection ID in the header. Each host sees a specific 4-tuple (IP_{src}, IP_{dst}, $port_{src}$, $port_{dst}$) which is not necessarily the mirror of the peer's one since middleboxes like NAT may modify packets' ports and IP addresses. Therefore, the uniflows on which the connection starts are called the *initial uniflows*. Hosts identify these initial uniflows using Uniflow ID 0.

6.2.2 Proposing Sending Uniflows to the Peer

There is a one-to-one mapping between the receive uniflows of a host and the sending uniflows of its peer. More specifically, the receive uniflow advertised by host a identified by the Receive Uniflow ID C and mapped to the Uniflow Source Connection ID γ corresponds to the sending uniflow of host b identified by the Sending Uniflow ID C and carrying packets with the Uniflow Destination Connection ID γ. Each host controls its number of receive uniflows — and so the number of sending uniflows of the peer — it wants to maintain. Once the uniflow is initialized, the host can advertise it to its peer using the adapted version of the NEW CONNECTION ID frame shown in Figure 6.2. This frame indicates to the receiver that this Sending Uniflow ID is now usable and that packets can be sent over it by including the communicated Connection ID in their headers. The NEW CONNECTION ID typically

CONNECTION ID frames. For the sake of simplicity, we assume now that this Connection ID does not change.

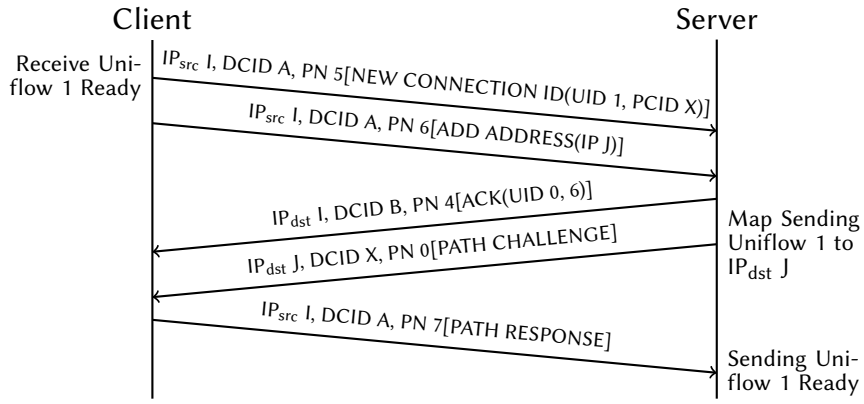


Figure 6.3: An example of a server starting using a new sending uniflow. The client starts the connection on its IP I and the server always uses the same IP. Only the client's Source IP (IP_{src}) and Destination IP (IP_{dst}), the Destination Connection ID (DCID), the packet number (PN) and the relevant frames are illustrated.

comes along with an ADD ADDR frame — using the same format as Multipath gQUIC illustrated in Figure 5.3 — to let the remote use the uniflow over the advertised IP address. Figure 6.3 illustrates how a client can propose a new uniflow to the server. Once both the NEW CONNECTION ID and the ADD ADDR frames received, the server decides that packets of its sending uniflow 1 are sent to the IP address J. However, this new IP address must first be validated before starting transmitting data. Hence, the server initiates a path validation over this new sending uniflow by delivering the PATH CHALLENGE frame. Since the client has only one sending uniflow — its initial one — it carries the corresponding PATH RESPONSE frame to the initial sending uniflow. Once the path validation completes, the server can start using the provided sending uniflow to send packets towards the client.

Notice that Multipath iQUIC keeps the Sequence Number field in the NEW CONNECTION ID frame as pointed out in Figure 6.2. This enables hosts to change the Connection ID used to mark packets over a given path for, e.g., privacy concerns.

The server must validate the provided remote address to prevent flooding a victim pointed by a malicious client. This process delays the usage of the new uniflow by a RTT. However, the validity of the provided address might be cached to save the latency introduced by the path validation.

Note that Multipath iQUIC keeps the same REMOVE ADDR frame as Multipath gQUIC to advertise the loss of a previously communicated address. The frame contains the Address ID of the corresponding address along with a sequence number ordering the events to that particular Address ID.

Frame Type	Uniflow ID	Largest Acknowledged
ACK Delay		ACK Range Count
First ACK Range		ACK Ranges (# ACK Range Count)

Figure 6.4: Multipath version of the ACK frame. All its fields are variable-length integers, the visual size of fields is for reader's convenience.

A host might also discover new peer's addresses by observing the 4-tuple of incoming packets. A client behind a NAT could send its private addresses in ADD ADDR frames, but those are typically not reachable by the server. Instead, the client can initiate the uplink flow on its desired addresses — for instance by performing path validation — such as the server discovers the publicly accessible addresses. The server can then validate the seen addresses in order to use them.

6.2.3 Negotiating the Multipath Extensions

During the handshake, hosts exchange the `max_sending_uniflow_id` QUIC transport parameter to indicate the maximum value of Sending Uniflow ID that they want to support. If both hosts advertise it, the multipath extensions are enabled on the connection. These values are independent, i.e., hosts may advertise different `max_sending_uniflow_id` values. Because of the mirroring between uniflows, the sending host sets the upper bound of the number of peer's receiving uniflows to the advertised value. Yet, this transport parameter does not impose the peer to provide the desired number of sending uniflows. The receiver keeps the final decision on the number of receiving uniflows it wants to support since it sends the NEW CONNECTION ID frames proposing them.

6.2.4 Ensuring Reliable Data Exchange

As Multipath gQUIC, Multipath iQUIC relies on STREAM frames to carry application data. Hence, we just need to acknowledge packets on a per-uniflow basis where each uniflow has its own packet sequence number space. Figure 6.4 shows that our Multipath extensions adapt the ACK frame to include the host's Receiving Uniflow ID — which is equivalent to the peer's Sending Uniflow ID — it acknowledges. The ACK frame uses the (internal) Uniflow ID instead of the related Connection ID for three reasons. First, the Uniflow ID takes fewer bytes than the Connection ID. Second, the Connection ID used by a uniflow might change over time. Third, the Uniflow ID is included in the nonce computation to avoid reusing a same nonce over distinct uniflows.

Frame Type		Sequence
Active Receiving Uniflow		Active Sending Uniflow
Receiving Uniflow ID 0	Local Address ID RU0	Remote Address ID RU0
... Other Receiving Uniflow Entries ...		
Sending Uniflow ID 0	Local Address ID SU0	Remote Address ID SU0
... Other Sending Uniflow Entries ...		

Figure 6.5: Format of a UNIFLOWS frame. All its fields are variable-length integers, the visual size of fields is for reader's convenience.

6.2.5 Acknowledging the Addresses Used by Uniflows

Multipath iQUIC hosts may observe different 4-tuples for a given uniflow. For instance, a NAT might be present such as the client sees a private address (e.g., 192.168.1.10) while the server observes a publicly reachable one. To get a complete view of the utilized addresses, Multipath iQUIC hosts rely on the ADD ADDR and UNIFLOWS frames. Each advertised address by a ADD ADDR has an identifier, as illustrated by Figure 5.3 on page 98. However, a client may communicate a private address to its peer, and the server would observe different addresses. Hence, the client also sends the UNIFLOWS frame shown in Figure 6.5 to indicate to which local Address ID the source (resp. destination) IP address of sending (resp. receiving) uniflows corresponds. This enables the frame receiver to map the observed IP address to an advertised one. The server can then appropriately react to a REMOVE ADDR, even if the 4-tuple diverges between hosts. The UNIFLOWS frame's sender also communicates the remote Address ID s whose it believes that its uniflows use. Note than unlike in Multipath gQUIC, the UNIFLOWS frame does not currently contain the host estimated round-trip-time.

6.2.6 Estimating the Latency of Uniflows

The major concern of introducing uniflows resides in their latency estimation. When using a bidirectional path, a reliable transport protocol can estimate its RTT by observing the delay between the data sending and its corresponding acknowledgment. However, with Multipath iQUIC there is no more guarantee that ACK frames are deterministically coming from a receiving uniflow. Previous works proposed techniques to estimate the One-Way-Delay (OWD) of a unidirectional path [CSP15] but getting precise OWD is hard without clock synchronization [DRT08]. Instead, we propose to consider the RTT be-

tween tuples of sending uniflows and receiving ones. This approach is easy to implement as the host knows on which sending uniflow it sent the packet and on which receive uniflow it received the corresponding acknowledgment. Therefore, duplicating acknowledgments on several uniflows does not modify the RTT estimation of a uniflow tuple.

6.2.7 Impacts on the Multipath-specific Algorithms

Congestion Control Algorithms. Compared to Multipath GQUIC, the path asymmetry does not impact the congestion control considerations. These schemes only operate on the sending uniflows.

Path Manager. It is now split into two parts. First, it decides how many receiving uniflows it provides to the connection. Second, it waits for proposed sending uniflows and associates them to a 4-tuple. The mapping algorithm can follow previously proposed strategies, such as the `full-mesh` one. Notice that unlike Multipath TCP and Multipath GQUIC, the server also performs its sending uniflows' management. To limit network interferences (NAT, firewalls,...), the server first waits for the reception of a packet with a given address before using it for its sending uniflows.

Packet Scheduling. In comparison to Multipath GQUIC, two elements affect the packet scheduling. First, the algorithm must take into account the sending uniflow — and its corresponding 4-tuple — it wants to validate to schedule a PATH CHALLENGE frame. This enables it to collect a latency estimation before starting sending data over it. Notice that in the case the 4-tuple was recently validated in a previous connection involving the same hosts, the scheduler might directly reuse it with cached information. Second, a latency-based packet scheduler should now rely on the RTT between tuples of uniflows to decide on which sending uniflows it should send packets. While the actual OWD of a sending uniflow is not available, such a scheduler can determine a partial order between the latency of the sending uniflows to prioritize the transmission of packets on the fastest uniflow. In practice, our latency-based packet scheduler performs a sending uniflow's RTT estimation based on its 2-tuples (sending uniflow ID, receiving uniflow ID) RTT estimations, weighted by the number of packets belonging to these 2-tuples.

6.2.8 Summary

Overall, our updated design leverages the features of IQUIC to address the concerns introduced by Multipath GQUIC described in Section 6.1 while keeping a clean design. Our Multipath extensions extend the IQUIC Connection

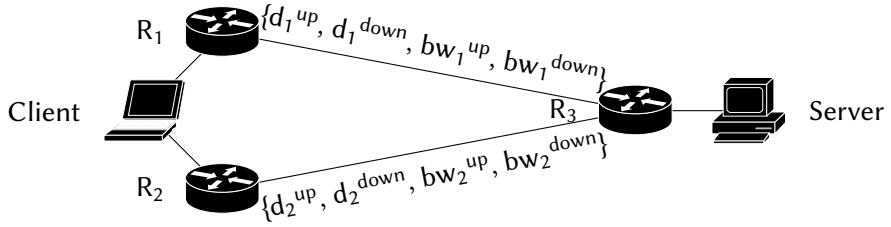


Figure 6.6: Network topology used for asymmetric experiments.

Factor	Min.	Max.
Capacity [Mbps]	5	50
One-Way-Delay [ms]	2.5	25

Table 6.1: Experimental design parameters for the asymmetric experiments.

ID asymmetry to its unidirectional flows to make hosts aware of the network imbalance. Each host controls the number of peer's sending uniflows while determining their associated Path Connection ID. The encrypted uniflow negotiation through NEW CONNECTION ID makes it hard for an attacker to figure out that a group of Uniflow Connection IDs actually belong to a same multipath connection. Multipath ıQUIC validates a new 4-tuple before spreading packets over a sending uniflow using it.

6.3 Exploring Asymmetric Network Use Cases

To demonstrate the feasibility of our design, we implement our multipath extensions in the `picoquic` implementation[2] [Hui18]. To assess its benefits, we evaluate Multipath ıQUIC 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 6.6. The links R_1-R_3 and R_2-R_3 are configured using `netem` [Hem+05] to add transmission delays and using `htb` to limit their bandwidth.

To cover a large range of link characteristics, we rely on the experimental design approach [Fis35] and define the parameters' ranges as shown in Table 6.1. We then use the WSP algorithm [SCS12] to broadly sample this parameter space into 139 points. Notice that the uplink one-way-delay d^{up} and the downlink one d^{down} are two different parameters. The same applies for the uplink bandwidth bw^{up} and the downlink one bw^{down}. Since our topology features two paths, we explore a 8-dimension parameter space. Each parameter configuration is run 9 times and the median run is reported.

[2]The full description of our implementation is available in Section 7.4.2.

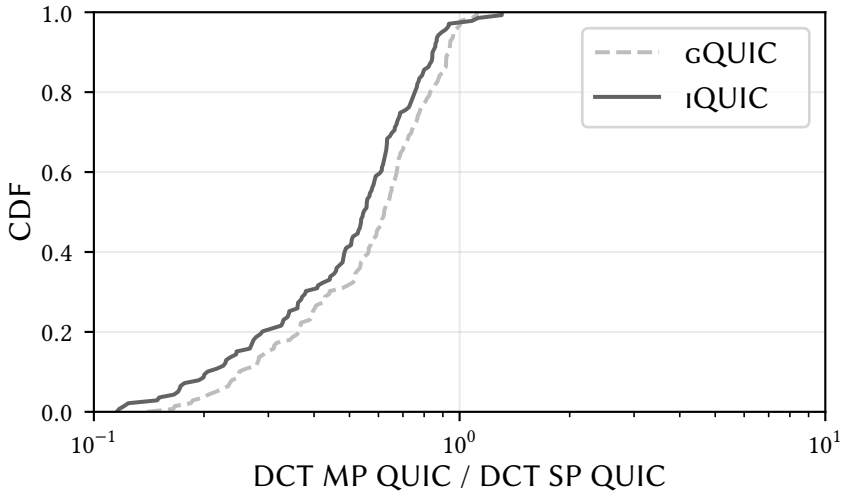


Figure 6.7: Comparing the benefits of the Multipath extensions between gQUIC and iQUIC.

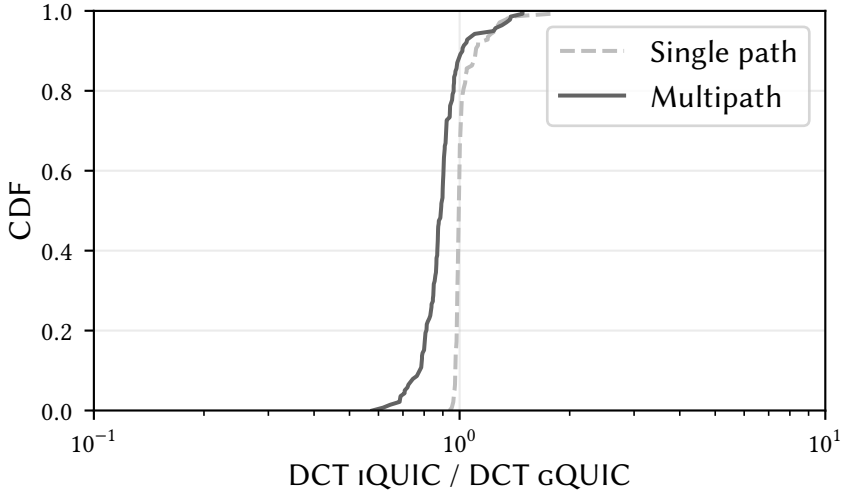


Figure 6.8: Comparing the performance of the gQUIC's implementation and the iQUIC's one.

In our experiments, we consider both our Multipath ıQUIC implementation based on `picoquic` and our previous Multipath gQUIC implementation `mp-quic`. Both use the lowest latency first packet scheduler. The client computes the Download Completion Time (DCT) between the issue of the GET request and the reception of the last byte of the server response. We consider here the download of a file of 10 MB over a single stream. For each implementation, we perform both single-path experiments over the R_1-R_3 link and multipath ones using both links. Figure 6.7 shows that while both QUIC implementations achieve lower DCT when using the two links, the multipath ability tends to benefit more to ıQUIC than to gQUIC. In our considered experiments, Multipath ıQUIC achieves a median speedup of 84% while the Multipath gQUIC one is 61%. By computing the DCT ratios between the implementation themselves, Figure 6.8 shows that single-path ıQUIC and gQUIC achieve similar performance. This means that comparing them directly, Multipath ıQUIC is in general faster than Multipath gQUIC.

Two factors explain these results. First, our Multipath ıQUIC packet scheduler leverages the path asymmetry awareness such that it can efficiently utilize a link exhibiting a low downlink OWD and a high downlink bandwidth, even if they exhibit a high uplink OWD and a low uplink bandwidth. In comparison, Multipath gQUIC considers both directions of a path as a whole and does not distinguish uplink characteristics from downlink ones. Such specific links affect the Multipath gQUIC's packet scheduler decisions. Second, our `mp-quic` implementation forces the ACK frames to be sent on the same path they acknowledge. This acknowledgment strategy addresses concerns about the variability of path's RTT estimations in Multipath gQUIC. Because the Multipath ıQUIC latency estimations are specific to a tuple (sending uniflow, receiving uniflow), there is no more such issue. Our Multipath ıQUIC client actually embeds ACK frames for all its receiving uniflows in all packets it sends to the server. This strategy is beneficial when one network path suffers from a very low uplink bandwidth.

To confirm our intuitions, we reconsider the same experiments with the same tested networks (139 points), except that in all these configurations we set ${bw_2}^{up}$ to 0.1 Mbps. Figure 6.9 confirms that in such situations, Multipath gQUIC barely benefits from the additional path as it receives its acknowledgments at a very low rate. Instead, our Multipath ıQUIC implementation achieves to aggregate the downlink bandwidth of the additional link, even if its uplink one is very low.

6.4 Making Connections Resilient to the Initial Path Choice

Our Multipath extensions enable hosts to simultaneously use several network paths once the connection is established. Still, Multipath ıQUIC remains

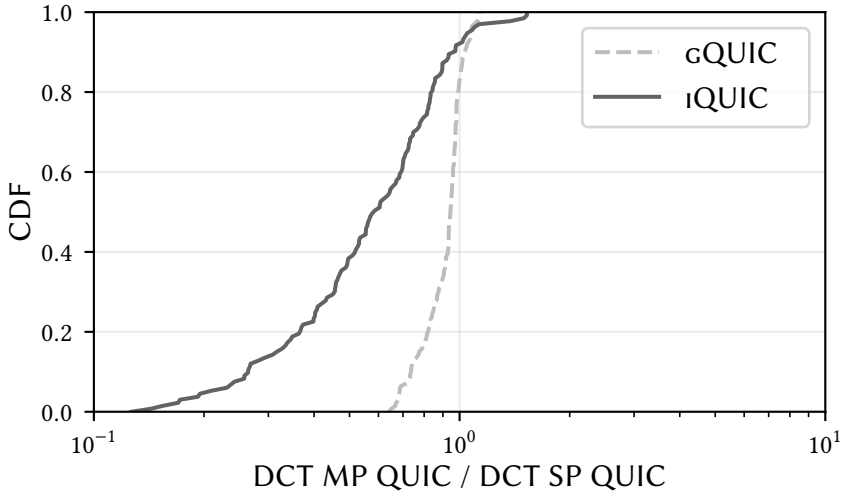


Figure 6.9: Comparing the benefits of the Multipath extensions between GQUIC and IQUIC when the additional path exhibits an uplink bandwidth of 0.1 Mbps.

single-path during the handshake since it needs to negotiate the usage of the extensions. The transfer time can therefore be affected by a high-latency initial path.

