Flexible QUIC loss recovery mechanisms for latency-sensitive applications

François Michel

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

October 2023

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

Thesis Committee:

Pr. Olivier **Bonaventure** (Advisor)	UCLouvain, Belgium
Pr. Etienne **Rivière**	UCLouvain, Belgium
Pr. Jérôme **Louveaux**	UCLouvain, Belgium
Pr. Anna **Brunström**	Karlstad University, Sweden
Pr. Oliver **Hohlfeld**	University of Kassel, Germany
Pr. Peter **Van Roy** (Chair)	UCLouvain, Belgium

Flexible QUIC loss recovery mechanisms for latency-sensitive applications

by François Michel

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.

I apologize for sending so many packets. If I had more time, I would surely have sent less.

Inspired by Blaise Pascal, Lettres Provinciales, 1657.

Acknowledgments

While this thesis focuses on the impact of loss recovery mechanisms on protocol performance, this acknowledgement section highlights the impact of other humans on the human writer of this document. Their presence had a direct positive impact on my productivity as well as my happiness throughout these years. Therefore, I thank all these people for making my life thrilling and stimulating.

I thank my parents Brigitte Duriau and Jean-Luc Michel, my brother Martin Michel and my brother-in-law Malik Duhaut for their long lasting support, advices and precious family moments. I thank my dear girlfriend Margerie Huet Dastarac for making my life sweet and exciting, for the numerous breakfasts, workoffees, sport sessions and the unforgettable life moments we live together. I thank all my friends, including my previous and current housemates, as well as most of my colleagues who constantly enhance my everyday life. I specifically want to thank Arnaud Devillers for growing and evolving together, attending concerts and festivals and revisiting the world for now more than twenty-five years. Speaking of music, I also thank François Dekeersmaeker for his contagious joy and the already numerous memories we have together. I thank Brandon Naitali for our frequent giggles, Robin Descamps for all his funny stories and our long discussions and Romain Laurent for our made-up expressions and private jokes. I thank my colleagues and friends from the IP Networking Lab and INGI department, Maxime Piraux, Louis Navarre, Thomas Wirtgen, Nicolas Ribowski, Quentin De Coninck, Tom Barbette, Tom Rousseaux, Mathieu Pigaglio, Donatien Schmitz, Christophe Crochet, Gorby Kabasele and Anthony Gego. I thank the jury members, Anna Brunström, Oliver Hohlfeld, Etienne Rivière, Jérôme Louveaux and Peter Van Roy for allocating some of their precious time to evaluate this thesis and providing relevant and insightful comments.

Last but not least, I thank my promoter Olivier Bonaventure for his precious work and life advices, his crazy ideas and for dedicating his time to make the world a better place in many more aspects than computer networking.

Contents

Acknowledgments **i**

Table of Contents **iii**

Preamble **1**

1 Background **9**
1.1 Sending data over a network . . . 9
1.2 Transport protocols . . . 10
1.2.1 The User Datagram Protocol . . . 10
1.2.2 The Transmission Control Protocol . . . 11
1.3 Reliable data transfer . . . 12
1.3.1 Selective-Repeat Automatic Repeat Request . . . 12
1.3.2 Loss detection . . . 13
1.3.3 Congestion control . . . 15
1.3.3.1 Loss-based congestion control . . . 16
1.3.3.2 BBR . . . 17
1.3.4 Transport layer security . . . 18
1.4 The QUIC protocol . . . 18
1.4.1 Reduced and secure connection establishment . . . 19
1.4.2 The QUIC packet . . . 20
1.4.2.1 QUIC packet format . . . 20
1.4.3 Stream multiplexing: avoiding head-of-line blocking . . . 22
1.4.3.1 Frames: control information as part of the encrypted payload . . . 23
1.4.4 QUIC loss recovery . . . 24
1.4.4.1 Loss detection . . . 25
1.4.5 A broad class of applications . . . 26
1.5 Forward Erasure Correction . . . 26
1.5.1 Coding theory primer . . . 27
1.5.1.1 Block codes and Reed-Solomon . . . 27
1.5.1.2 Fountain codes and Random Linear Network Coding . . . 29
1.5.1.3 Systematic codes . . . 30
1.5.2 Protecting network packets . . . 31

1.5.3 Mode of operation . . . 31
1.5.4 FEC as a transport loss recovery mechanism . . . 33

2 QUIC-FEC: A general loss recovery QUIC extension 35
2.1 Forward Erasure Correction for long-delay communications . 35
2.2 Integrating FEC into QUIC . . . 36
2.2.1 Defining and exchanging the source and repair symbols 36
2.2.2 The FEC Framework . . . 37
2.2.2.1 Studied FEC schemes. . . . 38
2.2.3 FEC and the congestion control . . . 39
2.3 Evaluation . . . 40
2.3.1 Methodology . . . 40
2.3.1.1 Experimental design . . . 41
2.3.1.2 Reproducible experiments . . . 42
2.3.2 Results with uniform losses . . . 43
2.3.2.1 Specific IFC use-cases . . . 43
2.3.2.1.1 Direct Air-To-Ground Communication (DA2GC) . . . 43
2.3.2.1.2 Mobile Satellite Service (MSS) . . . 45
2.3.2.2 Experimental design . . . 46
2.3.2.2.1 Large files transfers . . . 46
2.3.2.2.2 Small files transfers . . . 46
2.3.2.2.3 Comparing FEC codes . . . 47
2.3.2.3 Exploring the impact of redundancy overhead 47
2.3.2.4 The importance of recovery notification . . 49
2.3.3 Results with bursty losses . . . 50
2.4 Conclusion . . . 51

3 The interactions between FEC and congestion control 53
3.1 Symbols and packets are conceptually separate data units . . . 53
3.2 Congestion control behaviour upon symbol recovery . . . 55

4 PQUIC: towards really flexible transport protocols 57
4.1 Pluginizing QUIC . . . 59
4.1.1 Pluglet Runtime Environment (PRE) . . . 60
4.1.2 Protocol Operations . . . 61
4.1.3 Attaching Protocol Plugins . . . 63
4.1.4 Interacting with Applications . . . 66
4.2 Extending the loss recovery using protocol plugins . . . 66
4.2.1 Design & implementation . . . 66
4.2.2 Evaluation . . . 67
4.2.3 Plugin Overhead . . . 68

4.3 Validating Plugins . . . 70
4.4 Conclusion . . . 71

5 FlEC: application-tailored loss recovery using protocol plugins 73
5.1 Introduction . . . 73
5.2 Adaptive Forward Erasure Correction . . . 74
5.2.1 Bulk file transfer . . . 76
5.2.2 Buffer-limited file transfers . . . 76
5.2.3 Delay-constrained messaging . . . 77
5.3 FlEC . . . 78
5.3.1 Comparing FlEC and previous work . . . 81
5.4 Implementation . . . 82
5.5 Bulk file transfers . . . 82
5.5.1 Bulk loss recovery mechanism . . . 83
5.5.2 Evaluation FlEC for the bulk scenario . . . 83
5.5.2.1 Experimental design . . . 84
5.5.2.2 Experimenting with a real network . . . 86
5.5.2.3 CPU performance . . . 87
5.6 Buffer-limited file transfers . . . 87
5.6.1 Loss recovery mechanism . . . 88
5.6.2 Evaluation . . . 88
5.6.2.1 FlEC for SATCOM . . . 88
5.6.2.1.1 Transfer completion time and throughput. . . . 90
5.6.2.1.2 Delay-bandwidth tradeoff. . . . 91
5.6.2.2 Experimental design analysis . . . 92
5.7 Delay-constrained messaging . . . 93
5.7.1 Reliability mechanism . . . 94
5.7.1.1 Application-specific API . . . 94
5.7.1.1.1 send_fec_protected_msg(msg, deadline) . . . 94
5.7.1.1.2 next_message_arrival(time) . . . 94
5.7.1.2 Application-tailored stream scheduler . . . 94
5.7.1.3 FECPattern() and ds() for delay-constrained messaging . . . 96
5.7.2 Evaluation . . . 96
5.8 Conclusion . . . 101

6 Starlink: analyzing a new access network 103
6.1 A new wireless access network . . . 103
6.2 Testbed and Measurements . . . 104
6.3 Results . . . 107

6.3.1 Latency . . . 107
6.3.1.1 Latency during inactivity . . . 108
6.3.1.2 Latency under load . . . 109
6.3.2 Characterizing packet losses . . . 110
6.3.2.1 Packet losses during HTTP/3 transfers . . . 110
6.3.2.2 Packet losses during low bitrate transfers . . . 111
6.3.3 Throughput . . . 112
6.3.3.1 Speed test results . . . 112
6.3.3.2 HTTP/3 transfers . . . 113
6.3.4 Browsing Performance . . . 114
6.3.5 Middleboxes and traffic discrimination . . . 115
6.4 Conclusion . . . 116

7 QUIRL: improvements for real applications on real networks 117
7.1 Existing FEC extensions for QUIC . . . 117
7.2 QUIRL Design principles . . . 119
7.2.1 Identifying FEC-protected payloads . . . 119
7.2.2 Serializing the repair symbols . . . 120
7.2.3 QUIRL and congestion control . . . 120
7.2.4 Scheduling the repair symbols . . . 121
7.3 Implementing QUIRL . . . 122
7.3.1 Erasure Correcting Codes . . . 122
7.3.2 WebTransport . . . 122
7.4 Latency-sensitive video streams . . . 123
7.4.1 Redundancy scheduler . . . 123
7.4.2 Reducing the latency of GStreamer RTP flows . . . 124
7.4.3 Starlink setup . . . 126
7.4.4 Real network experiments results . . . 127
7.5 HTTP/3 objects . . . 130
7.5.1 Improving curl's TCT over Starlink . . . 131
7.5.2 Exploring diverse network configurations with Mininet 133
7.6 Conclusion . . . 135

8 Conclusion 137

Introduction

Since its wide deployment in 2017, the QUIC [RFC9000] protocol has progressively gained in popularity over the years. QUIC inherits from decades of protocol design experience and gathers the features of several transport protocols including UDP [RFC768], TCP [RFC791] and SCTP [RFC4960]. QUIC supports diverse transport services that make it suitable to welcome a broader set of applications than these three protocols taken separately. As a result, new applications with different goals and requirements are successively built on top of QUIC to serve diverse use-cases, some of them being strongly impacted by the connection latency. A good example is the MASQUE proxy, used to proxy all kinds of network traffic over QUIC [Sch23]. Previous works have also defined QUIC mappings for the RTP protocol [OE23]. More recently, the IETF chartered the Media Over QUIC working group to specify live media transport atop QUIC [Cur+23]. Currently, the QUIC protocol only relies on retransmissions to recover from packet losses [RFC9002]. Being efficient in terms of bandwidth, retransmissions work well for throughput-based applications such as large HTTP and file transfers. However, the protocol needs at least one round-trip time to detect lost packets and retransmit them. This added latency to recover from packet losses is undesirable for applications with strong latency requirements. This thesis therefore starts with the following problem statement.