Instead of adding complexity to the IQUIC handshake process, we propose a *resilient* connection start by initiating a Multipath IQUIC exchange on each available network path. The client will notice one of them that receives first a reply from the server. The client then closes the connection with the slowest handshake and continues the transfer with the fastest one. Consider the example shown in Figure 6.10. The client initiates two connections, C1 and C2, on two different network paths. As it receives first a response from C2 on Path 2, the client closes C1 and continues the transfer with C2. Based on the received NEW CONNECTION ID and ADD ADDR frames, the client then initiates path validation of Path 1. As C2 was established over Path 2, hosts can directly exchange regular IQUIC packets on it. Such a strategy makes the Multipath IQUIC performance independent of the initial network path choice. Notice that our resilient Multipath IQUIC allows taking advantage of the bidirectional path exhibiting the lowest RTT, but it does not consider network asymmetry. A similar strategy is currently considered for Multipath TCP at the IETF [AK19].

To assess the benefits of the resilient strategy, we consider a two-path network topology as shown in Figure 6.6. However, paths are symmetric —

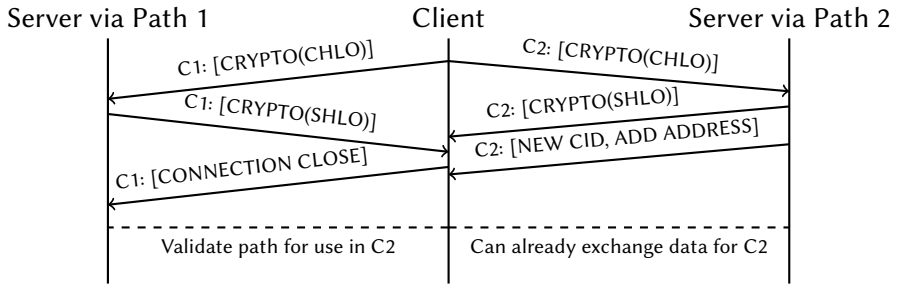


Figure 6.10: Resilient Multipath IQUIC connection establishment, here with the case of Path 2 being faster than Path 1.

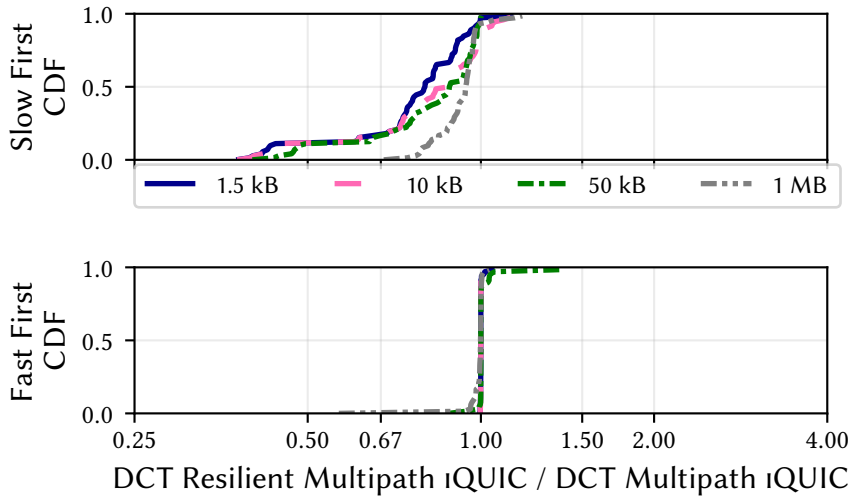


Figure 6.11: When the first path exhibits a higher latency than the additional one, launching a redundant connection is beneficial, especially for short transfers. Otherwise, the performance does not change.

same characteristics in upload and download — and the bandwidth is set to 50 Mbps. We restrict our parameter space to the bidirectional RTT of each path ranging from 10 ms to 400 ms and sample 139 points. The buffer sizes are set to the bandwidth-delay product. We run the experiments 9 times and the median run is reported. We consider here the time between the first packet sent by the client and the reception of the last byte of the fetched file by the client. Figure 6.11 splits our studied network scenarios into two categories. The first one contains the situations where the first path exhibits a higher RTT than the additional one. In that case, if the client initiates a second connection on this additional link — as resilient Multipath IQUIC does — its handshake first completes and the transfer time decreases. Such redundant connection establishment is especially beneficial to short transfers whose the connection handshake constitutes a major fraction of the download time. The relative ratio of completion time decreases with the size of the file transferred. The second category groups the situations where the first path shows a lower latency than the additional one. In that case, both versions of Multipath IQUIC have the same performance, as expected.

6.5 Conclusion

In the previous Chapter, we proposed Multipath extensions to GQUIC enabling hosts to simultaneously take advantage of several paths. However, they suffer from design issues that this Chapter first listed. Based on our experience, we designed and implemented Multipath extensions to IQUIC addressing all these concerns. A key point of Multipath IQUIC is its awareness of path asymmetries allowing client to server packets to flow on a given network while another path carries server to client ones. We performed an experimental design evaluation assessing the benefits of our approach in scenarios where links show asymmetric characteristics. We also presented a technique to limit the impact of using a slow path to initiate a Multipath IQUIC connection. All in all, we believe that this multipath design would benefit to other Internet actors and we hope it will be considered for standardization.

Pluginizing QUIC

7

Transport protocols including TCP [RFC793], RTP [RFC1889], SCTP [RFC2960] or QUIC [QUIC-T; Lan+17] play a key role in today's Internet. They extend the packet forwarding service provided by the network layer and include a variety of end-to-end services. A transport protocol is usually designed to support a set of requirements. Protocol designers know that these requirements evolve over time and all these protocols allow clients to propose the utilization of extensions during the connection's handshake.

These negotiation schemes have enabled TCP and other transport protocols to evolve over the last decades [RFC7414]. A modern TCP stack supports a long list of TCP extensions. These include Window Scale [RFC1323], Timestamps [RFC1323], Selective Acknowledgments [RFC2018], Explicit Congestion Notification [RFC3168] and Multipath extensions [RFC6824]. However, measurements indicate that it remains difficult to deploy TCP extensions [Fuk11; Hon+11]. Both the Window Scale and the Selective Acknowledgment options took more than a decade to be widely deployed [Fuk11]. The Timestamp option is still not supported by a major desktop OS [Bal18]. Multipath TCP is only available in one major mobile OS [App18]. This slow deployment of TCP extensions is caused by three main factors. First, popular stacks rarely implement TCP extensions unless they have been standardized by the IETF. Second, TCP is still part of the operating system and clients' and servers' implementations are not upgraded at the same pace. Often, maintainer of clients' (resp. servers') implementations wait until the servers' (resp. clients') ones support a new extension before implementing it. This results in a chicken-and-egg deployment problem. Third, some middleboxes interfere with the deployment of new protocol extensions [Hes+13; Hon+11].

The QUIC protocol [Lan+17] described in Chapter 4 addresses most of these deployment issues. Thanks to its encryption features and its framing mechanism encoding user data and control information, it prevents most of the middleboxes' interference [Lan+17]. In addition, its protocol version negotiation enabled Google to update its proprietary gQUIC without requiring IETF consensus. Measurements indicate that Google updated its gQUIC version — shipped as a library — at the same pace as its Chrome browser [Rüt+18].

We believe that the openness of the Internet is a key element of its success,

and ultimately anyone should be able to tune or extend Internet protocols to best fit their needs. Traditional transport protocols like TCP are tuned using configuration variables or socket options [DMT02] and more recently with eBPF code [Bra17]. Although recent works enable an application to add new TCP options [TB19], it remains impossible to precisely configure the underlying TCP stack.

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. Such an approach could enable QUIC applications to adapt their underlying transport layer to their specific needs, e.g., using specialized packet scheduling 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 into a set of *protocol plugins* 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 IQUIC. We add to `picoquic` a virtual machine that allows executing the bytecode of protocol plugins in a platform independent manner while monitoring their behavior.

- We demonstrate the benefits of PQUIC through a plugin that adds Multipath support and show how we can tune it to different use cases.

This Chapter is organized as follows. First, we present the design of PQUIC and how our PQUIC implementation supports plugins (§7.1). We next propose an overview of the security models, attacks and how PQUIC solves them when exchanging plugins over QUIC connections (§7.2). After reporting our experience with an early prototype (§7.3), we describe how we implement the Multipath extensions using only plugins and evaluate their overhead

(§7.4). We also explore how plugin properties, and in particular their termination, can be verified (§7.5). After contrasting with related works (§7.6) and discussing the broader implications and questions of this work (§7.7), we finally conclude and consider future work (§7.8).

7.1 Local Plugin Insertion

Our design for *Pluginized QUIC* (PQUIC) builds upon the QUIC protocol whose specification is being finalized within the IETF[1] [QUIC-T]. From a protocol viewpoint, there are few differences between PQUIC and QUIC. We defer the explanation of these differences until Section 7.2.

From the implementation viewpoint, the main difference between a PQUIC implementation[2] 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 behaviors 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 provide a small external API to applications. For example, a 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. These functions are shown in Figure 7.1 as light gray rectangles. 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. Sample protocol operations in PQUIC include the parsing and processing of frames, the preparation of the packet's header, setting the retransmission timer, updating the RTT, adding new data in the sending buffer, etc. By default, all the connections share a common built-in behavior provided by the PQUIC implementation.

[1] We base this work on the version 14 of the IETF QUIC drafts [QUIC-14].

[2] We applied the principles described in this Section to two very different QUIC implementations. We first implemented an early prototype based on `quic-go` [CS18] written in Go [DB19b]. We then developed a much more advanced implementation based on `picoquic` [Hui18] written in C. This Section focuses on the `picoquic` implementation, but we discuss the `quic-go` one in Section 7.3.

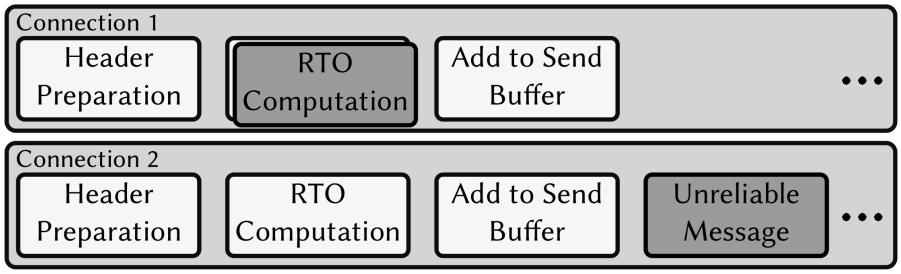


Figure 7.1: High-level architecture of the PQUIC gray-box model with two running connections. These concurrent connections can be tuned by distinct protocol plugins modifying or adding protocol functions. Protocol operations whose behavior is provided by protocol plugins are represented by darker squares.

On-the-fly protocol plugin insertion. A *pluglet* consists of bytecode instructions implementing a function, e.g., computing an RTT estimate. A *manifest* contains the globally unique plugin name and indicates how to link several pluglets to a connection, i.e., to which protocol operations they should be attached. The combination of the pluglets and the manifest forms a *protocol plugin*. Once a PQUIC connection is established, PQUIC can potentially load plugins at any time to provide a new behavior for functions, represented by dark gray rectangles in Figure 7.1. Notice that PQUIC can also integrate new protocol operations that were not present in the base protocol implementation, such as the processing of unreliable messages as pictured in Figure 7.1.

Isolation between connections and between plugins. Each plugin is instantiated to operate on a given connection. Distinct connections can load different plugins as shown in Figure 7.1 without any interference. 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 by any other plugin or connection. This yields strong memory safety guarantees for the plugins and the sharing of information. Interactions between them are still possible through the protocol operation interface or by calling the functions exposed by PQUIC. Nevertheless, these are clearly defined information flows that ease the reasoning about the behavior and the safety of the plugins.

The remaining of this Section details the core elements of PQUIC. We first describe the environment executing pluglets (§7.1.1). Next, we elaborate on the concept of protocol operations (§7.1.2). We then describe how PQUIC interfaces with pluglets (§7.1.3). Finally, we provide details about how plugins can interact with the application they serve (§7.1.4) and some optimization about plugin reuse (§7.1.5).

7.1.1 Pluglet Operating Environment (POE)

Pluglets are the building blocks of the protocol plugins. These pieces of bytecode are independent of the PQUIC implementation itself. Therefore, we need to provide an environment to execute them. This environment 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 both these issues, PQUIC executes pluglets inside a lightweight virtual machine (VM). Various VMs have been proposed for different purposes [BMG99; Fle17; Geo+10; Haa+17; Lin+14; Wan+15]. In our current PQUIC version, our *Pluglet Operating Environment* (POE) relies on a user-space implementation [IO 18] of the eBPF VM [Fle17]. eBPF provides a concise number — about 100 — of 64-bit RISC instructions. It contains 11 64-bit registers labeled from R0 to R10. An eBPF program has read/write access to the first ten ones — from R0 to R9. R10 contains the read-only pointer to the top of the 512-byte long stack. An eBPF program can also call external registered functions through a dedicated `call` opcode and relies on five registers — R1 to R5 — to communicate the arguments. This overall architecture is voluntary close to the modern CPU ones to make Just-In-Time (JIT) compilation simple. Although being present in the Linux kernel since 2014 where it has been used to support various services [Bra17; Edg15; Gre15], the eBPF VM can be too restrictive to implement some legitimate behaviors. The kernel-space eBPF VM includes a verifier that is very conservative, as it puts hard limits on the size and complexity of an acceptable eBPF program.

Our user-space implementation extends a relaxed version of the eBPF verifier with additional monitoring capabilities. Those are similar to works in Software-Based Fault Isolation [Wah+94; Yee+09]. First, our POE 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 POE 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 POE monitors the correct operation of the pluglets by injecting specific instructions when their bytecode is JIT compiled. These monitoring instructions check that the memory accesses operate within the allowed bounds. To achieve this, we add a register to the eBPF VM (R11) 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

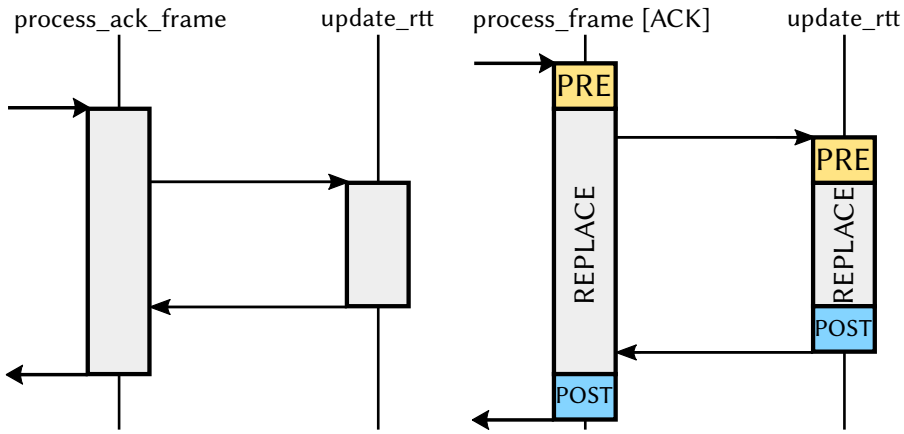
pluglet stack. Any violation of memory safety results in the call of a specific PQUIC-provided function removing the plugin and terminating the connection with an error. In practice, any memory access requested by pluglets leads to the inclusion of 20 additional eBPF instructions, plus the rewriting of the offsets of jump opcodes. 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.

7.1.2 Protocol Operations

In order to attach pluglets to PQUIC implementations, we need to define an API revealing the insertion points and the interface between PQUIC and the pluglets. For this, 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. PQUIC defines two kinds of protocol operations. On the one hand, a parameterizable protocol operation has a high-level goal, but its actual behavior — and therefore its function — changes depending on the given parameter. This provides a generic entry point allowing the definition of new behaviors without changing the caller for, e.g., the serialization of new QUIC frames. On the other hand, a non-parameterizable protocol operation defines a specific function. Our PQUIC implementation currently includes 72 protocol operations. Four of them take a parameter[3]. We can 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 data streams to send next, etc. A third category involves the QUIC packet management. It includes setting the Spin Bit [TK18], 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 behavior. For instance, a protocol operation exists after decoding all the frames of an incoming packet or when noticing a packet loss.

To illustrate how an implementation can be split into protocol operations, consider the example shown in Figure 7.2a. The processing of an ACK frame would likely be performed in its dedicated function. One of its sub tasks is the computation of the RTT estimation, which is implemented in its own function too. PQUIC keeps the same programming flow. As shown by Figure 7.2b, 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

[3]Please refer to `https://pquic.org` for the latest numbers.



(a) Monolithic design. (b) PQUIC design.

Figure 7.2: Turning a monolithic design into protocol operations with the ACK processing example. It also illustrates the different anchors for the pluglets.

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 behavior. 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 pluglet 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. In return, they only have read access to the connection context and the operation's arguments and outputs. The only write accesses they have are to their pluglet stack and their plugin-specific memory. In the rest of this Chapter, unless explicitly stated, we discuss pluglet insertion in REPLACE mode, and refer to pluglets inserted in PRE and POST as passive pluglets.

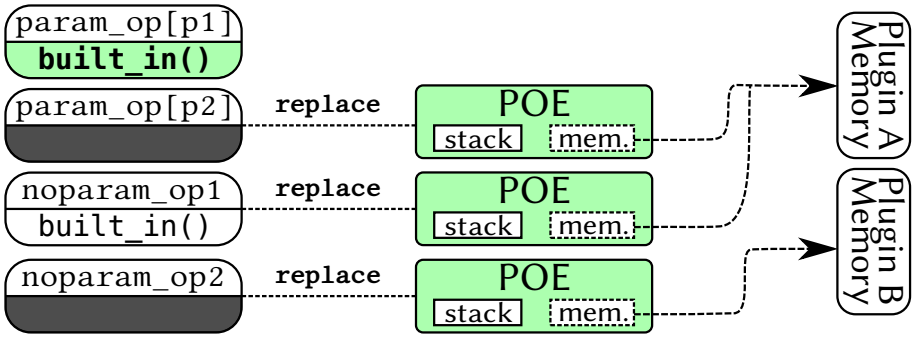


Figure 7.3: Attaching POEs in REPLACE mode to protocol operations.

7.1.3 Attaching Protocol Plugins

Implementing protocol extensions may require a combination of several pluglets forming a plugin. The POE 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 might need to share some state among its pluglets.

To address these needs, PQUIC is organized as illustrated in Figure 7.3. As explained previously, the behavior of a protocol operation is either provided by a built-in function (e.g., `param_op[p1]`) or overridden by a POE running 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 POE is created for each inserted pluglet. Each POE contains its own registers and stack. The POE heap memory points to an area common to all pluglets of a given plugin, as illustrated in Figure 7.3. This link, maintained 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 faulty plugin itself. Thanks to our POE, pointer dereferencing is restricted only to the pluglet stack and 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 POE. These functions form an API that pluglets can use. Table 7.1 presents the key functions of this API. We detail its six major operations below.

Exposing connection fields through getters and setters. Letting plugins directly access the fields of PQUIC structures makes the injected code

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

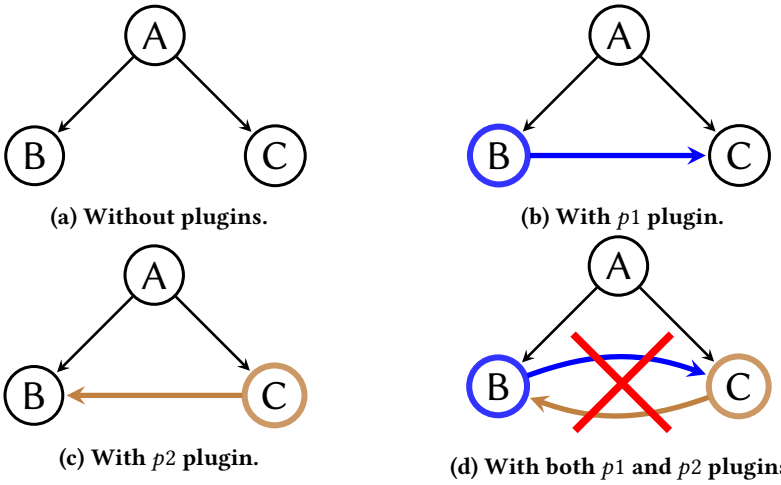
Table 7.1: PQUIC API exposed to pluglet bytecode.

very dependent on 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, leading to undefined behavior. Therefore, our proposed PQUIC interface abstracts the implementation internals from the pluglets, making them compatible with different PQUIC versions or implementations. From a practical viewpoint, core protocol structures are made available to the POE (connection, packet, path,...) as opaque tags. Pluglets can then access a specific structure field by providing both the opaque tag and a well-defined key to the getters and setters. In addition, our interface 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[4].

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 an approach provides algorithmic $\Theta(1)$ time memory allocation while limiting fragmentation which — from our experience — considerably affects the performance of plugins.

Retrieving data shared by pluglets. Pluglets from the same plugin might need to access a common data structure. It might be desirable for pluglets to attach a plugin-specific structure to a given core protocol structure, e.g., to mark a specific packet with a special value. PQUIC enables pluglets to associate some plugin-specific metadata with a specific identifier to a given core protocol structure.

[4] As of current writing, we do not expose TLS keys to plugins.



(a) Without plugins.

(b) With $p1$ plugin.

(c) With $p2$ plugin.

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

Figure 7.4: 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.

Modifying connection memory area. Plugins might need to modify memory outside the POE. For instance, a pluglet might need to write a new frame inside a buffer maintained by the PQUIC implementation. Our API keeps control on the plugin operations by checking the accessed memory areas.

Calling other protocol operations. This feature is required when a protocol operation depends on another one. However, such capability raises potential safety issues. As plugins can potentially 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. Nevertheless, ensuring this property for any combination of loop-free plugins is difficult to assess before executing them due to the combinatorial state explosion. Consider the example shown in Figure 7.4. There are three protocol operations A, B, and C, all guaranteed to terminate. Even if both $p1$ and $p2$ plugins are legitimate and do not create any cycle, their combination might introduce an infinite loop, as shown in Figure 7.4d. To avoid this situation, a PQUIC implementation keeps track of all the currently running protocol operations in the call stack. From an implementation viewpoint, this check can be done in $\Theta(1)$ time by dedicating a running bit in the protocol operation structure. The bit is set when invoking the protocol operation — before calling the PRE anchors — and reset once it completes — after all POST pluglets returned. 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 $x\%$ of the available congestion window. A deficit round-robin then distributes the remaining budget among the plugin frames.

7.1.4 Interacting with Applications

We showed how plugins can cooperate within PQUIC. Plugins can also interact with the application served by PQUIC. This allows them to extend the application-facing interface of PQUIC to bring new functionalities. For instance, a plugin could implement a message mode for QUIC to supplement the standardized ordered byte-stream abstraction [QUIC-T]. Another example would be a plugin proposing an application-driven Multipath QUIC path manager similar to the extended Multipath TCP socket API [HB16]. This communication channel 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` one, but it makes the protocol operation only executable by the application. This allows it to directly invoke new methods, e.g., queuing a message to be sent or requesting the usage of a path on a given 4-tuple. Second, a plugin can asynchronously push messages back to the application, so that it remains independent of the application control flow.

7.1.5 Reusing Plugins across Connections

The plugin injection involves the creation and the insertion of POEs at their respective anchors and the instantiation of the plugin heap. These remain dedicated to a given connection for its entire lifetime. Once the connection completes, the plugin resources may be freed. Nevertheless, it is likely that plugins get reused on subsequent connections. Furthermore, POEs only depend on the pluglets and are isolated from the connections on which they operate. Therefore, to limit the injection overhead, we introduce a cache storing

the plugin associated POEs and memory. When a new connection injects the same plugin, it can reuse the cached POEs as is, without verifying or compiling the pluglets again. The plugin heap must be reinitialized to avoid leaking information between unrelated connections.

7.2 Exchanging Plugins

The previous Section has described the system aspects of PQUIC and the possibility of extending a PQUIC implementation through protocol plugins that are injected on the local host. As TLS 1.3 is an integral part of QUIC, third parties such as middleboxes or attackers cannot modify the data exchanged over such a connection. Protocol plugins could, therefore, be exchanged over an existing QUIC connection. Accepting remote protocol plugins poses the challenge of establishing the trust in their validity, e.g., their termination.

To address this threat, we propose an open system. Our solution bears a similarity to Certificate Transparency [RFC6962]. Both are using a Merkle Prefix tree [Mer87] as the base for the system log and allow each party to contribute to their global protection only by checking their own safety. Our design enables independent developers to publish their own plugins for which the PQUIC peers' trust in their validity is established by independent plugin validators (PV). Yet, we have fundamental differences from Certificate Transparency in the various roles of the distributed system, and within the construction of the system log itself. As opposed to Certificate Transparency, no party has to track the entire log to keep the users safe. We directly offer to plugin developers an efficient mean — logarithmic in the number of plugins published — to check whether any spurious plugin has been published on their behalf. This prevents the need for a third party monitor to emerge. A second important design choice consists in letting PQUIC peers formulate their safety requirements by combining the PVs they trust. This process allows end-users to pin security requirements as a logic expression. The validity of plugins with respect to this expression can be efficiently checked.

In the remaining of this Section, we first describe how participants can distribute plugin trust (§7.2.1). After enumerating our threat model and the guarantees we need (§7.2.2), we detail our Secure Plugin Management System (§7.2.3). Finally, we describe our extensions to the QUIC protocol to support the exchange of protocol plugins over a QUIC connection (§7.2.4).

7.2.1 Distributing Trust

A simple approach for establishing trust in plugins would be for an application developed by `foo.com` and using PQUIC to only accept plugins from authenticated `foo.com` servers. However, such a technique would prevent

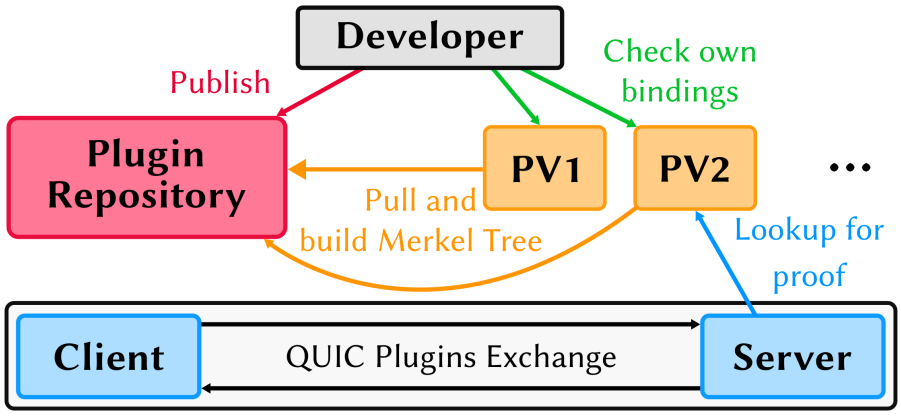


Figure 7.5: Our secure Plugin Management System.

`foo.com` developers to benefit from services provided by third-party servers, e.g., `bar.be`. We go beyond this restrictive approach and propose the open system illustrated in Figure 7.5. It includes four types of participants: (*i*) the plugin developers, (*ii*) the Plugin Repository (PR) that hosts protocol plugins, (*iii*) the plugin validators (PVs) that vouch for plugins validity and (*iv*) the PQUIC peers.

Plugin developers may be independent of the PQUIC implementers. They write plugins and publish them on the PR. Publishing a plugin forms a binding, which we define as the concatenation of the globally unique plugin name with the bytecodes of all its pluglets and the associated manifest, i.e.,

$$\text{binding} = \text{plugin}_{\text{name}} \,||\, \text{plugin}_{\text{manifest}} \,||\, \text{plugin}_{\text{code}}$$

Ensuring the uniqueness of plugin names can be done by hierarchical naming, e.g, a plugin providing multipath and written by `foo.com` can be named `com.foo.multipath`. The PR holds all protocol plugins from all developers and centralizes the secure communication between all participants.

A plugin validator (PV) validates the correct functioning of a plugin. The validation itself depends on the PV capabilities as described in Section 7.5. PVs can obtain the source code from developers willing to ease their validation, but must check that they are able to reproduce the submitted bytecode. PVs can then serve the bytecodes of plugins they validated. The state of our system, i.e., the plugins hosted on the PR and their validation by PVs, progresses on a discrete time scale defined by the *epoch* value. At each epoch, plugins can be added or updated, and each PV can update their plugins validation.

Each PV builds a Merkle Prefix Tree [Mer87] containing the plugins it successfully validated and digitally signs its root, forming a Signed Tree Root (STR). The STR is then sent to the PR. Such distribution enables participants

to fetch the STR even if the related PV is down. A PV can build at most one tree per epoch. We defer the explanation of building the Merkle Prefix Tree by PVs in Section 7.2.3. When the PR is offline, the PVs can serve their own STRs. The absence of a plugin in a tree can be due to two reasons. Either the validation failed or no validation took place at that epoch. The failure cause is communicated to the PR. Plugin developers monitor the validations published by PVs to ensure that the tested plugins match the submitted code.

Before exchanging plugins, PQUIC peers must provide evidence of plugin validity. Our system allows expressing requirements in terms of PV approbation. More precisely, if PV_i is the identifier of a PV, a PQUIC implementation can send a logical formula that expresses its required validation, e.g., $PV_1 \wedge (PV_2 \vee PV_3)$. This design allows the PQUIC peers to precisely express their required safety guarantees.

Our system is distributed. This makes PQUIC peers tolerant to participant failures. In the previous example, if both the PR and PV_3 are offline, then the peer can still rely on PV_1 and PV_2 to validate the provided plugin.

7.2.2 Threat Model and Security Goals

Our distributed system addresses the following threat model. Any participant can act maliciously. Plugin developers may publish malicious code. A PQUIC peer may want to inject illegitimate code on the remote host. PVs may give false assertions on the validity of a plugin. The PR may equivocate on the STRs received from PVs. Both PR and PVs may modify the code served, or impersonate the developers.

Our system offers the guarantee that some aforementioned problems are immediately detected, and the others are eventually discovered. It also ensures that a plugin name securely summarizes its code. Furthermore, a PQUIC peer is always able to identify which party faulted. As a result, given that the PR and PVs can be freely selected by PQUIC peers, we assume that they are willing to protect their reputations, which could be degraded upon discovering problems. In summary, our system covers the following security goals.

Central identities, distributed validation. The PR centralizes the identities of both developers and PVs. A PV can publish its current STR necessary for the *proof of consistency*, and notify developers about plugins that failed its validation. A developer can publish plugins, report PVs equivocations and inconsistencies of its bindings at all PVs.

Non-equivocation. A PV cannot equivocate by presenting distinct STRs to different PQUIC peers. If it does, participants eventually detect this with the

help of others. A STR acts as a tamper-resistant log for all the bindings validated by a PV. We assume that regular checks for non-equivocation of STRs are performed by other participants and reported on the PR. For instance, others PVs could query the STR of a particular PV and compare it with the STRs maintained by the PR. Any PQUIC peer may eventually learn about an equivocation on any received STR.

Secure human-readable names for plugins. When a PQUIC peer wants to use a given plugin, it does not need to reason about developers identity or plugin validity. The name is globally unique, human-readable and unequivocally matches a plugin.

Detection of spurious plugins. If a PV injects a spurious binding, the developer owning the plugin name will be able to detect this and alert PQUIC peers through the PR. A peer may be abused before the detection happens. However, the end-user will eventually know which PV faulted.

7.2.3 System Overview

PVs retrieve plugins from the PR. They build a Merkle Prefix Tree at each epoch containing all successful validated plugins. Each path to the leaf of this binary tree represents a prefix, and bindings are placed in leaves depending on the truncated bits of $H(plugin_{name})$. For instance, in a 3-depth tree, the prefix of the left-most leaf is '000'. Empty leaves are replaced by a large constant value c chosen by the PV. Interior nodes are hashed as $H(h_l \parallel h_r)$ with h_l (resp. h_r) being the hash value of the left (resp. right) child sub-tree.

Leaf nodes contain one or more *bindings*. Several bindings may be located at the same leaf node when the hash prefix of different names $H(plugin_{name})$ collides. In this case, the leaf node contains a linked list of bindings. Under the assumption of uniform hashing, we can engineer the depth of the tree such that a collision happens with low probability, depending on the number of plugins within the PR. Without collision, the leaf node value is defined as

$$h_{leaf} = H(binding)$$

If there is any collision, the leaf node value concatenates the bindings i,j,... as follows

$$h_{leaf} = H(H(binding_i) \parallel H(binding_j) \parallel ...)$$

After having updated its Merkle Tree, the PV digitally signs its root and publishes the STR to the PR where its public-key information is available for all participants. Note that the tree computation is inspired by CONIKS [Mel+15], but our construction differs so that bindings are located in

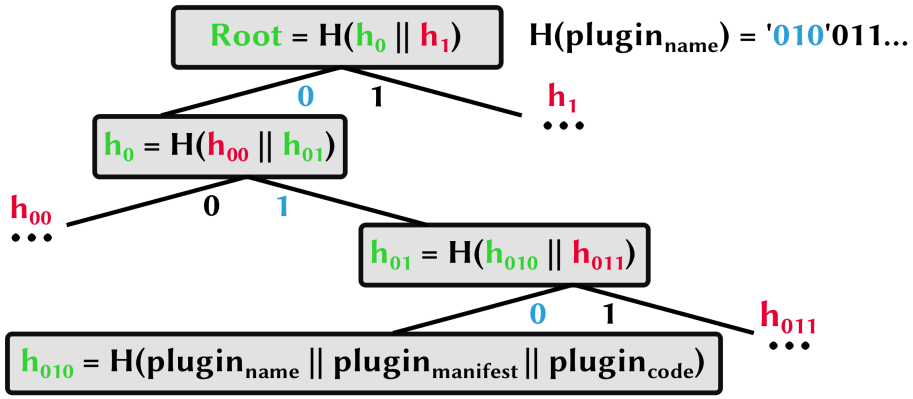


Figure 7.6: Performing the proof of consistency. Red values forms the authentication path, and green values are re-computed to verify that it matches the root.

the tree depending on the hash value of their plugin names, which makes it impossible for a PV to put two bindings for the same plugin and to stealthily populate one with a malicious code.

A PQUIC peer willing to send a plugin over a QUIC connection needs to provide the *authentication paths* from PVs that fulfill this peer's required validation, e.g., $PV_1 \wedge (PV_2 \vee PV_3)$. Obtaining a path only requires sending the name of the plugin to a PV. The PV then computes the *authentication path* in $\Theta(\log(n) + \alpha)$, with $\alpha = n/m$ being the load factor defined from n the number of plugins and m the number of available leaves.

The PV then sends back to the PQUIC peer the authentication path for the binding corresponding to the requested plugin name, as shown in Figure 7.6. The hash values of any other bindings that may be part of this leaf are also sent back by the PV. The PQUIC peer then sends the plugin alongside the corresponding authentication paths obtained for a set of validators matching the other peer's required validation. PQUIC peers can preemptively fetch authentication paths for the plugins they intend to use at each epoch.

Finally, the peer receiving the plugin recomputes the root value from the binding and the authentication path to match the STR cached for the current epoch as pictured in Figure 7.6. If the computed root matches the STR, then the plugin is accepted.

When verifying a binding at a PV, a developer sends the name of the corresponding plugin to the PV. If only a single hashed binding is present in the tree, the developer checks that it matches theirs. If multiple hashed bindings are present, i.e., because of a collision, the developer receives their clear text. This allows them to discern whether the collision was due to a prefix collision

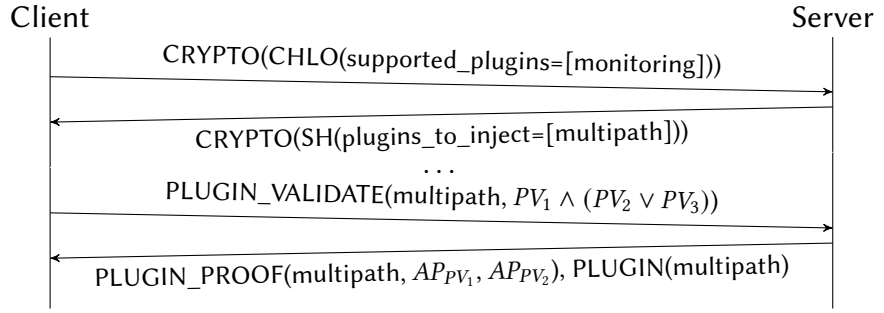


Figure 7.7: Example flow for the exchange of the FEC plugin.

or the PV added a spurious binding.

If the PV tree does not contain the hashed binding of a given plugin, its developer obtains a proof of absence, i.e., an authentication path to the linked-list without the developer's binding or an authentication path to the constant value c indicating an empty leaf such that it matches the truncated bits of $\mathsf{H}(\mathsf{plugin}_{\mathsf{name}})$.