> *The latency increase of retransmissions-based loss recovery mechanisms may prevent QUIC from being suitable for latency-sensitive applications on lossy networks.*

Our goal is to make the loss recovery mechanism of the QUIC protocol more general. More specifically, we want the protocol to support a broader class of applications by being able to recover faster from network losses using Forward Erasure Correction (FEC) techniques when needed. The contributions of this thesis are the following:

1. We study the overhead and congestion control impacts of adding FEC to the QUIC protocol that is also used for throughput-intensive workloads.
2. We make it easier to implement and deploy QUIC extensions such as FEC on a per-connection basis.

3. We propose a flexible QUIC loss recovery mechanism, allowing to tailor the loss recovery behaviour to the application requirements.

4. We provide an analysis of the Starlink network, showing that packet losses are still common nowadays.

5. We finally provide an efficient implementation of our flexible loss recovery mechanism and integrate it with two popular network applications. We show that a flexible QUIC loss recovery mechanism can provide latency improvements for real applications on real lossy networks.

FEC is not new and is already used at different levels of the network stack, whether it is performed at the physical layer to recover from bit errors or in applications themselves when they have strict latency requirements. What is less common is to see FEC being part of a multi-purpose protocol like QUIC. Intuitively, recovering faster from packet losses seems appealing in many scenarios and looks like an always-winning approach. However, a technique like FEC in the transport layer comes with its own lot of misconceptions and one needs to dismantle them with care before starting to get real benefits from the technique. An interesting fact is that an elementary form of FEC had already been implemented and evaluated at a large scale by Google even before the beginning of this thesis. The technique used by Google in this first study led to poor results in real-world experiments [Lan+17]. FEC was then evicted from the roadmap of the first version of QUIC and put aside for later.

In order to make it work, we first had to identify the different reasons behind these poor results. Aside the simple FEC techniques that were used, one of the most important reasons is probably that every application does not benefit the same way from FEC. Sending redundancy to protect file transfers simlarly to real-time media is going to provide bad results in most scenarios, even in a lossy network. This is why we rapidly reached the conclusion that FEC must be adapted to the application requirements if we want it to work in a multi-purpose transport protocol.

An interesting question is why implementing FEC at the transport layer ? Shouldn't it be left at the physical layer ? First, when applied at the physical layer, FEC can only recover from losses caused by the physical impediments of the channel itself. Losses due to congestion cannot be corrected. While it might seem counter-intuitive at a first glance to recover from congestion-induced losses using FEC, it is a good way to avoid the latency penalty of such losses during short-timed congestion events. Furthermore, we will see throughout this thesis that a FEC-enabled sender may obtain latency benefits while sending at the same rate as a regular QUIC sender, adapting its congestion window and sending rate in the exact same way.

One might also wonder whether FEC should not better be implemented by applications themselves. Indeed, application can always implement the transport features they need to optimize the data transfer to their own requirements. However, the price to pay is the increased implementation complexity. FEC can be and is sometimes implemented inside applications, but this claim also holds for flow control, congestion control QUIC stream scheduling or even multipath scheduling. Yet, these features are all part of the QUIC protocol itself and allow many applications to stay straightforward by reusing the existing transport services. In some parts of this thesis, we explore a hybrid approach, where the FEC signaling, encoding and decoding is performed by the transport protocol while the application can provide its own tailored redundancy scheduler that fit its needs. On top of that, it is interesting to observe that the transport layer acts at as an interesting rendez-vous point gathering knowledge from the network (e.g. delay, loss rate, congestion state) and the application requirements. With its broad set of features, a QUIC sender has to go through a large set of scheduling tasks. Among others, it must schedule the stream to send first, the frames to pack in a packet, and with the recent multipath extension, the network path on which a packet will be sent [Liu+23]. Implementing the FEC behaviour inside the transport protocol allows it to take clever decisions, being aware of all the different kinds of constraints that it has to fullfill. For instance, a conjoint scheduling of the sending path and the FEC redundancy may lead to significantly better results than performing these two tasks separately, as it allows recovering packet losses occuring on one path by sending FEC on the other. Finally, we show in this thesis that FEC can be beneficial for the performance of the QUIC protocol itself as it can be used to unblock the flow control mechanism earlier than retransmissions in buffer-limited scenarios. This cannot be done at the application level.

The QUIC specification changed significantly since the beginning of this thesis. There also exist different implementations of the QUIC protocol, each one having its own pros and cons. Some of them did not exist at the beginning of this thesis or were at an embryonic state. The different chapters therefore present different iterations of our solution using different QUIC implementations. The solution proposed in this thesis is built incrementally throughout the chapters, studying different aspects of the problem one after the other.

- Chapter 1 introduces the different concepts needed to clearly understand the system and problematics we work on. We introduce the base transport protocols concepts and take a closer look at the QUIC design. We then discuss the basics of Forward Erasure Correction and present the existing algorithms we leverage in this thesis.

- Chapter 2 presents QUIC-FEC, our first extended loss recovery mechanism for QUIC. It studies and compares several erasure correcting codes,

proposes a base protocol design for FEC-protected QUIC connections and studies interesting aspects to consider when implementing FEC in a transport protocol. This first version does not adapt the sending of redundancy to the network characteristics and clearly identifies the limits of this approach for throughput-based applications such as HTTP file transfers. This chapter provides a first basis on top of which the following works bring flexibility and adaptability.

- Chapter 3 extends the discussion on the interactions between FEC and congestion control started in Chapter 2. The work of Chapter 2 led us to discuss and present our solution at the Internet Engineering Task Force. We contributed actively to the Coding for efficient NetWork Communications Research Group (NWCRG) and iteratively developed there an internet draft dedicated to the joint use of FEC and congestion control. This draft then became the RFC9265 informational document. Chapter 3 focuses on a few aspects of this RFC.

- Chapter 4 presents PQUIC, a QUIC implementation allowing redefining protocol behaviours on a per-connection basis by inserting application-defined protocol plugins running in a sandboxed environment. Using PQUIC, some connections can use a FEC-enabled loss recovery mechanism while others can stick to the regular QUIC loss recovery. PQUIC also eases the deployment of such extensions, allowing servers to plug the FEC extension directly on the client on a specific connection, without the need to wait for web browsers and mobile clients to be updated to use FEC. This chapter proposes a first FEC extension fully implemented using protocol plugins, showing the flexibility of the approach. The evaluation also shows that there is an interest in adapting the amount of redundancy not only to the network but also to the underlying application. This is an idea we pursue in the following chapters.

- Chapter 5 proposes Flexible Erasure Correction (FlEC), the next iteration of our solution. Entirely based on protocol plugins, FlEC proposes a general FEC-enabled loss recovery mechanism adaptive to both the network loss characteristics and the application's traffic pattern and requirements. With only a few lines of code and thanks to protocol plugins, applications can define a FEC redundancy scheduler that closely fits their needs. We show the benefits of FlEC in three different scenarios, including real-time video transfers. We also discuss the limits of the PQUIC approach upon which FlEC relies. Indeed, while simulation results clearly show the benefits of FlEC, we observe that the computational overhead of the framework deteriorates the performance when applied on real networks. These observations pave the way for the

following chapters focused on providing benefits for real applications over real networks.

- Chapter 6 studies the characteristics of a Starlink vantage point we rent in Belgium. We specifically focus on loss events, their patterns and duration. We find that loss events are numerous and that an efficient implementation of a FEC-enabled loss recovery mechanism may bring significant improvements to latency-sensitive real-world applications, motivating the chapter that follows.

- Chapter 7 starts from the performance issues of FlEC and proposes QUIRL, an efficient design and implementation for our FEC-enabled loss recovery. Built on a production-ready and mature QUIC implementation, we show that QUIRL shows significant latency improvements for deployed network applications, namely curl and GStreamer over the Starlink network studied in Chapter 6.

The final prototype presented in Chapter 7 constitutes an efficient implementation of our extended loss recovery mechanism, adaptive to both the application and the network characteristics. Being based on the version 1 of QUIC, this implementation will be maintained up-to-date and improved over time. We encourage researchers and users around the world to experiment with our work and reach out to us for help and interesting findings. The source code, experimental scripts and results of every piece of work produced during this thesis is publicly available and listed in the Artefacts section hereunder.

Several other articles have been published during this thesis but are not discussed in this manuscript [Coh+21; Kuh+22; MB21b; Pir+22; Bon+20; NMB21].

Bibliographic notes

Conference Publications

1. F. Michel, Q. De Coninck, and O. Bonaventure. "QUIC-FEC: Bringing the benefits of Forward Erasure Correction to QUIC". in: *2019 IFIP Networking Conference (IFIP Networking)*. IEEE. 2019, pp. 1–9.

2. 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*. 2019, pp. 59–74.

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

Journal Publications

1. N. Kuhn, E. Lochin, F. Michel, and M. Welzl. *Forward Erasure Correction (FEC) Coding and Congestion Control in Transport*. RFC 9265. July 2022. DOI: 10.17487/RFC9265. URL: https://www.rfc-editor.org/info/rfc9265.

2. F. Michel, A. Cohen, D. Malak, Q. De Coninck, M. Médard, and O. Bonaventure. "FlEC: Enhancing QUIC With Application-Tailored Reliability Mechanisms". In: *IEEE/ACM Transactions on Networking* 31.2 (2023), pp. 606–619. DOI: 10.1109/TNET.2022.3195611.

Under submission

1. F. Michel and O. Bonaventure. "QUIRL: efficiently reducing QUIC applications latency using Forward Erasure Correction". In: (2023).

Artefacts

1. The code for QUIC-FEC (Chapter 2) and the scripts for the experiments [Mic23d] as well as `ebpf_dropper` [Mic23a] are publicly available.

2. The code for PQUIC (Chapter 4) and the different plugins implemented is publicy available [PquicRepo].

3. Simulations scripts and the code of FlEC (Chapter 5) are publicly available [Mic23b; Mic23c].

4. All the data gathered to compute the results discussed in Chapter 6 are publicly available [Mic+23b]. This includes pings, traceroute, Tracebox, speed test and BrowserTime results as well as more than 530 Gigabytes of QUIC packet captures along with their encryption keys.

5. The code for QUIRL (Chapter 7), the curl client, the RTP relay as well as the packet captures and decryption keys are publicly available [Mic23e].

Miscellaneous Contributions

1. A. Cohen, H. Esfahanizadeh, B. Sousa, J. P. Vilela, M. Luis, D. Raposo, F. Michel, S. Sargento, and M. Medard. "Bringing network coding into SDN: Architectural study for meshed heterogeneous communications". In: *IEEE Communications Magazine* 59.4 (2021), pp. 37–43.