7.2.4 Exchanging QUIC Plugins

Given the flexibility of QUIC, it is relatively simple to modify it to support the exchange of plugins. The QUIC connection establishment packets contain QUIC transport parameters such as the maximum number of streams, the maximum packet size or the idle timeout [QUIC-T]. To enable plugin injection, PQUIC proposes two new QUIC transport parameters: `supported_plugins` and `plugins_to_inject`, both containing an ordered list of protocol plugins' identifiers. The first `supported_plugins` parameter announces the plugins that a PQUIC peer can inject locally. These plugins are stored inside its local cache. The second `plugins_to_inject` announces the plugins a PQUIC peer would like to communicate to the other peer in order to apply them on the running connection. Once the QUIC handshake has completed, both peers have a complete view of the available and requested plugins. Then, there can be two outcomes. Either (*a*), all plugins requested for injection are already available. In this case, they are injected as local plugins, as explained in Section 7.1, in the order described by the `plugins_to_inject` transport parameter. Else (*b*), one or more plugins are unavailable locally, they are then transferred as illustrated in Figure 7.7. In this example, the client announces the support of a monitoring plugin while the server would like to inject a multipath plugin into the client. First, the client announces its required validation formula for missing plugins, hence multipath, with the PLUGIN_VALIDATE frame. Second, the server responds with authentication

paths from PVs that fulfill this formula in a PLUGIN_PROOF frame. The requested plugin is then transferred over the plugin stream in PLUGIN frames, akin to the QUIC cryptographic stream. When receiving the remote plugin, the client performs the check of the proof of consistency. Upon success, it stores the plugin in its local cache. Remote plugins are not activated for the current connection, but rather offered in subsequent connections as part of the locally available plugins. While PQUIC is capable of injecting plugins at any time (as explained in Section 7.1), synchronizing their injection between two hosts raises potential issues. For instance, consider the case of a plugin modifying the format of an existing QUIC frame, e.g., the multipath plugin adding a Path ID field at the beginning of the ACK frame. How can QUIC hosts agree when they should expect to parse the Path ID field in the incoming ACK frames? We prevent these issues by this conservative choice.

While the exchange mechanisms introduce some overhead, we believe it remains acceptable. The fixed cost of exchanging plugin bytecodes is only present during the first connection, as subsequent ones will take advantage of the PQUIC caching system described in Section 7.1.5. Furthermore, if the plugin is not mandatory for the use of the application (e.g., adding the multipath extension to better aggregate the available network links), the plugin exchange does not prevent data from being transmitted over the connection. Indeed, data and plugin streams can be concurrently used thanks to the QUIC frame multiplexing.

7.3 Exploring Simple Protocol Tuning

Before digging into large plugins, we first consider here simple plugins composed of only one pluglet inserted in REPLACE mode modifying specific algorithms. For this, we first implemented a proof-of-concept prototype based on our mp-quic implementation supporting Multipath GQUIC to assess the feasibility of our solution. Our plugins are written in C and we use the `clang` and `llvm` tools to produce the eBPF bytecode which can be injected inside our QUIC implementation.

To make the mp-quic implementation pluggable, we inserted (*i*) a map of plugins for each connection context with a dedicated API to inject code, (*ii*) an API implementing anchors, testing if a plugin is present to run it and processing the shared context, and (*iii*) code integrating the eBPF VM as a library. Notice that this version only integrated the POE (§7.1.1) thanks to the previously presented user-space eBPF VM written in C [IO 18]. This prototype thus does not integrate the generic API presented in Section 7.1.3 (as it relies on a ad-hoc, protocol operation specific structure stored in the shared memory), but helped to design it.

We use mp-quic to demonstrate that protocol plugins can be pushed in

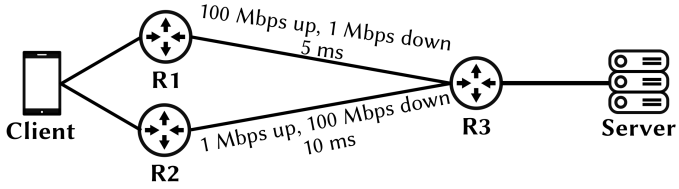


Figure 7.8: The network used for the ACK tuning experiment.

an implementation written in a different language and explore multipath use cases. To enable the eBPF VM to interact with mp-quic, we use `cgo`. This standard Go package allows calling C code from a Go program. However, it requires data to be copied into a particular structure passed to `cgo` and eventually to the VM.

In the remaining of this Section, we perform our experiments in the Mininet environment [Han+12]. We first show how a plugin can tune the Multipath QUIC ACK scheduling strategy (§7.3.1). We then demonstrate the dynamic loading of pluglets by limiting the sending rate (§7.3.2). Finally, we detail the overhead of protocol plugins in our prototype implementation (§7.3.3).

7.3.1 Tuning ACK Frame Scheduling

Our first multipath use case is a scenario with asymmetric bandwidth shown in Figure 7.8. In this environment, it would be beneficial to send data over the bottom path and the acknowledgments over the upper one. This is possible with QUIC since frames are independent of the packet carrying them in contrast with Multipath TCP where acknowledgments must be sent on the same path as data. However, as stated in Chapter 6, changing the acknowledgment strategy of Multipath gQUIC can affect the performance since latency measurements would be affected. Protocol plugins can be injected at remote side to ensure a consistent acknowledgment strategy.

Consider the network shown in Figure 7.8 with a 20 MB bulk download. Sending acknowledgments on the same path as data is not the best strategy, as acknowledgments would saturate the R2-R3 link. With this strategy, Figure 7.9 show that the client needs between 6 and 10 seconds to download the 20 MB file from the server. We then implement a pluglet of 30 lines of code that attaches to the SelectACKPath operation on the client to force it to send all acknowledgments on the upper path. Our results show that this strategy significantly reduces the download times as acknowledgments are returned quickly to the server and they do not saturate the upper link.

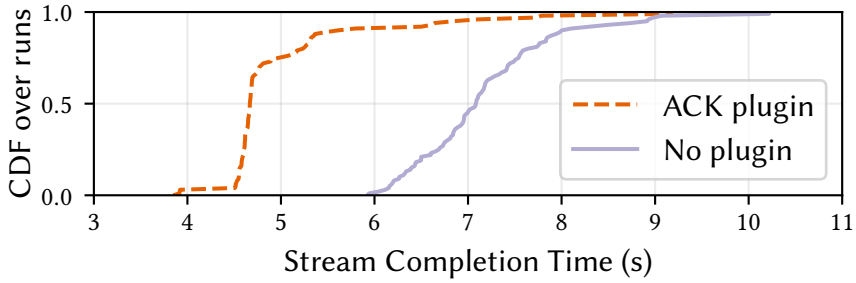


Figure 7.9: When plugin code forces ACK frame sending on the R1-R3 path, the transfer completes quicker.

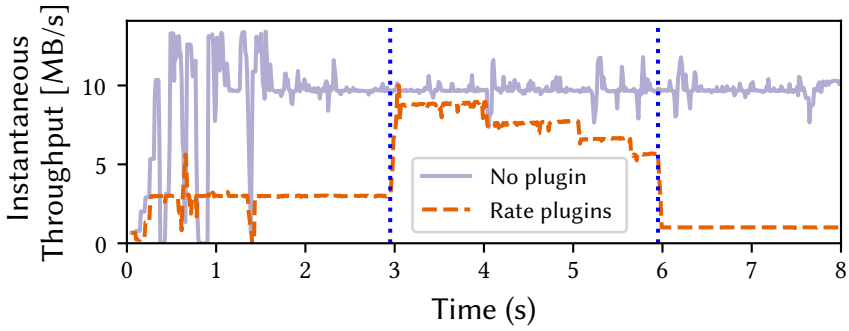


Figure 7.10: Three plugins, loaded at different times, controlling the rate over an expensive path.

7.3.2 Restricting the Pacing Rate

Our second use case is inspired by smartphones. Recall from Chapters 2 and 3 that Multipath TCP enables them to simultaneously use both the Wi-Fi and the cellular networks. However, many users have volume caps on their cellular plans and do not want to use the cellular network for large downloads when Wi-Fi is available.

In this situation, the smartphone needs to control the operation of the server to restrict the bandwidth consumed on the cellular network. However, a server cannot easily infer what is the desired bandwidth cap over a given path. We address this problem with a protocol plugin that is pushed by the smartphone to the server at the beginning of the connection. This plugin is attached to the congestion control mechanism on the server and throttles its congestion window over the cellular path, but not over the Wi-Fi one. By writing only 10 lines of C code, our plugin allows this behavior. Figure 7.10

eBPF code	cgo no-op	eBPF no-op	Preparing structure	Empty structure to cgo	Empty structure to eBPF
-25.1 %	-4.9 %	-6.1 %	-8.9 %	-21.7 %	-21.7 %

Table 7.2: The performance difference of mp-quic with the pluginized packet scheduler compared to native Go code.

shows the effect of three different plugin versions over a two-path scenario with 80 Mbps and 50 ms RTT links. The first one, injected after the handshake, limits the rate to 3 MB/s. The second one, inserted after three seconds after removing the first one, caps the speed to 10 MB/s. Finally, the last one, added after six seconds after dropping the previous one, constraints the link to 1 MB/s. Our results presented in Figure 7.10 confirm the effect of these plugins and demonstrate that they can be injected at any time on a connection after the handshake.

7.3.3 The Cost of Protocol Plugins

The proposed protocol plugins come with a trade-off between customization and performance. Two factors influence their impact. First, there is a network overhead to exchange protocol plugins on a per connection basis. Second, there is some computational overhead to call plugin code instead of native one.

Network Overhead. The transmission of the executable bytecode generates additional load on the network. With the eBPF user-space VM used in our prototype, the executable codes are ELF files containing metadata (sections header, strings table, symbols table,...) in addition to the raw eBPF bytecode. Our two simple use cases presented here introduce plugins requiring only a few lines of C code, leading to ELF files of 1 KB and 512 B for the ACK scheduling and the pacing rate, respectively. With very simple plugins, this network overhead is negligible for video streaming applications or the download of large web pages.

Computational Overhead. Executing the protocol plugins inside a VM provides portability and isolation, but is slower than native code. Furthermore, the interactions with the instrumented implementation can also introduce some additional load. To quantify this overhead, we benchmark our mp-quic implementation by measuring the throughput on localhost when transferring 50 MB. This is a standard benchmark used by the quic-go implementation.

The packet scheduler is a critical component for multipath protocols [Paa+14] and previous works proposed to make it flexible to application needs [Frö+17]. We consider the lowest-latency packet scheduler, both in native Go code — no plugin — and as a plugin. This plugin weights around 3 KB for 107 lines of C code. The packet scheduler is a critical element of the sending workflow, as it is called at each packet transmission occasion. Table 7.2 indicates that in our mp-quic implementation, this plugin reduces the achieved throughput by 25% compared to its native Go variant. As previously explained, cgo bridges the Go implementation with the eBPF VM. To communicate with the eBPF VM, mp-quic must first prepare a structure that exposes the connection context to the VM. As the core mp-quic implementation cannot predict which fields of the connection context will actually be used, it must put all of them in the structure. This consumes CPU times. If the scheduler in native Go code also fills such structure, its performance drops by 9%. This result demonstrates the value of proposing an API of getters and setters to the VM. Nevertheless, the main overhead comes from the stack switch induced by cgo. Its impact depends on the size of the interface structure. Running the native Go scheduler with a cgo no-op without any communicated data decreases the performance by 5%. Doing the same with a 8 KB communicated structure lowers the performance by 22%. In such cases, we do not observe any difference between cgo and eBPF no-ops. We could mitigate the performance impact of cgo by using a user-space eBPF VM written in native Go language. However, to our knowledge, such software was not available at the time we made our mp-quic prototype.

7.4 Building Complete Extensions with Plugins

Building on our previous experiences with our early mp-quic prototype, we now focus on a much more complete implementation of PQUIC based on the picoquic one [Hui18] written in C. It fully implements all the mechanisms presented in Section 7.1 as well as the plugin exchange described in Section 7.2.4. We first discuss some implementation details about how we make picoquic pluginizable (§7.4.1). Then, we describe how we managed to fully implement the multipath extensions using only plugins (§7.4.2) and evaluate its performance against a native Multipath QUIC implementation (§7.4.3). This approach enables hosts to provide different multipath-specific algorithms depending on the application requirements (§7.4.4). We also demonstrate that our approach allows PQUIC hosts to combine orthogonal plugins (§7.4.5). Finally, we conclude this Section by analyzing the overhead brought by protocol plugins (§7.4.6).

7.4.1 Making the Implementation Pluginizable

While the core ideas of PQUIC have been discussed in Section 7.1, they require a specific implementation design that we explain now. In particular, we describe (*i*) how we seamlessly call protocol operations regardless of their behavior being provided by pluglets or not, (*ii*) how we provide a generic interface to pass arguments to the POE, and (*iii*) how we handle nested protocol operation calls without any interference.

Refactoring Implementation Functions with Pointer Indirections

A naive solution to make a specific part of the implementation customizable by a pluglet is to adopt a similar approach as our early prototype described in Section 7.3 by inserting explicit checks — `if/else` — for a POE at specific code locations. Such an approach suffers from at least two drawbacks. First, it prevents plugins from adding protocol operations that were not already present in the base implementation. Second, it requires that built-in functions must be aware if a pluglet is present or not to run their behavior. The presence of pluglets should not be the concern of built-in, pluginizable functions.

Instead, our PQUIC implementation replaces native function calls by protocol operations identified by human-readable names. From a concrete viewpoint, each connection context includes a hash table making the link between the string identifier and an internal structure representing the protocol operation. When a connection is created, the PQUIC implementation registers all the protocol operations. The address of the built-in function is stored in the internal protocol operation. The awareness of the presence of POEs is thus delegated to this protocol operation structure. When a plugin defines a new protocol operation, the PQUIC implementation creates a new entry in the connection-related hash table, hence achieving the requested design. Notice that computing the hash of a protocol operation's identifier takes time — about 100 μs with our implementation. To mitigate its impact, we precompute the hash value at host initialization such that the core implementation can reuse it without further processing.

A Generic Interface for the POE

Distinct protocol operations often require different inputs and outputs. However, our user-space eBPF VM only supports pluglets taking only one argument. In our early `mp-quic` prototype, we used it to communicate the pointer to a protocol operation specific structure containing the fields of interest. Yet, this sets the issue of a generic interface between the core implementation and POEs, especially when defining new protocol operations. In addition, this raises the challenge of memory access management by plugins — as both

the content of the structure and the plugin heap should be accessible by the pluglet.

To solve these issues, the only argument provided to the pluglet is an opaque tag referencing the connection context. Although this tag actually corresponds to the related structure pointer, the pluglet cannot dereference it as it points to a memory area outside its allowed bounds. The pluglet then uses the provided tag as an argument to the getters and setters to retrieve the variables of interest. All the values handled by these getters and setters are `protoop_arg_t`, which actually correspond to `uint64_t`. When a plugin wants to access a specific structure, it first fetches its tag and then uses it in the structure-dedicated getter and setter. To retrieve the protocol operation arguments, the plugin uses a specific connection's getter with the index corresponding to the desired input. Similarly, a dedicated setter enables pluglets to communicate output(s) to the caller. Such a design ensures a common interface for all plugins while focusing the memory access management only to the plugin heap.

Handling Nested Protocol Operation Calls

Because all the codes defining the behavior of protocol operations — either built-in or provided by a plugin — use the common API to fetch inputs and provide outputs, there is a need for coordination to avoid interference between different protocol operations. In particular, a pluglet might call another protocol operation before fetching a given input or might generate an output before calling another protocol operation. To keep this isolation between different inputs and outputs, our central `plugin_run_protoop` function saves all the previous inputs and outputs on its stack before preparing the new ones for the protocol operation to be called. Once it returned, the inputs and outputs previously stored on the stack are restored and the pluglet continues its execution without noticing any change in its inputs or outputs. Notice that it gets the outputs of the called protocol operation in a structure it used to call it. In addition to this argument processing, our `plugin_run_protoop` function also checks for protocol operation call loops by setting (when calling) and resetting (when returning) a specific bit in the protocol operation structure.

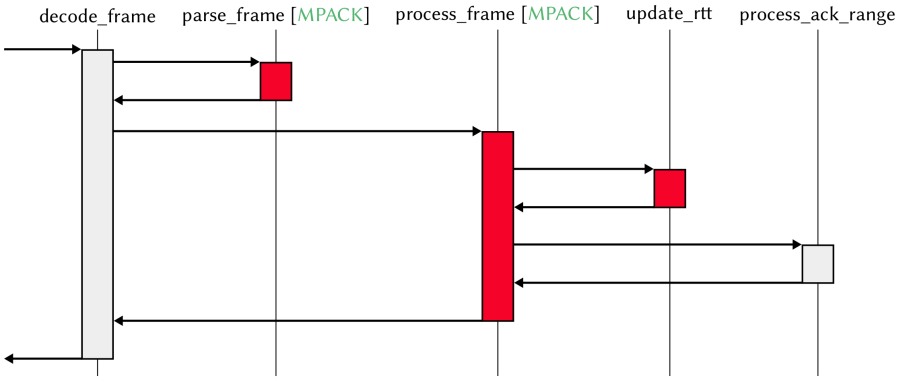
7.4.2 Implementing the Multipath Extensions with Plugins

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]. In this Section, we rather focus on the much more complex

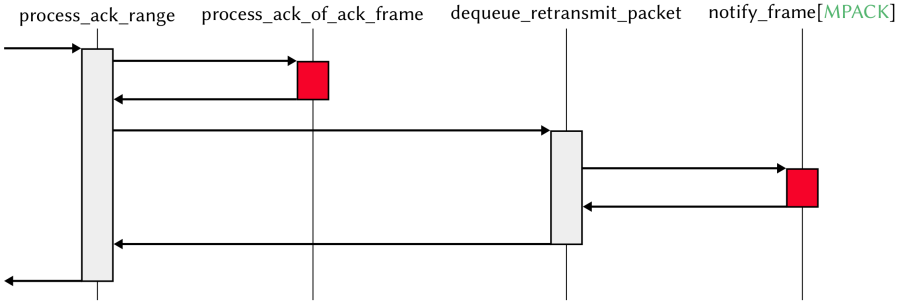
Multipath extensions for the QUIC protocol described in Chapter 6 to demonstrate the flexibility and the extensibility of PQUIC. Other use cases have been considered too, we invite the interested reader to look at our SIGCOMM 2019 paper for further detail [De +19]. From a concrete viewpoint, our Multipath plugin consists in 26 pluglets injected in 14 distinct protocol operations. Figure 7.11 illustrates most of the injected pluglets and their insertion point. To understand the design of our Multipath plugin, let us consider several common code paths.

First, upon reception of a packet containing frames, a PQUIC implementation parses its header mentioning the Connection ID. Because the base `picoquic` implementation supports connection migration (but not the simultaneous usage of multiple paths), it embeds the notion of path. Since the Connection ID implicitly indicates to which incoming path the packet was sent, we injected a pluglet in REPLACE mode to the get_incoming_path protocol operation as illustrated in Figure 7.11f. After decrypting the packet, the PQUIC implementation starts decoding each frame along with the related connection and incoming path contexts. QUIC frames always start with their Type variable-length integer field. This type acts as an identifier of the decoded frame. We split the frame decoding process into two protocol operations as pictured in Figure 7.11a. First, the parse_frame one interprets the frame's raw bytes and returns a C frame-specific structure containing all the frame's fields. Second, the process_frame one takes the output of the parse_frame protocol operation as input and performs actions depending on the actual content of the frame. This distinction between the parsing and the processing of frames — although not strictly required — aims at keeping focused protocol operations to avoid very large and complex pluglets. In addition, since our implementation stores sent packets as raw bytes, it requires parsing again its frames without doing any further processing. In our current Multipath plugin, we implemented ADD ADDRESS, ACK and NEW CONNECTION ID frames. Notice that the two last ones, which are actually small modifications of the original ones, are implemented with their own type (MPACK and MP NEW CONNECTION ID). This induces the addition of two pluglets (one in parse_frame, the other in process_frame) for each frame.