2. N. Kuhn, F. Michel, L. Thomas, E. Dubois, E. Lochin, F. Simo, and D. Pradas. “QUIC: Opportunities and threats in SATCOM”. in: *International Journal of Satellite Communications and Networking* 40.6 (2022), pp. 379–391.

3. F. Michel and O. Bonaventure. “Packet delivery time as a tiebreaker for assessing Wi-Fi access points”. In: *IAB Workshop on Measuring Network Quality for End-Users*. 2021.

4. M. Piraux, T. Barbette, N. Rybowski, L. Navarre, T. Alfroy, C. Pelsser, F. Michel, and O. Bonaventure. “The multiple roles that IPv6 addresses can play in today’s internet”. In: *ACM SIGCOMM Computer Communication Review* 52.3 (2022), pp. 10–18.

5. O. Bonaventure, Q. De Coninck, F. Duchêne, A. Gego, M. Jadin, F. Michel, M. Piraux, C. Poncin, and O. Tilmans. “Open educational resources for computer networking”. In: *ACM SIGCOMM Computer Communication Review* 50.3 (2020), pp. 38–45.

6. L. Navarre, F. Michel, and O. Bonaventure. “SRv6-FEC: bringing forward erasure correction to IPv6 segment routing”. In: *Proceedings of the SIGCOMM’21 Poster and Demo Sessions*. 2021, pp. 45–47.

Background

1

This chapter explains the different concepts that this thesis relies on. It also describes the current state-of-the-art techniques for end-to-end data transmission over the Internet.

1.1 Sending data over a network

A significant fraction of the electronic devices produced nowadays are "network capable": they are able to communicate with other devices using specific pieces of hardware called network adapters. These adapters rely on different communication technologies depending on the device and its intended use. For instance, personal computers can be connected to others using RJ-45 cables or wirelessly using Bluetooth or Wi-Fi adapters. Smartphones can transmit data over a cellular network in addition to Wi-Fi. Other devices can also communicate with satellites.

All these communication technologies have practical limitations that make it unfeasible to directly connect two devices solely through their network adapters if they are kilometers away from each other. The standard solution is to rely on a *network* to connect devices regardless of their geographic position. In a network, each host is connected to one or more *routers*. The role of a router is to forward the data towards its destination. The router will either forward the data directly to the destination if they are connected to each other or forward the data to another router that is closer to the destination when they are not in direct reach.

In order to operate correctly, devices and routers need to agree on how to identify the connected hosts of the network. The Internet Protocol (IP) [RFC791] defines semantics to transmit data between hosts identified by an IP address. An IP address is a fixed-size integer whose size depends on the version of the protocol (32 bits for IP version 4 and 128 bits for IP version 6 [RFC8200]).

Routers in an IP network can only handle data units of limited size called *packets*. The IP protocol defines the Maximum Transmission Unit (MTU) as the maximum size of an IP packet payload.

1.2 Transport protocols

Routers of an IP network essentially provide a best effort forwarding service for MTU-sized packets of the hosts. The forwarded packets can be lost, duplicated or altered due to impairments in the network (such as electromagnetic interferences, congestion in the network or power outages). Hosts that need more guarantees on their data transfer can rely on *transport protocols* that operate on top of IP. Transport protocols provide services to applications running on the hosts. The Linux kernel implements several transport protocols and provides their services through the socket API [SOCKET]. An application running on Linux can specify the kind of service it wants by setting the *socket type* when using the socket API, leading to different transport protocols being used. The most common socket types are DGRAM and STREAM.

- DGRAM: The application can send datagrams with a limited size in a best-effort manner. This is close to the regular services that IP networks natively provide. The application can optionally ask for not receiving packets that have been corrupted during their transit through the network.

- STREAM: The application has access to a bidirectional bytestream. There is no size limitation to the data exchanged. The data are transmitted reliably and in-order to the peer.

In the current Linux kernel, the underlying transport protocols are the User Datagram Protocol (UDP) [RFC768] for DGRAM sockets and the Transmission Control Protocol (TCP) [RFC791] for STREAM sockets. UDP and TCP packets are placed in the payload of IP packets.

1.2.1 The User Datagram Protocol

The User Datagram Protocol (UDP) provides a straightforward datagram delivery service to applications connected to an IP network. The UDP specification defines the packet format that peers need to implement in order to exchange datagrams using UDP. The packet format is depicted in Figure 1.1.

The *Source port* and *Destination port* fields identify the application sending and receiving the packet. This allows having several UDP endpoints using a single IP address. The *Payload length* field indicates the length of the application payload carried by the packet. The *Checksum* field provides a protection against transmission errors altering the packet payload when transiting in the network. Upon the reception of a packet, a UDP receiver can directly deliver the packet payload to the application, independently of the fact that the previous packet was already received or not.

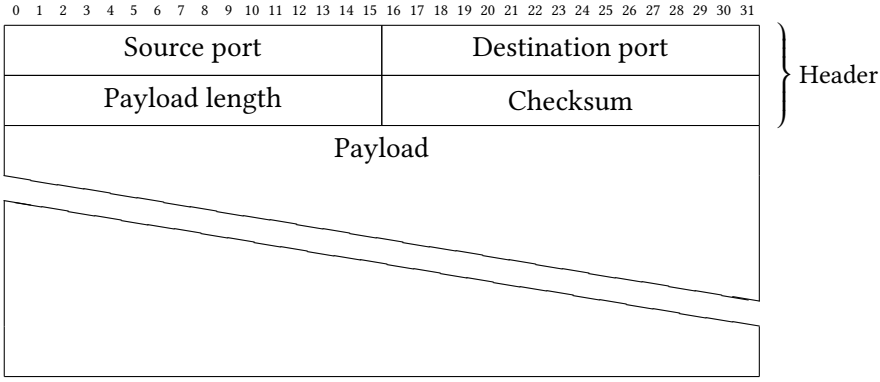


Figure 1.1: UDP packet format.

UDP is a *connectionless* transport protocol. It does not maintain any protocol state between the two communicating endpoints.

Recently, extensions have been proposed for the UDP packet format to attach options and metadata to a UDP packet [Tou22], allowing for instance to redefine the UDP checksum computation or authenticating cryptographically the packet.

1.2.2 The Transmission Control Protocol

The Transmission Control Protocol (TCP) allows providing a reliable bidirectional bytestream abstraction to the application. An application can send an arbitrary number of bytes on a STREAM socket and TCP will ensure that they will all be delivered in sequence to the receiver regardless of the presence of network impairments such as packet loss, corruption or reordering. The TCP packet format is depicted in Figure 1.2.

For the same reasons as UDP, TCP integrates a source and destination port as well as a checksum in its packet format. The *Sequence Number* field indicates the byte position of the packet payload in the bytestream. The *Acknowledgement Number* field is used by the TCP receiver to indicate the next expected sequence number for the stream to be delivered in sequence. There is no missing byte with a sequence number smaller than the acknowledgement number. The *Data Offset* field indicates the length of the TCP header, expressed in 32-bits word units. The *Window* field indicates the *receive window* of the receiver, indicating the amount of bytes that can be stored in the receiver's buffers starting from the acknowledgement number. Finally, the TCP header allows encoding one or several options to customize the protocol. Due to limitations of the TCP header size expressed in the Data Offset field (60 bytes), the Options field size is limited to a maximum of 40 bytes. This is an important

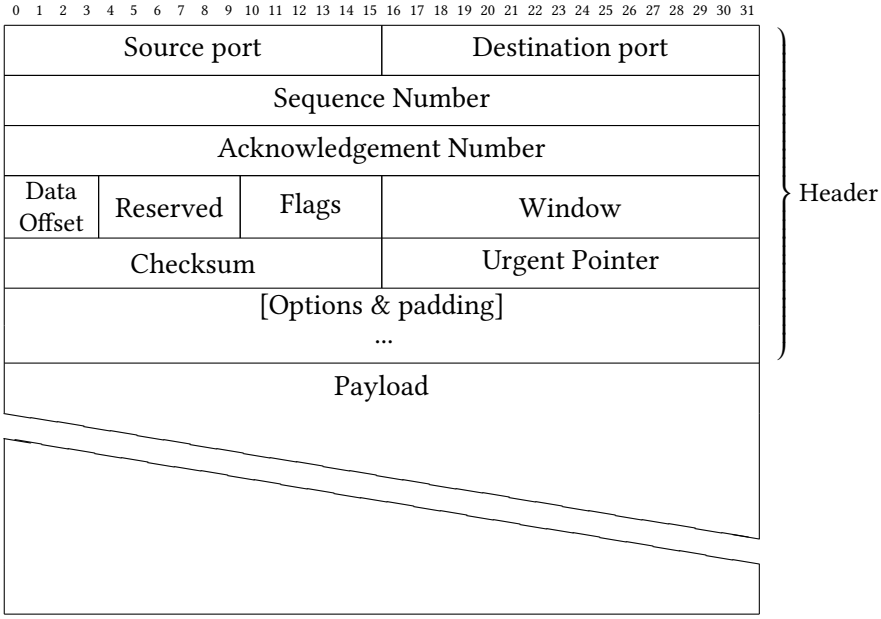


Figure 1.2: TCP packet format.

limitation of the TCP protocol reducing its extensibility, leading to recent but not-yet-standardized propositions of extending the Data Offset field [TE22].

TCP is a *connection-oriented* transport protocol: a stateful connection is created between the two communicating endpoints. Maintaining a connection state is needed by TCP to provide a reliable data transfer to the application.

1.3 Reliable data transfer

TCP ensures a fully reliable delivery of the application data exchanged by the peers and therefore has to cope with network impairments such as packet loss or corruption. In order to do so, transport protocols rely on the retransmission of the lost or corrupted data by implementing a *loss recovery mechanism*. The loss recovery mechanism defines strategies to identify the lost data from the receiver's acknowledgements and retransmit the missing parts of the bytestream. There exist several retransmission strategies of lost data. In this thesis, we focus on the Selective-Repeat Automatic Repeat Request (SR-ARQ) mechanism implemented by most recent transport protocols.

1.3.1 Selective-Repeat Automatic Repeat Request

In SR-ARQ, the receiver indicates the sequence numbers of the correctly received packets by sending an acknowledgement (ACK) to the sender. The

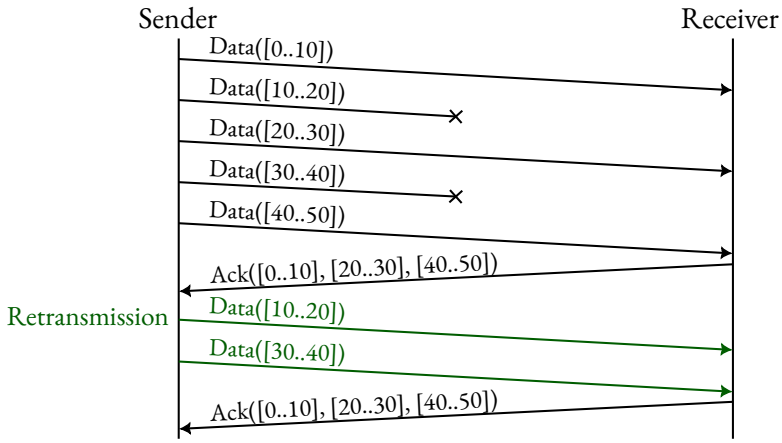


Figure 1.3: SR-ARQ recovering from packet losses.

sender identifies the lost packets using its *loss detection mechanism* and then selectively repeats every missing piece of data to the receiver. Figure 1.3 shows a generic SR-ARQ mechanism recovering from packet losses when transmitting 60 bytes on the network. In this example, the second and fourth packets are lost. The receiver indicates the received data in its acknowledgement. Once it identifies that some data are missing, the sender retransmits the lost packets to ensure the reliable delivery of the sent data.