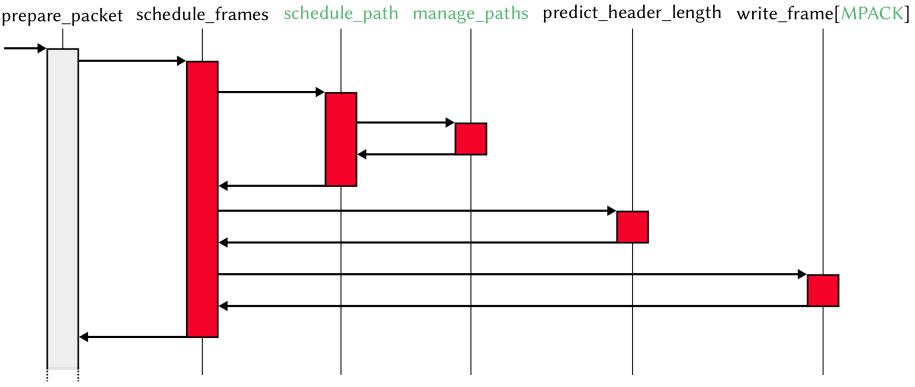
While the processing of the ADD ADDRESS and MP NEW CONNECTION ID frames does not involve additional protocol operation calls, the one of the MPACK frames includes other ones. When receiving an acknowledgment, the host can estimate the network latency. Therefore, the processing of MPACK frames requests the update_rtt protocol operation. The Multipath plugin inserts a pluglet in REPLACE mode to make path-aware RTT estimations. Then, it handles the acknowledged packets through the process_ack_range protocol operation. This one first adapts the range that it communicates through its MPACK frames through the process_ack_of_ack_frame protocol



(a) Handling the reception of a frame (MPACK case).

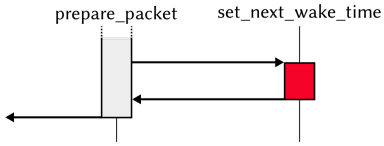


(b) Processing an ACK range (continuation of Figure 7.11a).

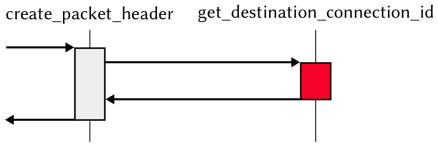


(c) Sending a QUIC packet.

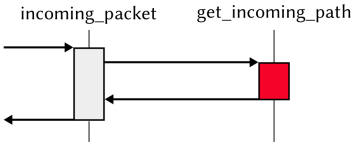
Figure 7.11: The modified code path to include the multipath extensions only with plugins. Added protocol operations and parameters are shown in green while pluglets inserted in REPLACE and POST modes are respectively represented in red and blue.



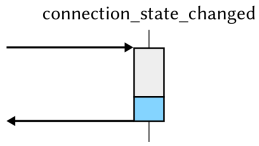
(d) Sending a QUIC packet (continuation of Figure 7.11c).



(e) Creating an header of a packet to be sent.



(f) Identifying the receive path of an incoming packet.



(g) Being notified that the connection state is ready.

Figure 7.11: The modified code path to include the multipath extensions only with plugins. Added protocol operations and parameters are shown in green while pluglets inserted in REPLACE and POST modes are respectively represented in red and blue.

operation. Under the hood, this last operation parses again the sent packets' raw bytes — hence calling parse_frame — and is updated by a pluglet to make it aware of MPACK frames. Finally, based on the received ACK range, the implementation can go through its sent packets buffer thanks to the dequeue_retransmit_packet protocol operation to determine if the packet can be freed or otherwise its content should be retransmitted. In any case, our PQUIC implementation informs plugins about the (non) delivery of their reserved frames through the notify_frame protocol operation. Again, a pluglet in REPLACE mode is inserted for each implemented frame.

Now that we completed the packet reception's code path, we focus on the sending one shown in Figures 7.11c and 7.11d. At some point, the code flow calls the prepare_packet protocol operation which itself requests the schedule_frames one. It determines which frames will be included in the next packet to be sent. Because the base PQUIC implementation only uses a single path, our Multipath plugin injects a pluglet in REPLACE mode allowing the usage of others paths. In particular, we dedicate the selection of the path to a new protocol operation schedule_path. With such decomposition, PQUIC hosts can change their path selection algorithm without considering the frames that will be put on the selected outgoing path. The Multipath plugin also introduces another protocol operation, manage_paths, associating the available paths to specific 4-tuples. In our current version, the schedule_path protocol operation always calls the manage_paths one, but it can also be extended to listen to specific events, such as the loss of an IP address.

Once the sending uniflow has been selected, the schedule_frames protocol operation estimates the available packet space for frames by calling the predict_header_length one. As each sending uniflow has a distinct Connection ID, with possibly different lengths, we put in a pluglet in that protocol operation. Once a frame has been selected for sending, it calls the write_frame protocol operation with its type as a parameter. Again, each of our four implemented frames has its own dedicated pluglet. Once the content of the packet has been determined, the prepare_packet protocol operation relies on the set_next_wake_time one to determine when the timer should fire. We adapt its operations with a pluglet to make it aware of the different available paths. Finally, just before sending the packet to the network, it requires writing its header through the create_packet_header protocol operation. As each sending uniflow has a different Destination Connection ID, we insert a pluglet at the get_destination_connection_id protocol operation, as shown in Figure 7.11e. Notice that since `picoquic` actually sends packets using the `sendmsg` call — filling its auxiliary data with the desired packet's 4-tuple from the path context — there is no need for the Multipath plugin to tweak it.

The last piece of our Multipath plugin resides in knowing when PQUIC hosts can start using several paths. As explained in Chapter 6, PQUIC peers must first complete the handshake before beginning multipath operations. This is why we insert a last pluglet in POST mode to the connection_state_changed protocol operation as pictured in Figure 7.11g. Once the connection becomes ready, our pluglet triggers the sending of both ADD ADDRESS and MP NEW CONNECTION ID frames, hence enabling the use of multiple paths.

Together, all these pluglets form a PQUIC plugin of 2750 lines of C code that provides basic multipath capabilities following the requirements enumerated in Chapter 6. Our plugin supports the exchange of Path Connection IDs and host IP addresses. It then associates a sending Uniflow ID between each pair of host addresses, similar to the `full-mesh` path manager used by Multipath TCP. We implement both the round-robin and the lowest latency first schedulers.

7.4.3 Evaluating the Multipath Plugin

We evaluate the performance of our Multipath plugin in a lab equipped with Intel Xeon X3440 processors, 16 GB of RAM and 1 Gbps, running Linux kernel 4.19 and configured as shown in Figure 7.12. The links R_1-R_3 and R_2-R_3 are configured using `netem` [Hem+05] to add transmission delays and using `htb` to limit their bandwidth. One-way delay `d` is expressed in milliseconds and bandwidth `bw` in Mbps.

To evaluate our plugin in a wide range of environments, we reuse the experimental design approach [Fis35]. We define ranges on the possible values

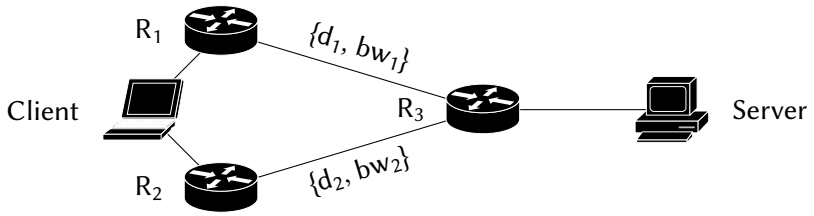


Figure 7.12: Network topology used for experiments.

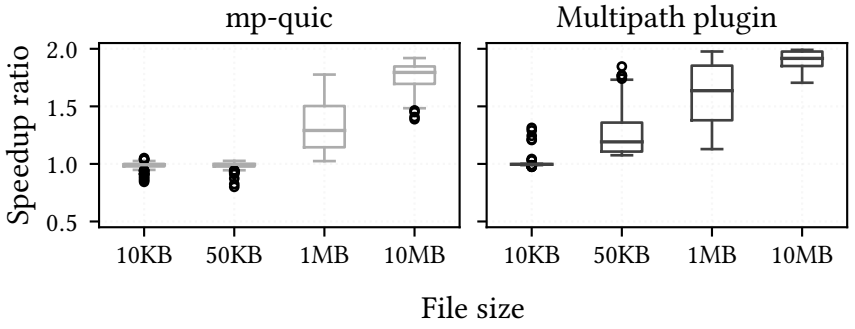


Figure 7.13: Over two symmetric network paths, multipath tends to complete transfers twice faster than single-path.

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. We use the parameter range $\{d_1 \in [2.5, 25]$ ms, $bw_1 \in [5, 50]$ Mbps, $d_2 = d_1, bw_2 = bw_1\}$, and assume that both links have similar bandwidth and delay characteristics. Note that when links are lossless, congestion-induced losses can still be observed due to the limited bandwidth and router buffers, set to the bandwidth-delay product. Our plugin is cached on both the client and the server.

For both single path and multipath settings, we record the time between a GET request issued by the client and the reception of the last byte of the server response. We then compute the ratio of the single path completion time over the multipath one to obtain the speedup ratio. We observe its evolution with the size of the requested file and compare it with the ratio obtained using the native mp-quic implementation presented in Chapter 5. Figure 7.13 shows that with small files, there is little gain in using two paths. This is not surprising since each path is constrained by its initial congestion window. Notice that the initial path window of mp-quic (32 KB), inherited from `quic-go` [CS18],

is twice the default one of PQUIC (16 KB). This explains the small speedup gain of our plugin on 50 KB files. With larger files, both mp-quic and our Multipath plugin efficiently use the two available paths. The speedup ratio of both the native mp-quic and our Multipath plugin tends to reach 2 with 10 MB files.

7.4.4 Adapting the Multipath Plugin to Interactive Use Cases

Providing tuned multipath-specific algorithms to dedicated use cases is one of the key motivations to distribute the Multipath extensions as plugins. Consider the case of packet schedulers. When network paths exhibit similar delays and bandwidths, a simple round-robin scheduler works well when performing bulk transfers. Once different delays are observed, a lowest-latency first one is more suitable. However, when the application requires a low-latency service within a wireless environment, solutions such as MultiMob (§3.3) are more appropriate.

Our PQUIC approach enables hosts to use all these different scheduling strategies without modifying their implementation. From a concrete viewpoint, our Multipath plugin is split into two parts. The first one contains the core pluglets providing the support for the extensions. The second one consists in the multipath-specific algorithms — here the path manager and the packet scheduler. While the first part always contains the same pluglets, a host can select the most appropriate multipath-specific algorithms in the plugin it negotiates with its peer.

To demonstrate the benefits of this approach, let us reconsider the scenario presented in Section 5.2.5 where a delay-sensitive application generating request/reponse traffic loses its lowest latency path (15 ms RTT) after 3 seconds (using `tc netem loss 100%`). We evaluate two variants of the Multipath plugin. The first one embeds the lowest latency first path scheduler at both client's and server's sides and prefers using the initial network path (15 ms RTT) over the additional one (25 ms RTT). The second variant still uses this lowest latency first scheduler at client's side but makes the server follow the client's scheduling decisions similar to the MultiMob's server-side packet scheduler (§3.3.1). In practice, it records the remote address of the last received packet and forces the server to send the next packet on a sending path reaching the given remote address. Each 1400-byte request sent every 400 ms triggers a 750-byte response. Unlike our previous experiments, each request/response takes place in its own stream, meaning that a delayed response does not impact subsequent requests. Notice that our Multipath plugin does not implement the PATHS frame, i.e., a host noticing a path failure does not advertise it to its peer.

We perform our experiments in the same lab as in Section 7.4.3. Fig-

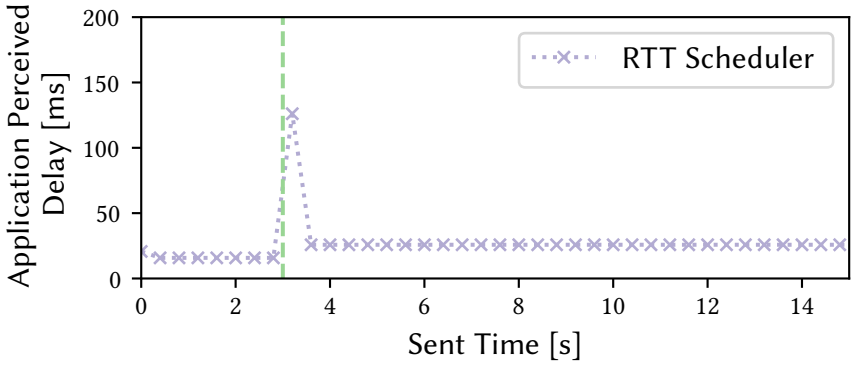


Figure 7.14: A server-side packet scheduler following the client's decisions prevents the server from suffering a RTO too.

ure 7.14 indicates that before the loss event, the two experiments uses the lowest latency network path. Notice that the initial request observes a response delay of 20 ms because the client still limits its packet size to 1252 bytes to avoid fragmentation as it did not send a path MTU discovery packet yet. Hence, the request is split into two packets. Since the latency scheduler wants to test the additional path, the complete request arrives at the server after 12.5 ms, which replies on the fastest path[5]. When the loss event occurs, it takes 50 ms for the client to notice that the preferred path is potentially failed and duplicate the frames over the additional path. While the server-side scheduler follows the last client's decision and directly replies on the additional path, the lowest latency first scheduler also experiences a RTO at server side after 50 ms. Then, both Multipath variants continues the traffic over the working additional path.

7.4.5 Combining Orthogonal Plugins

Given the isolation provided by PQUIC, it is possible to load different plugins on a given PQUIC implementation provided that they do not both replace a same protocol operation. To demonstrate this design flexibility, we show how our Multipath plugin can be injected together with another one providing unreliable messages. This last plugin relies on the EXTERNAL anchor presented in Section 7.1.4 to let the application request the unreliable delivery of data. Their combination typically suits a QUIC-based Multipath VPN application.

[5]The server-side scheduler also behaves the same way but for different reasons. At the time it fully receives the first request, the additional path is not validated yet, only leaving the fastest one for the response.

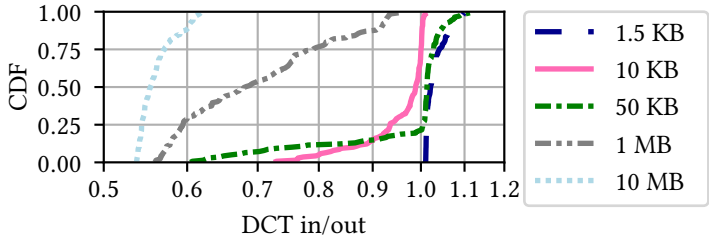


Figure 7.15: Download Completion Time ratio of TCP inside and outside a multipath client-server tunnel.

Benchmark	Native	eBPF JIT-compiled	eBPF Interpreted
Simple loop	839 ms	1706 ms	17000 ms
Get / Set	844 ms	5336 ms	40150 ms

Table 7.3: Execution time for the micro-benchmarks (median over 3 runs).

We evaluate the performance of combining these two plugins in the same network scenario explored in Section 7.4.3. Our scenario creates Linux tunnel interfaces at both client and server sides to carry IP datagrams inside a PQUIC connection. We measure the ratio of Download Completion Time (DCT) when running a single 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 tunneling a short TCP transfer, as pictured in Figure 7.15. However, as file size grows the benefits of using multiple paths become clear. By spreading the traffic over the two symmetric paths, our combined plugins reach a DCT ratio that tends to 0.55. This shows that our plugins enable TCP to aggregate both paths' bandwidth without being aware of the multipath usage.

7.4.6 Plugin Overhead

The balance between flexibility and performance is a classical trade-off. Google and the IETF have decided to run QUIC over UDP despite the fact that Google measured that *"QUIC's server CPU utilization was about 3.5 times higher than TLS/TCP "* [IS18]. The performance gap between TCP and UDP has since been slightly reduced, but UDP remains slower [BD18]. As PQUIC delegates the execution of the plugins to the POE, there is a processing overhead due to the Just-In-Time (JIT) compilation, the run-time verifications performed by our monitor at each memory access, and the utilization of the API to safely access PQUIC state variables.

Executing code in the POE is less efficient than running native code. To assess this, we write a simple for loop performing at each iteration a sum,

Plugin	$\tilde{x}$ Goodput	$\sigma/\tilde{x}$	$\tilde{x}$ Load Time
PQUIC, no plugin	1104.2 Mbps	3.8%	0.0 ms
Monitoring (a)	1037.3 Mbps	4.6%	6.35 ms
Multipath 1-path (b)	756.6 Mbps	3.1%	8.28 ms
a and b	714.2 Mbps	4.4%	13.00 ms

Table 7.4: Benchmarking plugins over 10Gbps links (20 runs).

a multiplication and a division on a 64-bit integer and evaluate it on a laptop running on a Intel Core i7-4810MQ CPU @ 2.80GHz with 16 GB of RAM. We implement it both as a plugin and in native C code and run the loop one billion times. Table 7.3 indicates that the POE is twice slower than native code. Such overhead could probably be mitigated by using a more optimized VM with a smarter JIT compiler. We also evaluate the impact of our get/set API by writing a for loop getting and setting two integer fields at each iteration. Running the loop 500 millions times, we observe that our get/set API is six times slower compared to direct memory accesses. Notice that in all our experiments, we consider JIT-compiled bytecode. Our user-space eBPF VM also enables us to run it in interpreted mode. However, on a computationally intensive benchmark, the interpreted mode is ten times slower than JIT-compiled one.

To observe this performance impact in more bandwidth-intensive environments, we benchmark our PQUIC implementation by measuring the completion time of a 1 GB download between two servers with 10 Gbps NICs. We initially performed our analysis with our lab described in Section 7.4.3, but we noticed that the main CPU overhead came from the TLS encryption and decryption of packets. This is because our initial processors — Intel Xeon X3440 — do not provide hardware support for AES encryption. Hence, we relaunch our experiments with servers equipped with Intel Xeon E5-2640 v3 CPUs. These benefit from newer instruction sets — notably AES-NI — and allow us to focus on the plugin execution overhead. Note that our PQUIC implementation is single-threaded and thus does not benefit from the additional CPU cores.

Table 7.4 reports the median achieved goodput, its relative variance and the plugin loading time for each plugin. PQUIC achieves a median goodput of 1104.2 Mbps. This low performance compared to TCP is partly due to the fact that `picoquic`, the implementation on which PQUIC is based, has not been optimized yet for performance. Indeed, as the QUIC specification was still evolving when we built our implementation, there was limited interest in deeply optimizing a QUIC implementation at that stage. Comparing PQUIC to the version of `picoquic` we based our work on could allow measuring

the impact of adding our approach to a QUIC implementation. However, this is technically challenging, as our PQUIC version also contains performance and bug fixes to `picoquic`[6]. There is thus no `picoquic` version matching the current state of PQUIC. We could also compare the latest version of `picoquic` with PQUIC. Nevertheless, both `picoquic` and the QUIC specification itself have substantially evolved during our work, adding many degrees of freedom that would confuse the performance evaluation.