Recent versions of TCP typically implement an SR-ARQ mechanism to ensure the reliable delivery of the bytestream. However, the TCP header only allows indicating the highest byte received in sequence, through its Acknowledgement Number field. In the example of Figure 1.3, a TCP peer cannot indicate the correct reception of bytes 20 to 30 and 40 to 50 only using the acknowledgement number. Modern TCP implementations therefore rely on the Selective Acknowledgements (SACK) TCP option [RFC2018] in order to indicate a discontinuous sequence of received data. Sadly, the restricted size of the TCP options (40 bytes) makes it hard to encode large SACK blocks, making TCP less efficient than a state-of-the-art SR-ARQ implementation.

1.3.2 Loss detection

The sender needs a mechanism to identify the packets lost in the network and avoid retransmitting packets that have simply been delayed or reordered. The sender can therefore not entirely rely on the gaps present in the received acknowledgements as some of them could be caused by packet reordering as illustrated in Figure 1.4. In this example, the second packet is reordered by the network and is received after the third packet. Directly retransmitting the second packet upon the reception of the ACK would trigger an unnecessary

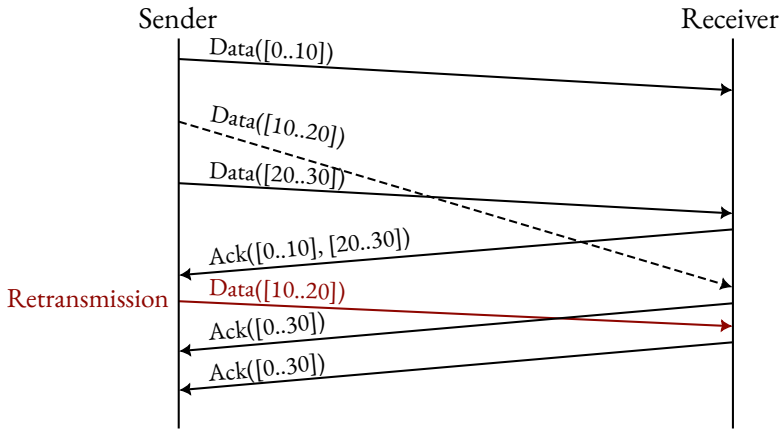


Figure 1.4: Illustration of a spurious retransmission.

retransmission. This phenomenon is called *spurious retransmission* and causes a waste of bandwidth as the retransmission could be avoided by waiting for the acknowledgement of the reordered packet.

Over the years, the TCP protocol has been extended with several mechanisms to avoid spurious retransmissions, such as relying on duplicated SACKs [RFC2883; RFC3708], *Fast Retransmit* [RFC5681; RFC6675] or more recently *Recent Acknowledgements (RACK)* [RFC8985]. On modern TCP implementations, a transmitted packet will be deemed lost if at least one of these conditions hold:

- The retransmission timeout (RTO) for this specific packet has expired without having received an acknowledgement for the packet. In current Linux kernel versions, the minimum value for the TCP RTO is set to 200 milliseconds, nearly one order of magnitude larger than the typical round trip time of modern Internet connections.

- The packet is identified as lost following the RACK heuristic [RFC8985]: packets with a newer sequence number have been acknowledged through selective acknowledgements for more than one quarter of the connection round-trip time. This additional time is used to wait for reordered packets to be acknowledged.

A lost packet will thus take more than one round-trip time to be detected as such[1]. Packet losses will thus cause an additional latency that can be detrimental for latency-sensitive applications running on TCP.

[1] It takes $\frac{5}{4} * RTT$ in the best case when RACK is enabled.

1.3.3 Congestion control

Due to the gain in popularity of computer networks and the rising number of connected devices, the amount of exchanged data increased gradually. This progressively augmented the pressure on the network routers and led to the first *congestion collapse* in 1986 [Jac88]. During this phenomenon, the connected devices repeatedly sent more data over the network than the amount that could be handled by the routers, leading to an important increase of the network delay due to the routers being overloaded. This in turn leads to a high number of retransmissions from the TCP endpoints, a good part of them being spurious as the network delay exceeded the protocol retransmission timeout. This phenomenon can result in a drop of throughput of several orders of magnitude as the traffic consists mostly in spurious retransmissions. Congestion collapse led directly to the design of *congestion control* solutions intended to avoid this phenomenon and improve the performance of transport protocols on loaded networks [Jac88].

Congestion control algorithms are designed to be implemented on the sending endpoint of transport protocols. The main principle is to lower the data sending rate in reaction to a *congestion signal* from the network. Early research contributions about congestion control called for making this signal explicit by modifying the forwarded packets on the congested routers [JR88]. As these solutions required to update every router on the network, solutions relying on an implicit congestion signal such as packet losses or network delay increase were therefore preferred in practice [Jai86; Jac88]. Explicit Congestion Notification (ECN) has however been proposed and standardized for IP networks [RFC3168] but is still not globally deployed. Currently, congestion control algorithms still mostly rely on implicit congestion signals to infer the congestion state of the network.

The available congestion control algorithms are manifold. They vary in the implicit congestion signals they use and the way they react to it. Most of them keep track of a *congestion window*, limiting the *bytes in flight*, i.e. the amount of bytes that can be transiting through the network at the same time. New packets can be sent as long as the amount of bytes in flight is smaller than the congestion window. When a packet gets acknowledged or is deemed lost, its bytes are subtracted from the bytes in flight, freeing up some space for new packets to be sent. The goal of a congestion control algorithm is to keep the congestion window close to the *bandwidth-delay product* (BDP) of the link. A congestion window larger than the BDP induces congestion on the network while a congestion window smaller than the BDP under-utilizes the available bandwidth. The different congestion control algorithms continuously update the congestion window in reaction to the congestion state of the link inferred by the congestion signals.

1.3.3.1 Loss-based congestion control

The first standardized congestion control mechanism [RFC5681] is based on Van Jacobson's work [Jac88] and uses packet loss as congestion signal under the hypothesis that most packet losses are due to routers discarding packets when congested. Its mode of operation can be decomposed in two phases: *slow start* and *congestion avoidance*.

The slow start phase ensures starting the transfer with a low sending rate in order to cope with low bandwidth networks. The transfer therefore starts with a low initial congestion window[2] and then doubles the window at every round-trip time until the first packet loss is seen, after which the congestion window is halved and the sender enters the congestion avoidance phase. During congestion avoidance, the congestion window follows an additive increase / multiplicative decrease pattern [JR88]. It is increased by one full packet after every RTT without congestion. Upon a light congestion event (receiving three duplicate acknowledgements consecutively), the congestion window is halved while it is set to one packet upon a strong congestion event (the packet retransmission timer has expired). This mechanism is often referred as the *New Reno* congestion control, named after the BSD release it was first implemented in. The *Cubic* loss-based congestion control algorithm [RX05; HRX08], is used as the default congestion control algorithm in the Linux kernel since 2006 [LinCub]. Its main difference with New Reno is its window growth function during the congestion avoidance phase that follows a cubic function whose inflection point (w_{max}) is equal to or lower than the value of the congestion window during the previous loss event. This ensures a slow window increase around W_{max} and a faster window increase once W_{max} is passed without loss event. The window growth of Cubic is also independent of the RTT, allowing a faster bandwidth probing over long-delay links compared to New Reno. Figure 1.5 illustrates the behaviour of New Reno and Cubic[3] during congestion avoidance under the presence of sporadic loss events represented by dashed lines. First, we can see that the window increase of New Reno is slower when the RTT is larger while Cubic is not affected by the RTT. Second, we can observe that the window reduction upon loss event is lighter with Cubic (30%) than New Reno (50%), providing a higher overall throughput for Cubic.

The second part of the graphs shows that the congestion window of loss-based congestion controls will collapse unnecessarily if the loss events represented on the Figure are unrelated to network congestion. Loss-based congestion controllers therefore perform badly over networks exposing a

[2]The recommended initial window size from RFC5687 is 3 packets. Nowadays, the Linux kernel uses an initial congestion window of 10 packets.

[3]Cubic is configured with the recommended parameters proposed by [RFC8312bis].

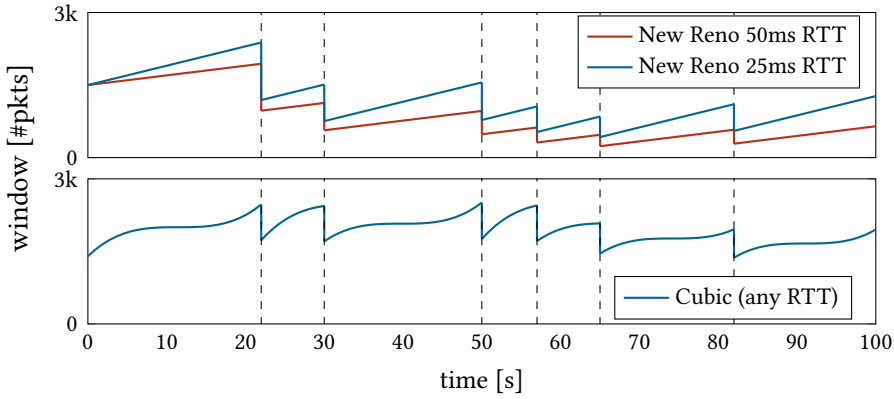


Figure 1.5: Loss-based congestion control window evolution.

high percentage or congestion-unrelated losses [Hua+13]. While this was uncommon when wired network access was predominant, non-congestion-induced losses have become more frequent since wireless network accesses gained in popularity.