Since QUIC encrypts everything, it is difficult for network operators to troubleshoot their connections. We propose a protocol plugin [De +19] — consisting of 500 lines of C code spread into 14 pluglets — that attaches bytecodes at strategic PRE and POST anchors to monitor the state of running PQUIC connections. When evaluating this monitoring plugin, we observe a goodput reduction of 7% which matches the additional 8% CPU instructions executed. We also observed a 10% increase of TLB load misses caused by the context switches between PQUIC codes and the POEs. Given that the monitoring code complexity is low, these results illustrate the overhead of adding several pluglets within the critical path.

The benchmark of the Multipath plugin over a single path achieves a median goodput of 756.6 Mbps. Several factors explain this reduction. First, the acknowledgments are created by the plugin using MPACK frames, which constitutes a significant part of the client execution time. Second, the Multipath plugin provides a path manager and a path-aware frame scheduler. This adds several new protocol operations into the packet processing code path.

Combining both the monitoring and the Multipath plugins results in a 6% goodput reduction compared to multipath only. This demonstrates that plugins with orthogonal features are efficiently combined using PQUIC.

Inserting plugins takes time, as described in the last column of Table 7.4. This time is proportional to the number of inserted pluglets and their complexity. The instantiation of POEs (between 4 and 7 ms) is the major contributor to this loading time. Note that this overhead is only present when there is no cached plugin available. If the host previously loaded the plugin in a completed connection, it can reuse its POEs as described in Section 7.1.5 to load the plugin in less than 30 μs.

Plugins can be exchanged between PQUIC peers over the network as described in Section 7.2.4. To limit the exchange overhead, we rely on a ZIP compression scheme to transfer plugins. We take advantage from the fact that pluglets from a given plugin can contain duplicate code from common functions. Compressing the Multipath plugin reduces that exchanged overhead from 138 KB (plain ELF files) to 40 KB (compressed ones). Assuming the plugin exchange starts at the beginning of the connection with an initial

[6] Actually, our current version of PQUIC achieves higher goodput than the original version of `picoquic` we base our work on.

congestion window of 16 KB, it requires 2 RTTs to be exchanged.

7.5 Validating Plugins

As explained in Section 7.2, PQUIC peers can request proofs of the validity of protocol plugins when receiving them over a QUIC connection. The validation is carried out by the Plugin Validators. These validators can apply a range of techniques, from manual inspection, privacy checks [Ege+11; Li+15], to fuzzing [PAJ18] or using formal methods [MZ19] to validate the plugins submitted by developers. Formal methods are attractive because they enable validators to provide strong proofs for network protocols [Bis+05; BBK17; Chu+18].

A very important property for any code is its (correct) termination. If a protocol plugin would be stuck in an infinite loop with some specific input, then it would obviously be unsafe to use it in a PQUIC implementation. To demonstrate the possibility of using formal techniques to validate protocol plugins, we used the state-of-the-art T2 [CPR06; BK19] automated termination checker. T2 has been extended to handle a large fragment of Computational Tree Logic (CTL) [Coo+07; CKV11]. Verifying CTL reduces to an extended termination proof of the program combined with CTL information in states. Therefore, we focus here on the termination property.

Using the appropriate tools [Khl+15; KG+19], we checked the termination of our Multipath pluglets by compiling their C source code to T2 programs. The T2 prover assumes the termination of external functions, i.e., functions of the PQUIC implementation available through the POE. On the 26 pluglets composing the Multipath plugin, we managed to prove the termination of 23 of them. To obtain those proofs, we had to slightly modify the source code of some pluglets to ease the proof process. For example, we added an explicit size to NULL-terminated linked lists and used it to bound the loops iterating over them. The three remaining multipath pluglets could not be proven due to their complexity. It seems that T2 badly handles codes containing multiple nested (bounded) loops. Since T2 can export its termination proofs in files, these could be attached to the plugins to be the proof-carrying code proposed by Necula [Nec02]. However, given the size and complexity of these proofs, it is unreasonable to expect a PQUIC implementation to download and process them when loading plugins.

7.6 Related Works

Improving the flexibility of networks is a topic widely studied in the literature. In the late nineties, active networks [TW96; Ten+97] were proposed as a solution to bring innovation inside the network. Various techniques

were proposed to place bytecode inside packets so that routers could execute it while forwarding them. PLAN [Hic+98], ANTS [WGT98] and router plugins [Dec+98] are examples of such active techniques. Interest in active networks slowly decreased in the early 2000s [Cal06], but Software Defined Networks [McK+08] and P4 [Bos+14] can be considered as some of their successors.

Most of the work on active networks focused on the network layer and only a few researchers addressed the extensibility of the transport protocols. CTP [WHS01] is a transport protocol that is composed of various microprotocols that can be dynamically combined at run-time through the Cactus [Hil+99] system. STP [Pat+03] enables the utilization of code written in Cyclone [Jim+02] to extend a TCP implementation. icTCP [GAA04] exposes TCP state information and control to user-space applications to enable them to implement various extensions. To our knowledge, these techniques have not been deployed. In comparison, we believe our approach can be deployed at large scale as it relies on QUIC which prevents middlebox interference and enables a safe exchange of protocol plugins. Our Multipath plugin goes beyond the extensions proposed for STP and icTCP. CCP [Nar+18] provides a framework to write congestion control schemes in transport protocols in a generic way. Although we did not explore it in this Chapter, a new congestion controller could easily be implemented as a protocol plugin.

To deploy protocol plugins, we design a secure plugin management system that bares similarities to Certificate Transparency [RFC6962]. Our Merkle Prefix Tree construction has the major difference that plugin developers do not have to scan the entire tree in order to detect spurious plugins linked to their owned name, but only the branches in which their plugins lie.

Our Plugin Operating Environment is based on a simple user-space implementation of the eBPF VM [Fle17; IO 18]. Other execution environments that provide built-in memory checks such as CHERI-MIPS [Woo+14] or WebAssembly [Haa+17] could be valid alternatives to our POE. Evaluating their relevance in the protocol plugin context would be an interesting future work.

7.7 Discussion

Extending the behavior of client-side protocol implementations is difficult. First, deploying client updates can take several months or even years. Second, it remains unpractical to tune a protocol implementation when connections require very different services. Currently, experimental QUIC extensions such as `mp-quic` [DB17a] and QUIC-FEC [MDB19] are implemented as source code forks. Updating the base QUIC protocol and combining these extensions impose a significant engineering and maintenance burden, which is currently only affordable by large Internet companies. We envision PQUIC

implementations to be both simple and stable, providing connection-specific extensions through plugins. PQUIC could enable developers to focus on one implementation interface while still supporting very different implementation's internal architectures, such as zero-copy and partial hardware offload.

Through its plugins, PQUIC provides quick deployments of extension prototypes. Still, the benefits of plugins for complex standardized extensions do no fade. For instance, while the Multipath specification describes its wire format, both the packet scheduling and path management algorithms depend on the application. Whereas server implementations would likely provide built-in support of the Multipath extension for better performance, PQUIC provides more flexible tuning at client side than a simple on/off extension switch. These algorithms could be pushed by the server itself.

7.8 Conclusion and Future Work

Extensibility is a key requirement for many protocol designs. We leverage the unique features of QUIC to propose a new extensibility model that we call Pluginized QUIC (PQUIC). A PQUIC implementation is composed of a set of *protocol operations* which can be enriched or replaced by *protocol plugins*. These plugins are bytecodes executed by a Protocol Operating Environment (POE) that ensures their safety and portability. The plugins can be dynamically loaded by an application that uses PQUIC or received from the remote host thanks to our secure plugin management system. We demonstrate the benefits of this approach by implementing the Multipath extensions to QUIC exclusively with plugins.

This new protocol extensibility model opens several directions for future work. First, a similar approach could be used for other networking protocols in both the data place and the control plane [Wir+19]. Second, new techniques ensuring the implementation conformance to protocol specifications could be explored. These could leverage the PQUIC interface to assess that a plugin composition is correct. A third direction would be to revisit how we design protocol robustness [RBP19].

Another direction would be to develop new verification techniques adapted to pluginized protocols. While our POE detects and prevents the execution of a range of incorrect and malicious programs, there remain areas of improvements in the domain of static eBPF verification techniques [Ger+19].

Finally, PQUIC could be the starting point for the next version of QUIC. This would require the IETF to also specify protocol operations to ensure the inter-operability of plugins among different implementations. To achieve this, one must identify the minimal core protocol operations required. This set should be simple enough to allow very different implementations, having possibly very specific internal architectures such as zero-copy support, to

interoperate. Instead of adding more and more features to a monolithic implementation, developers could leverage the inherent extensibility of a pluginized protocol to develop over a simple set of kernel features that are easy to extend.

Conclusion

8

Over time, the application requirements have evolved. While the Internet was initially designed to remotely access terminals and exchange bulk files between computers, it now serves very diverse functions such as voice-activated applications and video streaming. In addition, the rise of mobile devices motivated the support of multiple paths in transport protocols. Yet, protocol implementations are often optimized for specific needs and they typically lack of flexibility to new use cases.

This thesis tackled these issues by exploring how we can make multipath transport protocols flexible to different application requirements. First, we performed in Chapter 2 two different measurement campaigns evaluating the performance of Multipath TCP on smartphones. The first one focused on the implementation in the Linux kernel and observed the traffic generated by real users using unmodified Android applications through a Multipath TCP-enabled proxy. While this setup enables smartphones to simultaneously use both Wi-Fi and cellular networks, only a few connections actually take advantage of these multiple paths, the other ones generating network overhead without sending useful data on additional paths. Nonetheless, we showed that connections that survived from the loss of their initial path represent a consequent fraction of connections using multiple paths. To further investigate user mobility, we performed as second measurement campaign based on the iOS implementation. We developed a specific application generating active measurements in both stable and mobile conditions. In particular, we revealed that the handover process from the Wi-Fi network to the cellular one is not abrupt, i.e., Multipath TCP simultaneously uses both wireless networks to handle such mobile situations. Still, with user motion there remains room for improvement in terms of application-perceived latency.

Then, based on our previous findings, we presented in Chapter 3 how we can tune Multipath TCP to better fit latency-sensitive applications under device mobility while meeting user's expectations in terms of cellular network usage. Our solution consists in three parts: (*i*) a server-side packet scheduler following the client's last decisions; (*ii*) a path-manager creating cellular subflows when noticing a bad primary network; and (*iii*) protocol additions to detect idle paths and to quickly create additional subflows. Our evaluation assessed that our tuned Multipath TCP achieves reasonable application-

perceived latency while limiting the cellular network overhead. However, due to deployment and technical constraints, our measurements with real users were limited to a few devices. Also, the overall implementation effort was quite large compared to their actual benefits.

To get rid of both kernel-space and TCP constraints, we considered the QUIC protocol and proposed an initial multipath design in Chapter 5. Compared to Multipath TCP, our proposal is cleaner and simpler to deploy at large-scale. By following an experimental design approach, we evaluated both Multipath TCP and Multipath gQUIC in a broad range of emulated network scenarios. In lossless networks, we obtained similar results for both Multipath TCP and Multipath gQUIC. Thanks to its better loss signaling, QUIC still benefits from the usage of multiple paths exhibiting losses, unlike TCP. We then extended our evaluation to real wireless networks and confirmed that the performance of Multipath TCP and Multipath gQUIC are similar. Nonetheless, our experience with real networks pointed out some limitations in that initial design. In Chapter 6, we reconsidered the Multipath extensions for the QUIC protocol to address all the previously raised concerns. While being more deployable that the initial design, our new one also takes into account network asymmetries. Our in-lab experiments confirm that in such networks, our improved design provides higher benefits than the initial one.

Finally, we proposed in Chapter 7 to completely revisit how transport protocols are specified and implemented. To make them fully extensible, we proposed that pluginizable protocol implementations provides a flexible API to their serving applications. Hosts can extend their protocol behaviors by injecting protocol plugins on a per-connection basis. These can be exchanged to the peer over the QUIC connection. We demonstrated that our Multipath extensions can be provided only with plugins and showed that we can achieve similar performance than a native Multipath QUIC implementation. We envision the case where pluginizable client implementations can be finely tuned by protocol plugins served by native server implementations. Such an approach would make transport protocols fully flexible to the application requirements.

Open Problems

This thesis opened several directions for future research. First, our MultipathTester application provides an opportunity to collect multipath protocols measurements at a large scale thanks to the iOS support of Multipath TCP and the Multipath QUIC implementation bundled by the application. For the sake of fairness, we shipped a Multipath QUIC packet scheduler similar to the one of iOS Multipath TCP in interactive mode. Thanks to the flexibility of Multipath QUIC, it would be possible to explore other multipath-specific

algorithms or even explore other QUIC extensions, such as Forward Erasure Connection.

Second, while MultiMob is an adaptation of Multipath TCP, some of its ideas can be applied to Multipath QUIC as well. In particular, we could consider the break-before-make path management approach to Multipath QUIC and how the oracle could be adapted to monitor QUIC flows instead of TCP ones.

Third, the new design of Multipath QUIC explicitly takes into account the network asymmetry to use unidirectional paths. This motivates research works about one-way path characteristic estimations — such as one-way delay — to perform cleverer packet scheduling.

Last but not least, we believe that our Pluginized QUIC work opens several followup works on its own. A first direction would apply the proposed extensibility model to other protocols, possibly at other layers than the transport one. A second direction would be to explore new verification techniques to assess that plugins fulfill specific safety properties. A third direction would be to identify and specify the core operations of PQUIC— and possibly including an advanced permission model — such that any protocol plugin can seamlessly work regardless of the PQUIC implementation.

Bibliography

[gQUIC] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk. *QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2*. Internet-Draft draft-hamilton-early-deployment-quic-00. July 2016.

[MPTCPlk] C. Paasch, S. Barre, et al. "Multipath TCP in the Linux Kernel". http://www.multipath-tcp.org. 2019.

[QUIC-14] J. Iyengar and M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Internet-Draft draft-ietf-quic-transport-14. Work in Progress. 2018. 129 pp.

[QUIC-c] A. Langley and W.-T. Chang. "QUIC Crypto". Dec. 2016.

[QUIC-inv] M. Thomson. *Version-Independent Properties of QUIC*. Internet-Draft draft-ietf-quic-invariants-07. IETF Secretariat, Spetember 2019.

[QUIC-t] J. Iyengar and M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Internet-Draft draft-ietf-quic-transport-27. IETF Secretariat, Feb. 2020.

[rfc1195] R. Callon. *Use of OSI IS-IS for routing in TCP/IP and dual environments*. RFC 1195. Dec. 1990. DOI: 10.17487/RFC1195.

[rfc1323] V. Jacobson, R. Braden, and D. Borman. *TCP Extensions for High Performance*. RFC1323. Fremont, CA, USA: RFC Editor, May 1992.

[rfc1889] R. Frederick, S. L. Casner, V. Jacobson, and H. Schulzrinne. *RTP: A Transport Protocol for Real-Time Applications*. RFC 1889. Jan. 1996. DOI: 10.17487/RFC1889.

[rfc1958] B. Carpenter (Ed.) *Architectural Principles of the Internet*. RFC 1958 (Informational). RFC. Updated by RFC 3439. Fremont, CA, USA: RFC Editor, June 1996. DOI: 10.17487/RFC1958.

[rfc2018] S. Floyd, J. Mahdavi, M. Mathis, and D. A. Romanow. *TCP Selective Acknowledgment Options*. RFC 2018. Oct. 1996. DOI: 10.17487/RFC2018.

[rfc2328] J. Moy. *OSPF Version 2*. RFC 2328. Apr. 1998. DOI: 10.17487/RFC2328.

[RFC2460] B. Hinden and D. S. E. Deering. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 2460. Dec. 1998. DOI: 10.17487/RFC2460.

[RFC2582] T. Henderson and S. Floyd. *The NewReno Modification to TCP's Fast Recovery Algorithm*. RFC 2582. Apr. 1999. DOI: 10.17487/RFC2582.

[RFC2960] R. R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. J. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. *Stream Control Transmission Protocol*. RFC2960. RFC. Fremont, CA, USA: RFC Editor, Oct. 2000.

[RFC3022] K. B. Egevang and P. Srisuresh. *Traditional IP Network Address Translator (Traditional NAT)*. RFC 3022. Jan. 2001. DOI: 10.17487/RFC3022.

[RFC3168] K. K. Ramakrishnan, S. Floyd, and D. L. Black. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168. RFC. Updated by RFCs 4301, 6040, 8311. Fremont, CA, USA: RFC Editor, Sept. 2001.

[RFC4271] Y. Rekhter, S. Hares, and T. Li. *A Border Gateway Protocol 4 (BGP-4)*. RFC 4271. Jan. 2006. DOI: 10.17487/RFC4271.

[RFC4960] R. Stewart (Ed.) *Stream Control Transmission Protocol*. RFC 4960 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Sept. 2007. DOI: 10.17487/RFC4960.

[RFC5246] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246 (Proposed Standard). RFC. Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685, 7905, 7919. Fremont, CA, USA: RFC Editor, Aug. 2008. DOI: 10.17487/RFC5246.

[RFC6146] P. Matthews, I. van Beijnum, and M. Bagnulo. *Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers*. RFC 6146. Apr. 2011. DOI: 10.17487/RFC6146.

[RFC6181] M. Bagnulo. *Threat Analysis for TCP Extensions for Multipath Operation with Multiple Addresses*. RFC 6181. Mar. 2011. DOI: 10.17487/RFC6181.

[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.

[RFC6824] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. *TCP Extensions for Multipath Operation with Multiple Addresses*. Request for Comments 6824. Published: RFC 6824. IETF, Jan. 2013.

[RFC6824B] A. Ford, C. Raiciu, M. J. Handley, O. Bonaventure, and C. Paasch. *TCP Extensions for Multipath Operation with Multiple Addresses*. Internet-Draft draft-ietf-mptcp-rfc6824bis-18. Work in Progress. Internet Engineering Task Force, June 2019. 82 pp.

[RFC6962] B. Laurie, A. Langley, and E. Kasper. *Certificate Transparency*. RFC 6962. RFC. Fremont, CA, USA: RFC Editor, June 2013.

[RFC7258] S. Farrell and H. Tschofenig. *Pervasive Monitoring Is an Attack*. RFC 7258 (Best Current Practice). RFC. Fremont, CA, USA: RFC Editor, May 2014. DOI: 10.17487/RFC7258.

[RFC7413] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain. *TCP Fast Open*. RFC 7413. Dec. 2014. DOI: 10.17487/RFC7413.

[RFC7414] M. Duke, R. Braden, W. Eddy, E. Blanton, and A. Zimmermann. *A Roadmap for Transmission Control Protocol (TCP) Specification Documents*. RFC 7414. RFC. Fremont, CA, USA: RFC Editor, Feb. 2015. DOI: 10.17487/RFC7414.