1.3.3.2 BBR

Several congestion control algorithms have been designed to address this problem [AB18; Car+16a; Car+22]. Pushed by Google researchers in 2016 [Car+16a] and integrated in the Linux kernel since release 4.9 [LinBBR], BBR is currently the most notorious general-purpose congestion control mechanism that does not entirely rely on packet loss to infer the state of the network. Instead of reacting to congestion signals once the network is congested, BBR continuously adjusts its sending rate based on bandwidth estimations of the network. To compute these estimations, the BBR sender regularly samples the data receive rate by looking at the amount of acknowledged data over time. Most of the time, the BBR sender is in a state where the sending rate exactly matches the estimated bandwidth (*Cruise*). BBR then occasionally ensures to fill the network pipe and attempts to increase the observed receive rate (*Refill and Probe Up*). To this end, the sender inflates its sending rate to 1.25 times the current bandwidth estimation. This state may increase the size of the bottleneck router queues by sending at a higher rate than the available bandwidth. The sender thus finally lowers its sending rate below the estimated bandwidth during a short period of time in order to empty any queue that the probing phase has built on the routers (*Probe Down*). After that phase, the sender gets back to its *Cruise* state with its new bandwidth estimate. Figure 1.6 illustrates different states and behaviours of a BBRv2 sender after a sudden increase of the available bandwidth. The dashed line represents the available bandwidth.

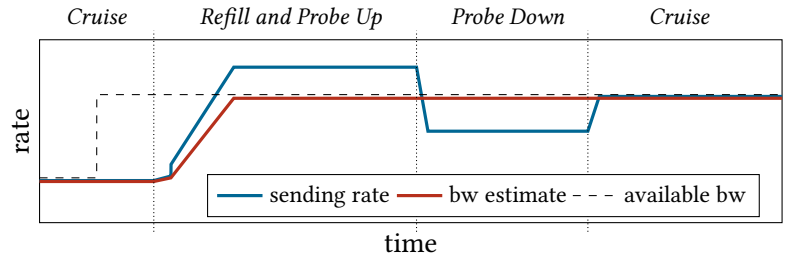


Figure 1.6: Evolution of the BBR sending rate after an increase of the available bandwidth.

The *bw estimate* is computed by the sender by observing the data receive rate from the received acknowledgements. While totally ignored in its first version, the second version of BBR reintroduces packet losses as a congestion signal in its algorithms, although their effect on the sending rate remains minor while the loss rate stays under 2% [Car+22].

1.3.4 Transport layer security

Having been designed forty years ago, TCP and UDP do not provide any cryptographic security feature such as encryption and authentication. These protocols are therefore vulnerable to communication eavesdropping and message forgery, exposing the applications to attackers on the Internet. To prevent these attacks and make TCP and UDP usable publicly, security solutions have been developed on top of these transport protocols to encrypt and authenticate the application payloads they carry. This is the purpose of the Transport Layer Security (TLS) protocol [RFC8446]. TLS allows two endpoints to negotiate cryptographic material (algorithms and keys) and encrypt and authenticate the application payload by encapsulating it inside *TLS records*. The encrypted TLS records are then sent over the TCP connection. By doing so, no attacker can access nor modify the application payload exchanged over the network. However, as TLS exclusively protects the application payload, the TCP header is neither encrypted nor authenticated and can still be altered by any intermediate on the network.

1.4 The QUIC protocol

First presented in 2013 [QUICBlog], the QUIC protocol has been designed to provide a modern transport protocol with improved performance and security features, with a focus on HTTP [RFC9000; RFC9114].

Started as an evolution of SPDY [Bel+], the precursor of HTTP/2 [RFC9113], QUIC combines in a single protocol the mechanisms usually found in TCP, TLS

and HTTP/2. In contrast with TLS/TCP, QUIC encrypts most of the protocol control information in addition to the application payload to prevent pervasive monitoring and interferences from on-path attackers. Since it is built over UDP, QUIC is easier to update than TCP. Indeed, QUIC implementations can be included as libraries inside applications that are regularly updated and do not need support from the operating system outside access to UDP sockets.

Given the positive results obtained by Google with QUIC [Lan+17], the IETF created a dedicated working group in 2016 to standardize the protocol starting from Google's initial design [IT16]. Measurements studies show that a growing number of servers now support QUIC [Rüt+18; Kos+22; QUICDNS] but also that the QUIC traffic grows [Lan+17; Tre+18], with studies showing more than 65% of HTTP requests in Belgium being performed with QUIC as of writing this thesis [Lab; Geo].

One of the strengths of the QUIC protocol is its extensibility. A QUIC packet payload contains a sequence of *frames*, each one being handled independently by the protocol. The STREAM frame transports application data. The ACK frame acknowledges the received packets to the sender. New types of frames can easily be added to the protocol.

Released and standardized more than thirty years after TCP, QUIC inherits from decades of experience and improvements of older transport protocols. As QUIC is the foundation this thesis builds upon, this section goes through several QUIC features and highlights how it differs from classical transport protocols such as TCP.

1.4.1 Reduced and secure connection establishment

Before actually exchanging application payload on the internet, connection-oriented transport protocols generally want to ensure a series of properties for the connection. For secured communications using TLS, cryptographic material must be exchanged by the endpoints to agree on encryption algorithms and keys. QUIC endpoints also need to ensure they run compatible protocol versions and understand each other's protocol extensions (e.g. QUIC datagrams). Finally, being publicly reachable, the server may first want to ensure that the client actually owns its IP address and is not performing an attack by spoofing the IP address of someone else. All these properties are performed during the protocol *handshake* that takes place at the connection startup. Unsuccessful handshakes will prevent the connection establishment. The TCP handshake requires one round-trip time to complete. For negotiating the cryptographic keys and algorithms, the version 1.3 of TLS requires an additional round-trip. This implies that applications using TCP+TLS have to wait for two full round-trip times before their payload can actually be sent. The TCP Fast Open (TFO) extension allows sending application data before

the TCP handshake is completed [RFC7413]. TLS 1.3 also permits sending data before the completion of the cryptographic handshake by using the *early data* mechanism: TLS endpoints can exchange cryptographic material over a session that will be used afterwards to send application data over subsequent sessions before the completion of the TLS handshake.

On its side, QUIC integrates TLS as part of its design and performs both transport and TLS handshakes at the same time in one round-trip, halving the time required for the connection establishment [RFC9001]. Finally, QUIC allows 0-RTT connection establishment using the TLS early data mechanism. 0-RTT connection establishment is already supported by most QUIC implementations while TCP fast open still struggles to be widely deployed [Paa16]. This makes QUIC connection establishment significantly faster than TCP in general.

1.4.2 The QUIC packet

The QUIC protocol is built atop UDP. This means that QUIC packets are placed in the payload of UDP packets. The advantage is twofold. First, UDP packets can be sent by any unprivileged application on common operating systems such as Linux. This means that implementing and running the QUIC protocol requires neither kernel modifications nor specific privileges on the system. Second, having existed for more than forty years and being the favoured transport protocol for real-time media communication, the UDP protocol has proven itself to traverse the Internet without issue. A new protocol developed directly on top of IP would have to pass through many kinds of network nodes called middleboxes, some of them being known to drop packets containing unknown protocols or protocol options. Building QUIC atop UDP allows passing through most middleboxes that already allow UDP traffic.

1.4.2.1 QUIC packet format

There are two kinds of QUIC packet headers: the *long header* and the *short header* [RFC9000]. While the long header is only used for the first few packets exchanged for connection establishment, the short header is used for most of the connection lifetime. The short header packet format for the version 1 of QUIC is depicted in Figure 1.7. It contains significantly fewer fields than the TCP header. This is because QUIC encodes most of its control information inside frames that are part of the QUIC *encrypted payload*. The first bit (H) indicates the header type (1 for long, 0 for short). The second bit (Q) is always set to 1, allowing quickly identifying non-QUIC packets. The third bit (S) is the *Spin bit*. This bit varies once per round-trip, allowing the network nodes to passively measure the RTT of QUIC connections. The *Rsv* bits are currently reserved. *K* is used to determine the cryptographic context used to encrypt

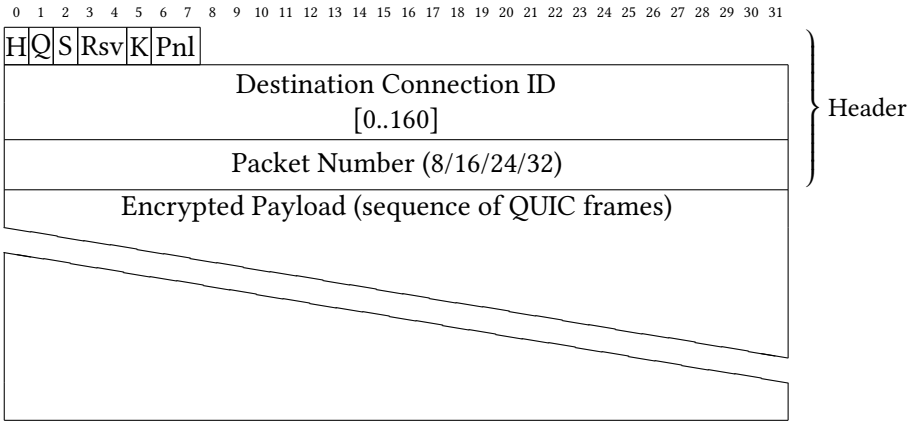


Figure 1.7: QUIC short header packet format.

the packet. *Pnl* is used to indicate the length of the *Packet Number* field: 8, 16, 24 or 32 bits.

The *Destination Connection ID* is used to uniquely identify the QUIC connection this packet belongs to. To do so, TCP and UDP both only rely on the $(IP_{src}, IP_{dst}, Port_{src}, Port_{dst})$ 4-tuple. Connection IDs add one level of multiplexing, allowing several QUIC connections to run on the same 4-tuple when the connection ID length is not zero.

The *Packet Number* field contains the least significant bits of the 62-bits packet number. The packet number uniquely identifies a QUIC packet in a QUIC connection. Two different QUIC packets cannot have the same packet number. The Packet number is monotonically increasing: it increases by at least one for each newly sent packet, ordering the packets by the time they were sent. QUIC packets are never retransmitted as is. If the content of a QUIC packet needs retransmission, this content is placed in a new packet with a dedicated packet number.

The remaining part of the QUIC short header packet is the *Encrypted Payload* and spans the remaining UDP payload. The QUIC payload is encrypted and authenticated, ensuring end-to-end integrity and confidentiality of the data it contains. This prevents any inspection or modification of the packet payload by attackers or network middleboxes. Aside from the security guarantees it provides, it also mitigates the *ossification* problem plaguing the extensibility of cleartext protocols on the Internet such as TCP [Hon+11; Rai+12; ED19], where middleboxes drop packets with unknown TCP extensions of modifications. Unlike TCP and UDP payloads that directly contain application data, the decrypted QUIC payload consists in a sequence of QUIC *frames*.

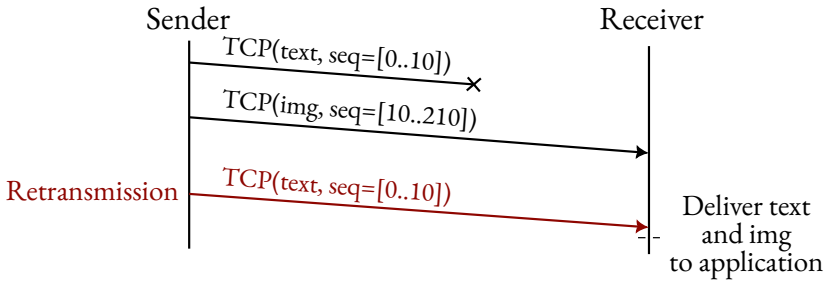


Figure 1.8: Head-of-line blocking on a TCP connection.