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

[RFC768] J. Postel. *User Datagram Protocol*. RFC 768 (Internet Standard). RFC. Fremont, CA, USA: RFC Editor, Aug. 1980. DOI: 10.17487/RFC0768.

[RFC791] J. Postel. *Internet Protocol*. RFC 791. Sept. 1981. DOI: 10.17487/RFC0791.

[RFC793] J. Postel. *Transmission Control Protocol*. RFC 793. Sept. 1981. DOI: 10.17487/RFC0793.

[RFC8446] E. Rescola. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446 (Internet Standard). RFC. Fremont, CA, USA: RFC Editor, Dec. 2018. DOI: 10.17487/RFC8446.

[AEK09] A. Asheralieva, T. J. Erke, and K. Kilkki. "Traffic Characterization and Service Performance in FON Network". In: *2009 First International Conference on Future Information Networks*. Oct. 2009, pp. 285–291. DOI: 10.1109/ICFIN.2009.5339600.

[AK19] M. Amend and J. Kang. *Multipath TCP Extension for Robust Session Establishment*. Internet-Draft draft-amend-mptcp-robe-01. Work in Progress. Internet Engineering Task Force, Nov. 2019. 27 pp.

[Alm+99] W. Almesberger et al. *Linux network traffic control—implementation overview*. 1999.

[And12] O. Andersson. *Experiment!: planning, implementing and interpreting*. John Wiley & Sons, 2012.

[AOSP] Google. *Android Open Source Project source code*. Branch android-6.0.1-r77. Dec. 2016. URL: https://android.googlesource.com/platform/frameworks/base.

[App18] Apple. *Improving Network Reliability Using Multipath TCP*. 2018. URL: https://developer.apple.com/documentation/foundation/urlsessionconfiguration/improving%5C_network%5C_reliability%5C_using%5C_multipath%5C_tcp.

[Arz+14] B. Arzani et al. "Deconstructing MPTCP Performance". In: *ICNP'2014*. 2014, pp. 269–274.

[Ass+16] M. Assefi, G. Liu, M. P. Wittit, and C. Izurieta. "Measuring the Impact of Network Performance on Cloud-Based Speech Recognition Applications". In: *International Journal of Computer Applications-IJCA* 23.1 (2016), pp. 19–28.

[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.

[Bal18] P. Balasubramanian. *usage for timestamp options in the wild*. Sept. 2018. URL: https://mailarchive.ietf.org/arch/legacy/msg/tcpm/11522.

[BBK17] K. Bhargavan, B. Blanchet, and N. Kobeissi. "Verified models and reference implementations for the TLS 1.3 standard candidate". In: *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 483–502.

[BBV09] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. "Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications". In: *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*. ACM Press, 2009, pp. 280–293.

[BD18] W. de Bruijn and E. Dumazet. "Optimizing UDP for content delivery: GSO, pacing and zerocopy". In: *Linux Plumbers Conference*. 2018.

[Bev+13] R. Beverly, W. Brinkmeyer, M. Luckie, and J. P. Rohrer. "IPv6 alias resolution via induced fragmentation". In: *PAM'13*. Springer, 2013, pp. 155–165.

[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. ACM, 2005, pp. 265–276.

[Bit+10] A. Bittau, M. Hamburg, M. Handley, D. Mazieres, and D. Boneh. "The Case for Ubiquitous Transport-Level Encryption." In: *USENIX Security Symposium*. 2010, pp. 403–418.

[BK19] M. Brockschmidt and H. Khlaaf. *T2 Temporal Prover*. Jan. 2019.

[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.

[Bon13] O. Bonaventure. *Apple seems to also believe in Multipath TCP*. 2013. URL: http://perso.uclouvain.be/olivier.bonaventure/blog/html/2013/09/18/mptcp.html.

[Bon18] O. Bonaventure. *The first Multipath TCP enabled smartphones*. 2018. URL: http://blog.multipath-tcp.org/blog/html/2018/12/10/the_first_multipath_tcp_enabled_smartphones.html.

[Bos+14] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. "P4: Programming protocol-independent packet processors". In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95.

[BP95] L. Brakmo and L. Peterson. "TCP Vegas: end to end congestion avoidance on a global Internet". In: *IEEE Journal on Selected Areas in Communications* 13.8 (Oct. 1995), pp. 1465–1480. ISSN: 0733-8716, 1558-0008. DOI: 10.1109/49.464716.

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

[BS16] O. Bonaventure and S. Seo. "Multipath TCP Deployments". In: *IETF Journal*. Vol. 12. 2. Nov. 2016, pp. 24–27.

[Bud+12] h. Budzisz, J. Garcia, A. Brunstrom, and R. Ferrús. "A taxonomy and survey of SCTP research". In: *ACM Computing Surveys (CSUR)* 44.4 (2012), p. 18.

[Cal06] K. Calvert. "Reflections on network architecture: an active networking perspective". In: *ACM SIGCOMM Computer Communication Review* 36.2 (2006), pp. 27–30.

[CDM15] G. Carlucci, L. De Cicco, and S. Mascolo. "HTTP over UDP: an Experimental Investigation of QUIC". In: *SAC'15*. ACM, 2015, pp. 609–614.

[CH10] A. Carroll and G. Heiser. "An Analysis of Power Consumption in a Smartphone". In: *USENIX'10*. Vol. 14. Boston, MA, 2010.

[Che+13] Y.-C. Chen, Y.-s. Lim, R. J. Gibbens, E. M. Nahum, R. Khalili, and D. Towsley. "A measurement-based study of MultiPath TCP performance over wireless networks". In: *Proceedings of the 2013 conference on Internet measurement conference - IMC '13*. Barcelona, Spain: ACM Press, 2013, pp. 455–468. ISBN: 978-1-4503-1953-9. DOI: 10.1145/2504730.2504751.

[Che+15] X. Chen, A. Jindal, N. Ding, Y. C. Hu, M. Gupta, and R. Vannithamby. "Smartphone Background Activities in the Wild: Origin, Energy Drain, and Optimization". In: *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*. Paris, France: ACM Press, 2015, pp. 40–52.

[Chu+18] A. Chudnov, N. Collins, B. Cook, J. Dodds, B. Huffman, C. MacCárthaigh, S. Magill, E. Mertens, E. Mullen, S. Tasiran, et al. "Continuous formal verification of amazon s2n". In: *International Conference on Computer Aided Verification*. Springer, 2018, pp. 430–446.

[CKV11] B. Cook, E. Koskinen, and M. Vardi. "Temporal property verification as a program analysis task". In: *International Conference on Computer Aided Verification*. Springer, 2011, pp. 333–348.

[Cle+17] L. Clemente et al. "A QUIC implementation in pure go". 2017.

[Coo+07] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Y. Vardi. "Proving that programs eventually do something good". In: *ACM SIGPLAN Notices* 42.1 (2007), pp. 265–276.

[CoreL] Apple. *Core Location*. 2019. URL: https://developer.apple.com/documentation/corelocation.

[Cos+18] S. Costea, M. O. Choudary, D. Gucea, B. Tackmann, and C. Raiciu. "Secure Opportunistic Multipath Key Exchange". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security - CCS '18*. Toronto, Canada: ACM Press, 2018, pp. 2077–2094. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243791. (Visited on 12/16/2019).

[CPR06] B. Cook, A. Podelski, and A. Rybalchenko. "TERMINATOR: beyond safety". In: *International Conference on Computer Aided Verification*. Springer, 2006, pp. 415–418.

[CS18] L. Clemente and M. Seemann. "quic-go". 2018.

[CSP15] M. Coudron, S. Secci, and G. Pujolle. "Differentiated pacing on multiple paths to improve one-way delay estimations". In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. ISSN: 1573-0077. May 2015, pp. 672–678. DOI: 10.1109/INM.2015.7140354.

[CSP17] S. Cheshire, D. Schinazi, and C. Paasch. "Advances in Networking, Part 1". June 2017.

[CT14] Y. Chen and D. Towsley. "On bufferbloat and delay analysis of multipath TCP in wireless networks". In: *2014 IFIP Networking Conference*. June 2014, pp. 1–9. DOI: 10.1109/IFIPNetworking.2014.6857081.

[Darwin] Apple. *XNU Kernel Source Code*. 2018. URL: https://opensource.apple.com/source/xnu/.

[DB] G. Detal and S. Barré. *Multipath TCP v0.89.5 kernel for Google Nexus 5*. URL: http://dl.acm.org/citation.cfm?doid=3132062.3132066.

[DB17a] Q. De Coninck and O. Bonaventure. "Multipath QUIC: Design and Evaluation". In: *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies - CoNEXT '17*. Incheon, Republic of Korea: ACM Press, December 2017, pp. 160–166. ISBN: 978-1-4503-5422-6. DOI: 10.1145/3143361.3143370.

[DB19b] Q. De Coninck and O. Bonaventure. *The Case for Protocol Plugins*. Tech. rep. 2019.

[De +19] 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 ACM Special Interest Group on Data Communication - SIGCOMM '19*. Beijing, China: ACM

Press, 2019, pp. 59–74. ISBN: 978-1-4503-5956-6. DOI: 10.1145/3341302.3342078.

[Dec+98] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. "Router plugins: A software architecture for next generation routers". In: *ACM SIGCOMM Computer Communication Review* 28.4 (1998), pp. 229–240.

[Den+14] S. Deng, R. Netravali, A. Sivaraman, and H. Balakrishnan. "WiFi, LTE, or Both?: Measuring Multi-Homed Wireless Internet Performance". In: *Proceedings of the 2014 Conference on Internet Measurement Conference - IMC '14*. Vancouver, BC, Canada: ACM Press, 2014, pp. 181–194. ISBN: 978-1-4503-3213-2. DOI: 10.1145/2663716.2663727.

[Det+13] G. Detal, B. Hesmans, O. Bonaventure, Y. Vanaubel, and B. Donnet. "Revealing middlebox interference with tracebox". In: *Proceedings of the 2013 conference on Internet measurement conference - IMC '13*. Barcelona, Spain: ACM Press, 2013, pp. 1–8. ISBN: 978-1-4503-1953-9. DOI: 10.1145/2504730.2504757.

[Dha+12] A. Dhamdhere, M. Luckie, B. Huffaker, A. Elmokashfi, E. Aben, et al. "Measuring the deployment of IPv6: topology, routing and performance". In: *IMC'12*. ACM, 2012, pp. 537–550.

[DMT02] T. Dunigan, M. Mathis, and B. Tierney. "A TCP tuning daemon". In: *Supercomputing, ACM/IEEE 2002 Conference*. IEEE, 2002, pp. 9–9.

[DRT08] L. De Vito, S. Rapuano, and L. Tomaciello. "One-Way Delay Measurement: State of the Art". In: *IEEE Transactions on Instrumentation and Measurement* 57.12 (Dec. 2008), pp. 2742–2750. ISSN: 1557-9662. DOI: 10.1109/TIM.2008.926052.

[Duk+13] N. Dukkipati, N. Cardwell, Y. Cheng, and M. Mathis. "Tail loss probe (TLP): An algorithm for fast recovery of tail losses". In: *draft-dukkipati-tcpm-tcploss-probe-01. txt* (2013).

[Ear13] P. Eardley. *Survey of MPTCP Implementations*. Internet-Draft draft-eardley-mptcp-implementations-survey-02. Work in Progress. Internet Engineering Task Force, July 2013. 38 pp.

[Edg15] J. Edge. "A seccomp overview". In: *Linux Weekly News* (Sept. 2015).

[Ege+11] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. "PiOS: Detecting Privacy Leaks in iOS Applications." In: *NDSS*. 2011, pp. 177–183.

[Fal+10] H. Falaki, D. Lymberopoulos, R. Mahajan, S. Kandula, and D. Estrin. "A first look at traffic on smartphones". In: *Proceedings of the 10th annual conference on Internet measurement - IMC '10*. Melbourne, Australia: ACM Press, 2010, p. 281. ISBN: 978-1-4503-0483-2. DOI: 10.1145/1879141.1879176.

[FDA14a] S. Ferlin-Oliveira, T. Dreibholz, and Ö. Alay. "Tackling the challenge of bufferbloat in Multi-Path Transport over heterogeneous wireless networks". In: *2014 IEEE 22nd International Symposium of Quality of Service (IWQoS)*. May 2014, pp. 123–128. DOI: 10.1109/IWQoS.2014.6914310.

[FDA14b] S. Ferlin, T. Dreibholz, and Ö. Alay. "Multi-path transport over heterogeneous wireless networks: Does it really pay off?" In: *2014 IEEE Global Communications Conference*. Dec. 2014, pp. 4807–4813. DOI: 10.1109/GLOCOM.2014.7037567.

[Fer+16] S. Ferlin, O. Alay, O. Mehani, and R. Boreli. "BLEST: Blocking Estimation-based MPTCP Scheduler for Heterogeneous Networks". In: *2016 IFIP Networking Conference (IFIP Networking) and Workshops*. Vienna, Austria: IEEE, May 2016, pp. 431–439. ISBN: 978-3-901882-83-8. DOI: 10.1109/IFIPNetworking.2016.7497206.

[FF96] K. Fall and S. Floyd. "Simulation-based comparisons of Tahoe, Reno and SACK TCP". In: *ACM SIGCOMM Computer Communication Review* 26.3 (1996), pp. 5–21.

[Fis35] R. A. Fisher. "The design of experiments." In: (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.

[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. ACM, 2013, pp. 159–170.

[Fle17] M. Fleming. "A thorough introduction to eBPF". In: *Linux Weekly News* (Dec. 2017).

[Fro+16] A. Frommgen, T. Erbshäußer, A. Buchmann, T. Zimmermann, and K. Wehrle. "ReMP TCP: Low latency multipath TCP". In: *2016 IEEE International Conference on Communications (ICC)*. ISSN: 1938-1883. May 2016, pp. 1–7. DOI: 10.1109/ICC.2016.7510787.

[Frö+17] A. Frömmgen, A. Rizk, T. Erbshäußer, M. Weller, B. Koldehofe, A. Buchmann, and R. Steinmetz. "A programming model for application-defined multipath TCP scheduling". In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference on - Middleware '17*. Las Vegas, Nevada: ACM Press, 2017, pp. 134–146. ISBN: 978-1-4503-4720-4. DOI: 10.1145/3135974.3135979. (Visited on 12/09/2019).

[Frö17] A. Frömmgen. "Mininet /Netem Emulation Pitfalls: A Multipath TCP Scheduling Experience". In: (2017).

[Fuk11] K. Fukuda. "An analysis of longitudinal tcp passive measurements (short paper)". In: *International Workshop on Traffic Monitoring and Analysis*. Springer, 2011, pp. 29–36.

[GAA04] H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. "Deploying Safe User-Level Network Services with icTCP." In: *OSDI*. 2004, pp. 317–332.

[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.

[Ger+19] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv. "Simple and precise static analysis of untrusted Linux kernel extensions". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2019*. Phoenix, AZ, USA: ACM Press, 2019, pp. 1069–1084. ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314590. (Visited on 12/11/2019).

[Gre15] B. Gregg. "eBPF: One Small Step". May 2015.

[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.

[Han+12] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. "Reproducible Network Experiments Using Container-Based Emulation". In: *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM, 2012, pp. 253–264.

[Han+15] B. Han, F. Qian, S. Hao, and L. Ji. "An Anatomy of Mobile Web Performance over Multipath TCP". In: *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies - CoNEXT '15*. Heidelberg, Germany: ACM

Press, 2015, pp. 1–7. ISBN: 978-1-4503-3412-9. DOI: 10.1145/2716281.2836090.

[Han+16] B. Han, F. Qian, L. Ji, and V. Gopalakrishnan. "MP-DASH: Adaptive Video Streaming Over Preference-Aware Multipath". In: *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies - CoNEXT '16*. Irvine, California, USA: ACM Press, 2016, pp. 129–143. ISBN: 978-1-4503-4292-6. DOI: 10.1145/2999572.2999606.

[HB14] B. Hesmans and O. Bonaventure. "Tracing multipath TCP connections". In: *Proceedings of the 2014 ACM conference on SIGCOMM - SIGCOMM '14*. Chicago, Illinois, USA: ACM Press, 2014, pp. 361–362. ISBN: 978-1-4503-2836-4. DOI: 10.1145/2619239.2631453.

[HB16] B. Hesmans and O. Bonaventure. "An Enhanced Socket API for Multipath TCP". In: *Proceedings of the 2016 Applied Networking Research Workshop*. ACM Press, 2016, pp. 1–6.

[Hem+05] S. Hemminger et al. "Network emulation with NetEm". In: *Linux conf au*. 2005, pp. 18–23.

[Hem16] J. Hempel. *Voice Is the Next Big Platform, and Alexa Will Own It*. Dec. 2016. URL: https://backchannel.com/voice-is-the-next-big-platform-and-alexa-will-own-it-c2cf13fab911%5C#.3eewnvjbp.

[Hes+13] B. Hesmans, F. Duchene, C. Paasch, G. Detal, and O. Bonaventure. "Are TCP Extensions Middlebox-proof?" In: *CoNEXT Workshop HotMiddlebox*. 2013. DOI: 10.1145/2535828.2535830.

[Hes+15] B. Hesmans, H. T. Viet, R. Sadre, and O. Bonaventure. "A first look at real Multipath TCP traffic". In: *TMA'15*. Apr. 2015.

[Hic+98] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. "PLAN: A packet language for active networks". In: *ACM SIGPLAN Notices* 34.1 (1998), pp. 86–93.

[Hil+99] M. A. Hiltunen, R. D. Schlichting, X. Han, M. M. Cardozo, and R. Das. "Real-time dependable channels: Customizing QoS attributes for distributed systems". In: *IEEE Transactions on Parallel and Distributed Systems* 10.6 (1999), pp. 600–612.

[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 - IMC '11*. Berlin, Germany: ACM Press, 2011, pp. 181–194. ISBN: 978-1-4503-1013-0. DOI: 10.1145/2068816.2068834.

[HRX08] S. Ha, I. Rhee, and L. Xu. "CUBIC: a new TCP-friendly high-speed TCP variant". In: *ACM SIGOPS Operating Systems Review* 42.5 (July 2008), pp. 64–74. ISSN: 01635980. DOI: 10.1145/1400097.1400105.

[Hu+15] X. Hu, L. Song, D. Van Bruggen, and A. Striegel. "Is There WiFi Yet?: How Aggressive Probe Requests Deteriorate Energy and Throughput". In: *Proceedings of the 2015 ACM Conference on Internet Measurement Conference - IMC '15*. Tokyo, Japan: ACM Press, 2015, pp. 317–323. ISBN: 978-1-4503-3848-6. DOI: 10.1145/2815675.2815709.

[Hua+10] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. "Anatomizing application performance differences on smartphones". In: *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 2010, pp. 165–178. DOI: 10.1145/1814433.1814452.

[Hua+12] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. "A Close Examination of Performance and Power Characteristics of 4G LTE Networks". In: *Proceedings of the 10th international conference on Mobile systems, applications, and services - MobiSys '12*. Low Wood Bay, Lake District, UK: ACM Press, 2012, p. 225. ISBN: 978-1-4503-1301-8. DOI: 10.1145/2307636.2307658.