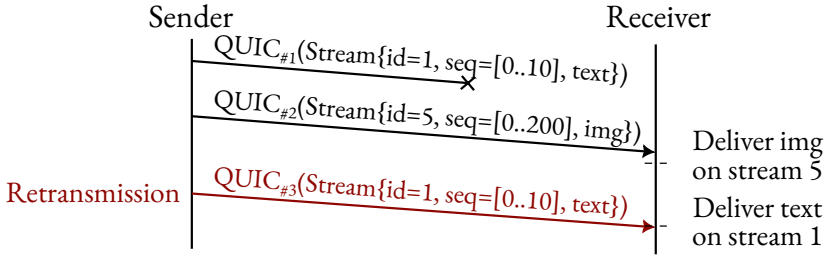


Figure 1.9: No head-of-line blocking with QUIC stream multiplexing.

1.4.3 Stream multiplexing: avoiding head-of-line blocking

The most important feature included in QUIC is *stream multiplexing*. As illustrated previously, QUIC allows applications to communicate using several reliable bidirectional streams concurrently, whereas TCP only offers a single stream per connection. While this has no effect on the amount of data that can be exchanged over the connection, stream multiplexing addresses the problem of *head-of-line blocking* occurring upon packet losses when several independent objects are sent over the same connection. Figure 1.8 shows an example of head-of-line blocking with TCP when sending a text and an image subsequently over the same connection. As TCP guarantees an in-order reliable delivery of the bytestream, any packet loss concerning the text will prevent the subsequent image from being delivered to the application, even if all the bytes of the image have been successfully received. Delivering the image bytes before the text would break TCP's reliability guarantees. The only way to process the text and image independently using TCP is to use separate TCP connections. Using QUIC, independent data can be transmitted through separate streams over a single connection as shown in Figure 1.9. In this example, the text is transmitted over stream 1 and the image over stream 5. Each stream is guaranteed to be delivered reliably and in-order, so the lost text data will be retransmitted in a new packet (#3). However, packet losses concerning the text do not prevent the image from being delivered as both objects are sent independently on separate streams.

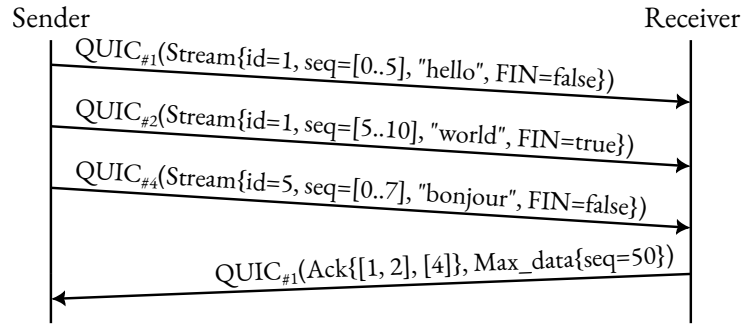


Figure 1.10: Two QUIC endpoints exchanging packets and frames.

1.4.3.1 Frames: control information as part of the encrypted payload

Frames are structured units of control information exchanged by QUIC peers. They are carried inside QUIC packets and serve many purposes. They are encoded in a *type-value* fashion in the encrypted payload. Their format and purpose differ based on the frame type. Here is a list of QUIC frames that are of interest for this thesis.

- **STREAM frame.** The STREAM frames carry the application data sent over QUIC streams. In addition to the application payload, they encode several control fields, such as the stream byte offset of the data they carry (similarly to the TCP sequence number), the ID of the concerned QUIC stream and the FIN flag, indicating that the carried data are the last of the byte stream in that direction. Several STREAM frames can be carried in the same QUIC packet in order to transmit data of several streams in the same packet when needed. Alike TCP, application data transiting through streams are delivered in-order and reliably to the receiving application.
- **DATAGRAM frame.** Not part of the base QUIC specification, DATAGRAM frames are proposed as a QUIC extension [RFC9221]. They allow applications to send datagrams over a QUIC connection. Similarly to the API offered by UDP sockets, the datagrams are sent in a best-effort manner: they can be lost, delayed and re-ordered. Unlike UDP datagrams, the datagrams sent over a QUIC connection are subject to congestion control. When a connection is limited by the congestion control, the DATAGRAM frames will either be delayed or dropped by the QUIC implementation.
- **MAX_DATA and MAX_STREAM_DATA frames.** These frames indicate the maximum amount of bytes that can be sent either on a specific stream (MAX_STREAM_DATA) or over the whole connection

(MAX_DATA). This implements a flow control mechanism similar to TCP's receive window.

- **PADDING frame.** This frame can be added to QUIC packets to increase its size. PADDING frames do not require specific action when processed by the receiver.
- **CONNECTION_CLOSE frame.** This frame is sent by an endpoint to close the QUIC connection.
- **ACK frame.** The ACK frames are sent by a QUIC receiver to indicate the packets that were successfully received. Similarly to the TCP SACK option, QUIC ACK frames allow performing selective acknowledgements indicating precisely the packet numbers having been received over the network. Packets that contain other frames than ACK, PADDING or CONNECTION_CLOSE frames are considered to be *ack-eliciting*. Upon reception of an ack-eliciting packet, the receiver must acknowledge it by sending an ACK frame within a short delay.

QUIC frames form a natural way of extending the protocol: new protocol behaviours can be implemented by defining new frames.

Figure 1.10 illustrates the packets exchanged on an open connection with a sender transmitting "helloworld" over stream 1 and "bonjour" over stream 5. In this example, the sender deliberately sends this data using three STREAM frames, each one being carried by a different QUIC packet. By correctly setting the FIN flag, the sender indicates that it won't send any more bytes over stream 1 but may still send new data over stream 5. The sender chooses to never send a packet with packet number 3, which is a valid behaviour for QUIC endpoints. On its side, the QUIC receiver selectively acknowledges packets #1, #2 and #4 by sending an ACK frame. It also informs the sender that it currently won't accept more than 50 stream bytes over the whole connection. Having already sent 17 bytes, the sender can still send 33 stream bytes before being blocked by the QUIC flow control mechanism.

1.4.4 QUIC loss recovery

Alike TCP, QUIC offers a reliable bytestream service. It must ensure the correct delivery of the streams data in presence of packet losses. For the same reasons as TCP, it also implements a congestion control mechanism for which packet loss can be a crucial signal. A key difference between QUIC and TCP is that QUIC packet numbers are never reused during a connection. Lost QUIC packets are never retransmitted as is. The content that needs to be retransmitted is placed inside new frames contained in a new QUIC packet with a new packet number. This avoids the ambiguities present in

TCP acknowledgements in which it is impossible to distinguish the loss of a packet from the loss of its retransmission only relying on the sequence number [RFC7323; RFC9002].

1.4.4.1 Loss detection

Inspired by the TCP RACK-TLP mechanism [RFC8985], QUIC implements an *acknowledgement-based* and a *probe timeout* (PTO) loss detection mechanisms. Using the acknowledgement-based loss detection, a QUIC packet with packet number x is deemed as lost when *any* of the following conditions is met :

- A packet with number $y \geq x + kReorderingThreshold$[4] has been acknowledged.
- A more recent packet than x has already been acknowledged and x has been sent for more than $\frac{9}{8} * RTT$[5]

Once a packet is marked as lost by the loss detection mechanism, the content of its STREAM frames is retransmitted in a new QUIC packet to ensure the reliability guarantee of the QUIC streams.

The acknowledgement-based strategy can mark a packet as lost only when more recent packets have been acknowledged. This means that it cannot detect the loss of the last packets of the flight. The probe timeout mechanism is used to trigger the sending of acknowledgements by sending new packets. If the probe timeout (PTO) expires without new packets being acknowledged, the QUIC sender sends one or two new QUIC ack-eliciting packets containing new stream data if available. These packets will trigger the sending of new ACK frames from the peer, allowing to detect the potentially lost packets using the acknowledgement-based strategy. The firing of the PTO in itself does not mark any packet as lost. The PTO can be computed using the following formula:

$$PTO = smoothed\_rtt + 4 * rttvar + max\_ack\_delay \tag{1.1}$$

where $smooth\_rtt$ is an estimation of the connection round-trip time, $rttvar$ is the mean RTT variation and max_ack_delay is the maximum amount of time a receiver waits before sending an acknowledgement upon reception of an ack-eliciting packet.

With its acknowledgement-based and probe timeout strategies, the QUIC loss detection mechanism needs at least one RTT to mark lost packets and

[4]The recommended initial value for $kReorderingThreshold$ is 3 [RFC9002].

[5]This value is the recommended time threshold from [RFC9002] but can be changed. The original RACK-TLP recommendation for TCP was to wait for $\frac{5}{4} * RTT$ [RFC8985].

trigger retransmissions. Only relying on retransmissions after loss may be insufficient for latency-sensitive applications that cannot afford to wait for an additional RTT to recover from losses. For this reason, this thesis considers extending QUIC's loss recovery mechanism using *Forward Erasure Correction*.

1.4.5 A broad class of applications

While the first application built atop QUIC is HTTP/3 [RFC9114], several classical protocols originally running over UDP or TCP have also been updated to use QUIC, such as DNS [RFC9250], WebSocket [RFC9220] and RTP [PO18], with DNS being already successfully deployed on the Internet [Kos+22]. Recently, the DNS resolver on Android devices was updated to use QUIC as well [QUICDNS]. The new WebTransport protocol aims at providing the different QUIC services over HTTP to web applications running in the browser [FKV23]. The MOQT protocol uses WebTransport in its turn to convey live media traffic over QUIC in the browser [Cur+23]. While these different use-cases would previously have been using either TCP or UDP based on their requirements, thanks to its multi-stream and datagram capabilities, built-in end-to-end encryption and easy extensibility, the QUIC protocol is general enough to welcome different use-cases with contrasting needs.

1.5 Forward Erasure Correction

Relying uniquely on retransmissions alike TCP might be insufficient for QUIC to provide suitable performance for latency-sensitive applications due to the added delay. Over high latency network configurations, the retransmitted data might arrive too late at the receiver to be considered useful by the application. In the case of video-conferencing applications, a late video frame can cause a stalling of the video playback, alter the picture and deteriorate user experience. The Forward Erasure Correction (FEC) loss recovery mechanism aims at sending the redundant information necessary for the receiver to recover the lost network packets without the need to wait for retransmissions. The simplest form of Forward Erasure Correction is to duplicate every sent packet ("opportunistic retransmissions") to be able to recover from the loss of any of them. Duplicating every packet doubles the needed bandwidth for the transfer, increasing considerably the load on the network. Several techniques based on information coding theory have been developed and proposed to reduce the load while providing more robust loss recovery capabilities than pure packet duplication.