[Hui18] C. Huitema. "picoquic". 2018.

[ICA11] ICANN. *Available Pool of Unallocated IPv4 Internet Addresses Now Completely Emptied*. Feb. 2011. URL: https://www.icann.org/en/system/files/press-materials/release-03feb11-en.pdf.

[IO 18] IO Visor Project. "Userspace eBPF VM". 2018.

[iPerf] J. Dugan et al. *iPerf*. 2019. URL: https://iperf.fr/.

[IS18] J. Iyengar and I. Swett. "QUIC: Developing and Deploying a TCP Replacement for the Web". In: *Netdev 0x12*. 2018.

[Iye18] S. Iyengar. "Moving fast at scale: Experience deploying IETF QUIC at Facebook". Keynote presented at the Workshop on the Evolution, Performance, and Interoperability of QUIC – EPIQ'18. Dec. 2018.

[Jac95] V. Jacobson. "Congestion avoidance and control". In: *ACM SIGCOMM Computer Communication Review* 25.1 (1995), pp. 157–187.

[Jim+02] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. "Cyclone: A Safe Dialect of C." In: *USENIX Annual Technical Conference, General Track*. 2002, pp. 275–288.

[Kak+17] A. M. Kakhki, S. Jero, D. Choffnes, A. Mislove, and C. Nita-Rotaru. "Taking a Long Look at QUIC: An Approach for Rigorous Evaluation of Rapidly Evolving Transport Protocols". In: *IMC'17*. ACM, 2017.

[Kas12] D. Kaspar. "Multipath aggregation of heterogeneous access networks". In: *ACM SIGMultimedia Records* 4.1 (2012), pp. 27–28.

[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.

[KG+19] T. Kremenek, D. Gregor, et al. "Clang: a C language family frontend for LLVM". 2019.

[Kha+12] R. Khalili, N. Gast, M. Popovic, U. Upadhyay, and J.-Y. Le Boudec. "MPTCP is not Pareto-Optimal: Performance Issues and a Possible Solution". In: *ACM CoNEXT*. 2012. DOI: 10.1145/2413176.2413178.

[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. DOI: 10.1109/TNET.2013.2274462.

[Khl+15] H. Khlaaf, M. Brockschmidt, S. Falke, D. Kapur, and C. Sinz. "llvm2KITTeL tailored for T2". 2015.

[Kni16] W. Knight. "10 Breakthrough Technologies 2016: Conversational Interfaces". In: *MIT Technology Review* (Feb. 2016).

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

[KT] S. Seo. "KT's GiGA LTE". In: Presented as the 93th IETF in Prague, Czech Republic, 2015.

[Lan+17] A. Langley, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, A. Riddoch, W.-T. Chang, Z. Shi, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, and I. Swett. "The QUIC Transport Protocol: Design and Internet-Scale Deployment". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication - SIGCOMM '17*. Los Angeles, CA, USA: ACM Press, 2017, pp. 183–196. ISBN: 978-1-4503-4653-5. DOI: 10.1145/3098822.3098842. (Visited on 11/08/2019).

[Leb17] D. Lebrun. "Reaping the Benefits of IPv6 Segment Routing". PhD thesis. UCLouvain / ICTEAM / EPL, Oct. 2017.

[LED16] I. Livadariu, A. Elmokashfi, and A. Dhamdhere. "Characterizing IPv6 control and data plane stability". In: *IEEE INFOCOM 2016*. IEEE, 2016, pp. 1–9.

[Li+15] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. "Iccta: Detecting inter-component privacy leaks in android apps". In: *Proceedings of the 37th International Conference on Software Engineering*. IEEE Press, 2015, pp. 280–291.

[Li+18] L. Li, K. Xu, T. Li, K. Zheng, C. Peng, D. Wang, X. Wang, M. Shen, and R. Mijumbi. "A measurement study on multi-path TCP with multiple cellular carriers on high speed rails". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication - SIGCOMM '18*. Budapest, Hungary: ACM Press, 2018, pp. 161–175. ISBN: 978-1-4503-5567-4. DOI: 10.1145/3230543.3230556.

[Lim+14a] Y.-s. Lim, Y.-C. Chen, E. M. Nahum, D. Towsley, and R. J. Gibbens. "How Green is Multipath TCP for Mobile Devices?" In: *Proceedings of the 4th workshop on All things cellular: operations, applications, & challenges*. ACM, 2014, pp. 3–8.

[Lim+14b] Y.-s. Lim, Y.-C. Chen, E. M. Nahum, D. Towsley, and K.-W. Lee. "Cross-layer path management in multi-path transport protocol for mobile devices". In: *INFOCOM, 2014 Proceedings IEEE*. IEEE, 2014, pp. 1815–1823.

[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 - CoNEXT '17*. Incheon, Republic of Korea: ACM Press, 2017, pp. 147–159. ISBN: 978-1-4503-5422-6. DOI: 10.1145/3143361.3143376.

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

[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: *Computer Communications Workshops (INFOCOM WKSHPS), 2015 IEEE Conference on*. IEEE, 2015, pp. 312–317.

[Lyc+15] R. Lychev, S. Jero, A. Boldyreva, and C. Nita-Rotaru. "How secure and quick is QUIC? Provable security and performance analyses". In: *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 214–231.

[MCC03] M. Mellia, A. Carpani, and R. L. Cigno. "TStat: TCP STatistic and analysis tool". In: *Quality of Service in Multiservice IP Networks*. Springer, 2003, pp. 145–157.

[McK+08] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. "OpenFlow: enabling innovation in campus networks". In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74.

[MDB19] F. Michel, Q. De Coninck, and O. Bonaventure. "QUIC-FEC: Bringing the benefits of Forward Erasure Correction to QUIC". In: *Proceedings of the 18th IFIP Networking Conference (IFIP Networking) and Workshops – IFIP Networking'19*. IEEE, May 2019. DOI: 10.23919/IFIPNetworking.2019.8816838.

[Mel+15] M. Melara, A. Blankstein, J. Bonneau, E. Felten, and M. Freedman. "CONIKS: Bringing Key Transparency to End Users". In: *USENIX Security Symposium*. Vol. 2015. 2015, pp. 383–398.

[Mer87] R. C. Merkle. "A digital signature based on a conventional encryption function". In: *Conference on the theory and application of cryptographic techniques*. Springer, 1987, pp. 369–378.

[MKM16] P. Megyesi, Z. Krämer, and S. Molnár. "How quick is QUIC?" In: *ICC'16*. IEEE, 2016, pp. 1–6.

[MPiOS] Apple. *Improving Network Reliability Using Multipath TCP*. 2019. URL: https://developer.apple.com/documentation/foundation/nsurlsessionconfiguration/improving_network_reliability_using_multipath_tcp.

[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 - SIGCOMM '19*. Beijing, China: ACM Press, 2019, pp. 227–240. ISBN: 978-1-4503-5956-6. DOI: 10.1145/3341302.3342087. (Visited on 12/11/2019).

[Nar+18] A. Narayan, F. Cangialosi, D. Raghavan, P. Goyal, S. Narayana, R. Mittal, M. Alizadeh, and H. Balakrishnan. "Restructuring endpoint congestion control". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 30–43.

[Nec02] G. C. Necula. "Proof-carrying code. Design and implementation". In: *Proof and system-reliability*. Springer, 2002, pp. 261–288.

[Nik+15] A. Nika, Y. Zhu, N. Ding, A. Jindal, Y. C. Hu, X. Zhou, B. Y. Zhao, and H. Zheng. "Energy and Performance of Smartphone Radio Bundling in Outdoor Environments". In: *Proceedings of the 24th International Conference on World Wide Web - WWW '15*. Florence, Italy: ACM Press, 2015, pp. 809–819. ISBN: 978-1-4503-3469-3. DOI: 10.1145/2736277.2741635.

[Nik+16] A. Nikravesh, Y. Guo, F. Qian, Z. M. Mao, and S. Sen. "An In-depth Understanding of Multipath TCP on Mobile Devices: Measurement and System Design". In: *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking - MobiCom '16*. New York City, New York: ACM Press, 2016, pp. 189–201. ISBN: 978-1-4503-4226-1. DOI: 10.1145/2973750.2973769.

[Not18] M. Nottingham. *Getting to consensus on packet number encryption*. Apr. 2018. URL: https://mailarchive.ietf.org/arch/msg/quic/mZDX1vcmHPOiTDazPrrB8IKPagQ.

[OL15] B.-H. Oh and J. Lee. "Constraint-based proactive scheduling for MPTCP in wireless networks". In: *Computer Networks* 91 (2015), pp. 548–563.

[Paa+12] C. Paasch, G. Detal, F. Duchene, C. Raiciu, and O. Bonaventure. "Exploring Mobile/WiFi Handover with Multipath TCP". In: *ACM SIGCOMM CellNet workshop*. event-place: Helsinki, Finland. 2012, pp. 31–36. ISBN: 978-1-4503-1475-6.

[Paa+14] C. Paasch, S. Ferlin, O. Alay, and O. Bonaventure. "Experimental evaluation of multipath TCP schedulers". In: *Proceedings of the 2014 ACM SIGCOMM workshop on Capacity sharing workshop*. ACM, 2014, pp. 27–32.

[Paa16] C. Paasch. “Network support for TCP Fast Open”. In: *Presentation at NANOG* 67 (2016).

[PAJ18] S. Pailoor, A. Aday, and S. Jana. “MoonShine: Optimizing ${$OS$}$ Fuzzer Seed Selection with Trace Distillation”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 729–743.

[Pat+03] P. Patel, A. Whitaker, D. Wetherall, J. Lepreau, and T. Stack. “Upgrading Transport Protocols using Untrusted Mobile Code”. In: *ACM SIGOPS Operating Systems Review* 37.5 (2003), pp. 1–14.

[Pat+08] A. Pathak, H. Pucha, Y. Zhang, Y. C. Hu, and Z. M. Mao. “A Measurement Study of Internet Delay Asymmetry”. In: *Passive and Active Network Measurement*. Ed. by M. Claypool and S. Uhlig. Vol. 4979. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 182–191. ISBN: 978-3-540-79231-4 978-3-540-79232-1. DOI: 10.1007/978-3-540-79232-1_19. (Visited on 12/16/2019).

[PEK11] C. Pluntke, L. Eggert, and N. Kiukkonen. “Saving Mobile Device Energy with Multipath TCP”. In: *ACM MobiArch 2011*. 2011, pp. 1–6.

[Pen+14] Q. Peng, M. Chen, A. Walid, and S. Low. “Energy Efficient Multipath TCP for Mobile Devices”. In: *Proceedings of the 15th ACM international symposium on Mobile ad hoc networking and computing - MobiHoc ’14*. Philadelphia, Pennsylvania, USA: ACM Press, 2014, pp. 257–266. ISBN: 978-1-4503-2620-9. DOI: 10.1145/2632951.2632971. (Visited on 09/25/2019).

[Pen+16] Q. Peng, A. Walid, J. Hwang, and S. H. Low. “Multipath TCP: Analysis, design, and implementation”. In: *IEEE/ACM Transactions on Networking* 24.1 (2016), pp. 596–609.

[Pin15] C. Pinedo. *Improve active/backup subflow selection*. Jan. 2015. URL: https://github.com/multipath-tcp/mptcp/pull/70.

[PKB13] C. Paasch, R. Khalili, and O. Bonaventure. “On the benefits of applying experimental design to improve multipath TCP”. In: *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. ACM, 2013, pp. 393–398.

[Ra+10] M.-R. Ra, J. Paek, A. B. Sharma, R. Govindan, M. H. Krieger, and M. J. Neely. "Energy-Delay Tradeoffs in Smartphone Applications". In: *Proceedings of the 8th international conference on Mobile systems, applications, and services - MobiSys '10*. San Francisco, California, USA: ACM Press, 2010, pp. 255–269. ISBN: 978-1-60558-985-5. DOI: 10.1145/1814433.1814459.

[Rad+11] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. "TCP Fast Open". In: *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*. Tokyo, Japan: ACM, 2011, 21:1–21:12. ISBN: 978-1-4503-1041-3. DOI: 10.1145/2079296.2079317.

[Rai+11a] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. "Improving Datacenter Performance and Robustness with Multipath TCP". In: *ACM SIGCOMM 2011*. eventplace: Toronto, Ontario, Canada. 2011. ISBN: 978-1-4503-0797-0. DOI: 10.1145/2018436.2018467.

[Rai+11b] C. Raiciu, D. Niculescu, M. Bagnulo, and M. Handley. "Opportunistic Mobility with Multipath TCP". In: *ACM MobiArch 2011*. 2011.

[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: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 29–42.

[RBP19] F. Rochet, O. Bonaventure, and O. Pereira. "Flexible Anonymous Network". In: *12th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2019)*. 2019.

[RD08] E. Rescorla and T. Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. Request for Comments 5246. Published: RFC 5246. RFC Editor, Aug. 2008. DOI: 10.17487/RFC5246.

[Reach] Apple. *Reachability*. 2016. URL: https://developer.apple.com/library/archive/samplecode/Reachability/%20Introduction/Intro.html.

[RHB18] A. Rabitsch, P. Hurtig, and A. Brunstrom. "A Stream-Aware Multipath QUIC Scheduler for Heterogeneous Paths: Paper # XXX, XXX pages". In: *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC - EPIQ'18*. Heraklion, Greece: ACM Press, 2018, pp. 29–35. ISBN: 978-1-4503-

6082-1. DOI: 10.1145/3284850.3284855. (Visited on 11/27/2019).

[Ros13] J. Roskind. "QUIC(Quick UDP Internet Connections): Multiplexed Stream Transport Over UDP". 2013.

[Rüt+18] J. Rüth, I. Poese, C. Dietzel, and O. Hohlfeld. "A First Look at QUIC in the Wild". In: *Passive and Active Measurement*. Ed. by R. Beverly, G. Smaragdakis, and A. Feldmann. Vol. 10771. Cham: Springer International Publishing, 2018, pp. 255–268. ISBN: 978-3-319-76480-1 978-3-319-76481-8. DOI: 10.1007/978-3-319-76481-8_19. (Visited on 11/14/2019).

[Sah+17] S. K. Saha, A. Kannan, G. Lee, N. Ravichandran, P. K. Medhe, N. Merchant, and D. Koutsonikolas. "Multipath TCP in Smartphones: Impact on Performance, Energy, and CPU Utilization". In: *Proceedings of the 15th ACM International Symposium on Mobility Management and Wireless Access - MobiWac '17*. Miami, Florida, USA: ACM Press, 2017, pp. 23–31. ISBN: 978-1-4503-5163-8. DOI: 10.1145/3132062.3132066.

[SB12] J. Sommers and P. Barford. "Cell vs. WiFi: on the performance of metro area mobile connections". In: *Proceedings of the 2012 ACM conference on Internet measurement conference - IMC '12*. Boston, Massachusetts, USA: ACM Press, 2012, p. 301. ISBN: 978-1-4503-1705-4. DOI: 10.1145/2398776.2398808.

[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, A. Krishnamurthy, S. Hasan, S. Ratnasamy, C. Scott, and V. Sekar. "Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service". In: *SIGCOMM Comput. Commun. Rev.* 42.4 (Oct. 2012), pp. 13–24. ISSN: 0146-4833.

[SHG16] H. Sinky, B. Hamdaoui, and M. Guizani. "Proactive Multipath TCP for Seamless Handoff in Heterogeneous Wireless Access Networks". In: *IEEE Transactions on Wireless Communications* 15.7 (July 2016), pp. 4754–4764. ISSN: 1536-1276, 1558-2248. DOI: 10.1109/TWC.2016.2545656.

[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.

[TB19] V. H. Tran and O. Bonaventure. "Beyond socket options: making the Linux TCP stack truly extensible". In: *IFIP Networking* (2019).

[Ten+97] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. "A survey of active network research". In: *IEEE communications Magazine* 35.1 (1997), pp. 80–86.

[Tes19] Tessares. *5G ATSSS solution: Turning heterogeneous 4G/5G & Wi-Fi networks into a converged network*. May 2019. URL: https://www.tessares.net/solutions/5g-atsss-solution/.

[TK18] B. Trammell and M. Kuehlewind. *The QUIC Latency Spin Bit*. Internet-Draft draft-ietf-quic-spin-exp-01. Published: Working Draft. Oct. 2018.

[Tra19] V.-H. Tran. "Measuring and Extending Multipath TCP". PhD thesis. UCLouvain, Louvain-la-Neuve, Belgium, 2019.

[TW96] D. L. Tennenhouse and D. J. Wetherall. "Towards an active network architecture". In: *ACM SIGCOMM Computer Communication Review* 26.2 (1996), pp. 5–17.

[Vie+18] T. Viernickel, A. Froemmgen, A. Rizk, B. Koldehofe, and R. Steinmetz. "Multipath QUIC: A Deployable Multipath Transport Protocol". In: *2018 IEEE International Conference on Communications (ICC)*. ISSN: 1938-1883. May 2018, pp. 1–7. DOI: 10.1109/ICC.2018.8422951.

[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+11] Z. Wang, Z. Qian, Q. Xu, Z. M. Mao, and M. Zhang. "An Untold Story of Middleboxes in Cellular Networks". In: *Proceedings of the ACM SIGCOMM 2011 conference*. ACM, 2011, p. 13.

[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.

[Wes18] M. Westerlund. "Proposal for adding ECN support to QUIC." 2018.

[WGT98] D. J. Wetherall, J. V. Guttag, and D. L. Tennenhouse. "ANTS: A toolkit for building and dynamically deploying network protocols". In: *Open Architectures and Network Programming, 1998 IEEE*. IEEE, 1998, pp. 117–129.

[WHB08] D. Wischik, M. Handley, and M. Braun. "The Resource Pooling Principle". In: *ACM SIGCOMM Computer Communication Review* 38.5 (2008), pp. 47–52.

[WHS01] G. Wong, M. Hiltunen, and R. Schlichting. "A configurable and extensible transport protocol". In: *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society*. IEEE, 2001, pp. 319–328.

[Wil+14] N. Williams, P. Abeysekera, N. Dyer, H. Vu, and G. Armitage. *Multipath TCP in Vehicular to Infrastructure Communications*. Tech. rep. 2014, p. 15.

[Wir+19] T. Wirtgen, C. Dénos, Q. De Coninck, M. Jadin, and O. Bonaventure. "The Case for Pluginized Routing Protocols". In: *Proceedings of the 27th IEEE International Conference on Network Protocols – ICNP'19*. Chicago, Illinois, USA: IEEE, Oct. 2019, p. 12.

[Wis+11] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. "Design, Implementation and Evaluation of Congestion Control for Multipath TCP". In: *USENIX NSDI*. 2011.

[Woo+14] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. "The CHERI capability model: Revisiting RISC in an age of risk". In: *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE, 2014, pp. 457–468.

[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: *2009 30th IEEE Symposium on Security and Privacy*. ISSN: 2375-1207. May 2009, pp. 79–93. DOI: 10.1109/SP.2009.25.