This section starts by introducing the coding theory concepts required to fully understand the techniques leveraged in this work. It then gives an overview of the different FEC techniques applied in this thesis.

S_1S_2	00	01	10	11
$C_1C_2C_3$	110	101	011	000

Table 1.1: Example erasure correcting code with two binary source symbols

1.5.1 Coding theory primer

An *erasure correction code* efficiently encodes a message adding redundant information in order to decode the original message even if parts of the coded message have been erased (lost) by the network. Table 1.1 shows an example of a simple erasure correcting code with an original message composed of two binary source symbols and a coded message composed of three binary coded symbols. Sending the coded message instead of the original message has the advantage that the original message can be recovered even when one of the coded symbols gets erased by the network as soon as the receiver knows which symbols were received and erased. Let us take an example where a host receives the message "1∅1" where ∅ represents a symbol erasure, meaning that the second symbol is never received. By looking at the code defined in Table 1.1 the host can determine that the only coded message that could have possibly been sent is the message 101, allowing it to decode the received message back to the original message (01). In this thesis, we focus on two kinds of erasure correction codes: block codes and fountain codes.

1.5.1.1 Block codes and Reed-Solomon

Block codes are a popular family of erasure correcting codes. A (n, k) block code takes as input a message composed of k *source symbols* $S_1, ..., S_k$ and generates a coded message composed of n *coded symbols* $C_1, ..., C_n$ ($k \leq n$) such that the original message can be recomputed from the coded message even if a subset of the coded symbols is missing. We define the *code rate* as $\frac{k}{n}$, the ratio between the amount of source and coded symbols. The lower the code rate, the more redundancy is added. The code in Table 1.1 seen previously is an example of a $(3, 2)$ block code. This erasure correcting code has the nice property of being a *Maximum Distance Separable* (MDS) code: the minimum Hamming distance between any pair of coded messages is equal to $n - k + 1$ $(= 2)$. This property implies that the original message can be decoded as soon as any k of the n coded symbols are received. The code of Table 1.1 is pre-computed for every possible message of two binary digits. The *Reed-Solomon* block erasure correcting code provides an algorithm to generate an arbitrary number n of coded symbols for any message composed of k source symbols [RS60].

The Reed-Solomon erasure correcting code is an MDS code based on the

mathematical property that for any polynomial of degree $k-1$ as follows :

$$P(x) = a_k x^{k-1} + ... + a_1,$$

the a_i coefficients can all be retrieved from any set $\mathcal{S}$ of k points like the following :

$$\mathcal{S} = \{(x_1, P(x_1)), ..., (x_k, P(x_k)) \mid x_1 \neq ... \neq x_k\}.$$

Based on that property, the Reed-Solomon error correcting code defines a polynomial for any message of length k by using its k source symbols as coefficients for the polynomial. For instance, to protect a message composed of k source symbols $m = (S_1, ..., S_k)$, a (n, k) Reed-Solomon encoder builds the following polynomial :

$$P(x) = S_k x^{k-1} + ... + S_1$$

and evaluates it for n arbitrary values v_i. The result is a coded message composed of n coded symbols $c = (C_1, ..., C_n) = (P(v_1), ..., P(v_n))$ that is sent to the decoder. The original source symbols can therefore be received as soon as k out of the n coded symbols are received by the decoder. The encoding procedure can be written as the following dot product implying a *Vandermonde* matrix :

$$\begin{bmatrix} 1 & v_1 & v_1^2 & \dots & v_1^{k-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & v_k & v_k^2 & \dots & v_k^{k-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & v_n & v_n^2 & \dots & v_n^{k-1} \end{bmatrix} \cdot \begin{bmatrix} S_1 \\ \vdots \\ S_k \end{bmatrix} = \begin{bmatrix} C_1 \\ \vdots \\ C_k \\ \vdots \\ C_n \end{bmatrix}, \tag{1.2}$$

which can be rewritten

$$V \cdot m = c. \tag{1.3}$$

This allows the encoding procedure to be implemented as a matrix product. For the decoding procedure, the decoder takes k received coded symbols and constructs a variant of Equation 1.2 keeping only the matrix rows related to the k received C_i's, resulting in a $k \times k$ Vandermonde matrix. The decoding procedure can then be implemented as a matrix inversion or linear system solving problem. This class of problems can be solved if and only if the matrix determinant is nonzero. For Vandermonde matrices, this is guaranteed to be the case when the v_i's are all different [Mil]. Choosing distinct v_i's therefore ensures the recoverability of any Reed-Solomon block. For practical and computational reasons, Reed-Solomon erasure correcting codes generally operate over $GF(2^q)$ Galois Fields [Mor12] and the v_i arbitrary values are

chosen to be $\{1, \alpha, \alpha^2, ..., \alpha^{n-1}\}$ with alpha being one of the *root* values of the underlying Galois Field.

One of the drawbacks of block codes such as Reed-Solomon is that source symbols cannot be added easily to the encoder without having an effect on the whole encoding process: in Equation 1.2, adding a new source symbol S_{k+1} affects the values of $C_1, .., C_k$. A second drawback is the number of coded symbols that can be generated by Reed-Solomon in practice. Efficient implementations generally encode the Vandermonde matrix entries using 8-bits values. As the v_i's must all be distinct, this limits n to 256.

1.5.1.2 Fountain codes and Random Linear Network Coding

The family of fountain codes remove these block codes limitations, being able to produce an arbitrarily large number of coded symbols from a set of source symbols. The coded symbols can also be generated on the fly, without any fixed code rate. This is why they are also called *rateless* codes. Being introduced with Tornado [Bye+98] and significantly improved with LT codes in the early 2002's [Lub02], fountain codes became popular with the introduction of Raptor codes [Sho06] providing a linear time decoding algorithm that was also practical for network applications (see Section 1.5.1.3).

Random Linear Network Coding (RLNC or RLC) [Ho+03] provides a natural way to generate coded symbols in a rateless manner. A coded symbol C protecting a set of k source symbols $S_1, ..., S_k$ is generated as their linear combination using randomly generated coefficients as follows :

$$C = c_1 S_1 + ... + c_k S_k,$$

with every c_i being selected randomly or using a pseudo-random number generator. New coded symbols can be created by generating new random coefficients. To be loss-resilient, the RLC encoder generates and transmits more than k symbols to ensure that at least k symbols are received on-time. Upon receiving the coded symbols $C_1, ..., C_k$, the decoding procedure can be implemented by solving the following system of linear equations with $S_1, ..., S_k$ as the system unknowns :

$$\begin{cases} c_{11}S_1 + c_{12}S_2 + \cdots + c_{1k}S_k & = C_1 \\ c_{21}S_1 + c_{22}S_2 + \cdots + c_{2k}S_k & = C_2 \\ & \vdots \\ c_{k1}S_1 + c_{k2}S_2 + \cdots + c_{kk}S_k & = C_k \end{cases}, \tag{1.4}$$

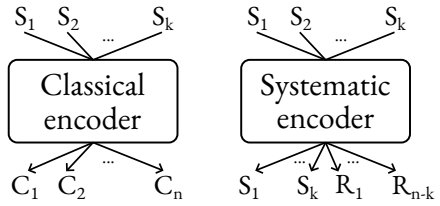


Figure 1.11: Classical encoder versus systematic encoder. $S_1, ..., S_k$ **can be processed directly upon reception and** $R_1, ..., R_{n-k}$ **can be ignored by the receiver if no loss occurred.**

which can be rewritten as the following dot product :

$$\begin{bmatrix} c_{11} & c_{12} & \dots & c_{1k} \\ c_{21} & c_{22} & \dots & c_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ c_{k1} & c_{k2} & \dots & c_{kk} \end{bmatrix} \cdot \begin{bmatrix} S_1 \\ S_2 \\ \vdots \\ S_k \end{bmatrix} = \begin{bmatrix} C_1 \\ C_2 \\ \vdots \\ C_k \end{bmatrix}. \tag{1.5}$$

The system can be solved if the matrix is *full rank*, i.e. it contains no pair of linearly dependent rows. The coefficients c_{ij} being drawn at random, the rows only have a small probability of being linearly dependent. In the case the matrix is not full rank, the decoder will have to receive new equations to decode the source symbols. The system solving can easily be implemented using Gaussian elimination.

1.5.1.3 Systematic codes

The techniques presented until now generate coded symbols that are sent in place of the original source symbols, the latter being retrieved once at least k coded symbols are successfully received. Having to wait for the reception of k coded symbols before being able to decode the source symbols is impractical for latency-sensitive applications needing to process the data as soon as possible. Fortunately, most erasure correction techniques can be implemented in a *systematic* way, in which the sent symbols can be separated into two parts, the first part being the k unmodified source symbols and the second part being the remaining *repair symbols*. Figure 1.11 compares the symbols sent on the network by a classical and a systematic encoder. Using the sytematic encoder, the source symbols $S_1, ..., S_k$ can be processed by the receiver as soon as they are received from the network. The repair symbols $R_1, ..., R_k$ are only used to recover the missing source symbols if losses occur in the network. If no loss occurs, the repair symbols can simply be ignored and no decoding operation takes place. In addition to being able to directly process the received source symbols, systematic codes drastically reduce the number of decoding

operations, making the FEC decoder drastically less CPU intensive when loss events are uncommon. A straightforward example of systematic code consists in generating a repair symbol as the result of the bytewise XOR over all the source symbols. The repair symbol can then be used to recompute a single missing source symbol by XORing the repair symbol with the $k - 1$ received source symbols.

The Reed-Solomon encoder can be rewritten in an equivalent systematic encoder. Concerning fountain codes, while LT codes are not systematic, Raptor codes can be implemented with a systematic encoder, explaining its popularity among fountain codes. Finally, RLC naturally provides a systematic encoder by transmitting the source symbols and sending the repair symbols as random linear combinations of the source symbols. The unknowns of the system to solve are only the lost source symbols. This thesis focuses on systematic erasure correcting codes for every proposed solution.

1.5.2 Protecting network packets

The erasure correcting codes we just described take as input a message m that is a set of k source symbols. In the example illustrated in Table 1.1, the source symbols are binary digits, making the erasure correcting codes operating at the bit level. Transport protocols commonly encounter *packet erasures* but rarely see *bit erasures* during the transfer. When congested, the network routers drop entire packets, not parts of them. When packets are partially corrupted by the network or physical medium imperfections, the checksums of UDP and TCP and the cryptographic authentication tag of QUIC make the whole packet to be considered as invalid and entirely ignored by the receiver. In this thesis we perform *packet-level* protection, meaning that the source symbols are entire packets payloads. This can be done without loss of generality or modification in any of the presented erasure correcting codes as packet payloads being a sequence of bytes, they can be considered as large numbers. To perform efficient mathematical operations on the packets, the erasure correcting codes are implemented using Galois Field operations in the $GF(2^8)$ finite field. In this field, the addition and subtraction between two packet payloads are both implemented as their bytewise XOR operation and the multiplication and division are based on pre-computed tables.

1.5.3 Mode of operation

FEC encoders can operate in *block* and in *convolutional* (or *sliding window*) modes. In block mode, the source symbols are grouped in separate and independent blocks with repair symbols being generated to protect each block. This behaviour is illustrated at the left of Figure 1.12. In this example, the

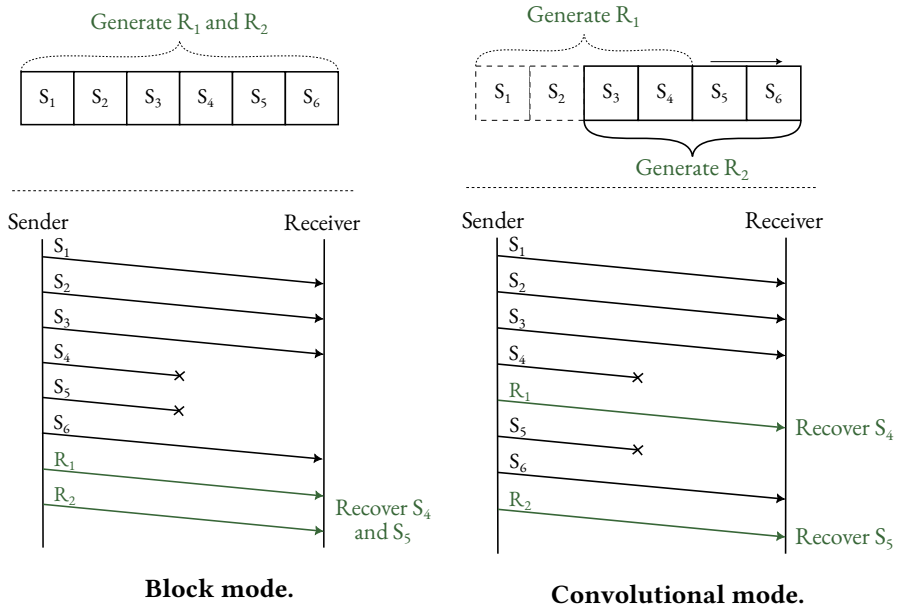


Figure 1.12: In block mode, the receiver has to wait for both R_1 and R_2 to recover S_4. In convolutional mode, S_4 can be recovered as soon as R_1 is received.

source symbols S_1, ..., S_6 form a block with the repair symbols R_1 and R_2 specifically protecting this block. After generating R_1 and R_2, new source symbols will be placed into a new separate block that will be protected by its own repair symbols. Block erasure correcting codes such as Reed-Solomon can only be used in block mode.

In convolutional mode, the sender maintains a window of protected source symbols and periodically generates repair symbols protecting the current window. The repair symbols of different windows can then be mixed together to recover the lost symbols. The right side of Figure 1.12 illustrates the convolutional mode of RLC. The repair symbols R_1 and R_2 are generated following the classical RLC procedure :

$$\begin{cases} c_{11}S_1 + c_{12}S_2 + c_{13}S_3 + c_{14}S_4 &= R_1 \\ c_{21}S_3 + c_{22}S_4 + c_{23}S_5 + c_{24}S_6 &= R_2 \end{cases} \tag{1.6}$$

with c_{ij} the randomly generated coefficients. In the example of Figure 1.12, only the symbols S_4 and S_5 are missing. The equations can thus be reduced like the following, forming a system using both symbols from different windows :

$$\begin{cases} c_{14}S_4 = R_1 - c_{11}S_1 - c_{12}S_2 - c_{13}S_3 \\ c_{22}S_4 + c_{23}S_5 = R_2 - c_{21}S_3 - c_{24}S_6 \end{cases} \tag{1.7}$$

with S_4 and S_5 the only two unknowns of the system. This capability of mixing repair symbols allows RLC to be used in convolutional mode. With the example of Figure 1.12, S_4 can be recovered sooner using convolutional mode than with block mode as only R_1 needs to be received to recover S_4.

Roca *et al.* compare block and convolutional codes using Reed-Solomon and Random Linear Codes (RLC) to represent both families of codes [Roc+17]. They show that while the Reed-Solomon block codes provide a higher encoding speed, RLC allows recovering the packets with a reduced latency compared to Reed-Solomon. In this thesis, we focus on Reed-Solomon for block mode and RLC for convolutional mode.

Mathematical conventions We define a (n, k) block code as a code sending a block of k source symbols followed by $n - k$ repair symbols. We define a (n, k, c) convolutional code as a convolutional code sending $n-k$ repair symbols every k source symbols. The repair symbols protect the c previous source symbols. The left side of Figure 1.12 shows an example of a $(6, 4)$ code. The right side of Figure 1.12 shows an example of a $(3, 2, 4)$ convolutional code.

1.5.4 FEC as a transport loss recovery mechanism

General-purpose reliable transport protocols such as TCP and QUIC essentially rely on SR-ARQ to recover from loss events. While FEC has already been tuned for specific link-layer technologies [Bie93; Kat+08; Gie+18], many existing works have considered using FEC for TCP [Hui97; LK04; Sun+11; BLK04; Fer+18; Cui+14; Clo+13]. However, these solutions were either deployed as a tunnel below TCP or implemented in a simulated environment. Indeed, using FEC as the loss recovery mechanism for TCP faces several obstacles. First, TCP is generally implemented in the kernel of operating systems, making it hard to deploy complex and resource consuming systems such as FEC as part of the OS. Second, the extensibility of TCP is limited due to both its small option space and ossification problem [Rai+12]. QUIC does not suffer from any of these problems. First, almost all its control information is encrypted, making it easy to deploy new protocol features without middlebox interference. Second, the design of QUIC allows to easily implement new protocol behaviour through the use of QUIC frames that can span entire QUIC packets, whereas TCP options are limited to 40 bytes.

This is why FEC was originally considered as part of the QUIC protocol in early prototypes [Ros12]. It has however rapidly been dropped due to negative experimental results [Lan+17; Swe17]. The experiments were using a simple XOR-based erasure correction code, making it only possible for QUIC to recover from single packet losses, while the authors encountered more than 70% of loss events implying two or more packets. Finally, the only evaluated

use-cases were web search and video-on-demand. In this thesis, we start from these early conclusion and entirely revisit the use of FEC as part of the QUIC loss recovery mechanism. We implement, integrate and compare powerful erasure correcting codes inside the QUIC loss recovery mechanism in Chapter 2. Based on the findings of this chapter, we provide a short discussion on the interaction of FEC and congestion control in Chapter 3. We imagine new ways to extend a transport protocol in Chapter 4 and use it to adapt the loss recovery behaviour to the application and network conditions in Chapter 5. We study the characteristics of a lossy wireless network in Chapter 6 to evaluate to which extend FEC can be beneficial in real networks. We finally apply our techniques on popular network applications on this network and show that we can obtain actual performance improvements for real applications on real networks in Chapter 7.

QUIC-FEC: A general loss recovery QUIC extension

2

In this chapter, we design, implement and evaluate a first FEC extension to QUIC. This extension is mainly intended for high-delays and lossy communications such as In-Flight Communications services where losses are frequent and retransmissions impact user experience [Rul+18].

We propose three main contributions in this chapter. First, a modular QUIC extension that enables the use of a variety of FEC techniques. Our extension already goes beyond the experiments carried out by Google with a simple FEC technique in Chrome [Lan+17; Swe17]. Furthermore, our design makes the congestion control aware of packets that were either normally received or recovered by FEC. Second, we provide a complete implementation of the proposed extension in `quic-go` [al22] with three different FEC techniques. Third, our evaluation, over a wide range of parameters, indicates that the proposed FEC techniques improve the performance of QUIC for short file transfers.

This chapter is organised as follows. We first discuss the support of FEC within QUIC in Section 2.1. We then define in Section 2.2 the design and implementation details of QUIC-FEC, our extension enabling the use of FEC-protected transfers with QUIC. We finally assess its performance and compare different erasure correction codes through experiments using a wide range of network and loss configurations in Section 2.3.

The content of this chapter has been published in the article *QUIC-FEC: Bringing the benefits of Forward Erasure Correction to QUIC* [MDB19] and presented at the IFIP Networking 2019 conference.

2.1 Forward Erasure Correction for long-delay communications

FEC is especially interesting compared to retransmission mechanisms when the delay and loss rate are high. Rula *et al.* [Rul+18] revealed that In-Flight Communications (IFC) are highly deteriorated by the important latency and loss rate, making it an interesting candidate for evaluating the benefits of FEC. IFC technologies rely on cellular and satellite technologies. Despite built-in redundancy and retransmission mechanisms often proposed by such

technologies, they may not be able to recover from packet losses, especially when the user is mobile [Kuh+18], which explains why losses can be perceived from higher layer perspectives in the IFC use-case. Packets can also be dropped due to congestion and routers doing Active Queue Management (AQM).

In their work on IFC communications, Rula *et al.* [Rul+18] study the potential impact of the new technologies for IFC aiming at improving the link's bandwidth. They conclude that improving the link bandwidth does not improve significantly the performance as the bottleneck resides in the high losses and latencies. They also recognise that reducing the latency and loss rate in this use-case is challenging. We thus focus on the IFC scenario for our first FEC extension as it presents a first ideal setup for using Forward Erasure Correction where the benefits of the approach are straightforward and intuitive.

In this chapter, we use the word *FEC scheme* to refer to the system taking care of both the erasure correcting code and the signalling that is required to encode and decode source and repair symbols using that particular erasure correcting code.

2.2 Integrating FEC into QUIC

In this section, we propose a generic Forward Erasure Correction extension for QUIC and implement it using the `quic-go` [al22] implementation. The application can select the FEC scheme that suits its needs and recover from losses without waiting for retransmissions.

Adding such a mechanism requires addressing several points. We first describe how we advertise which are the source and repair symbols to the peer in Section 2.2.1. We then explain how we manage to transparently handle different FEC schemes in Section 2.2.2. We finally discuss the impact of FEC on congestion control in Section 2.2.3.

2.2.1 Defining and exchanging the source and repair symbols

This first design considers QUIC packet payloads as source symbols. We use a previously reserved flag of the QUIC header to inform the peer that the packet must be considered as a source symbol and call it the *FEC* flag. We add a 32-bits field to the packet header when the FEC flag is set to transmit FEC scheme-specific values to the peer[1]. We only protect packets containing STREAM frames carrying user data. Successive ACK frames contain redundant information by design, reducing the impact of their losses compared to STREAM frames.

[1]The use of the FEC flag and this added field will be reconsidered in the next design iterations of this thesis, relying uniquely on QUIC frames instead.

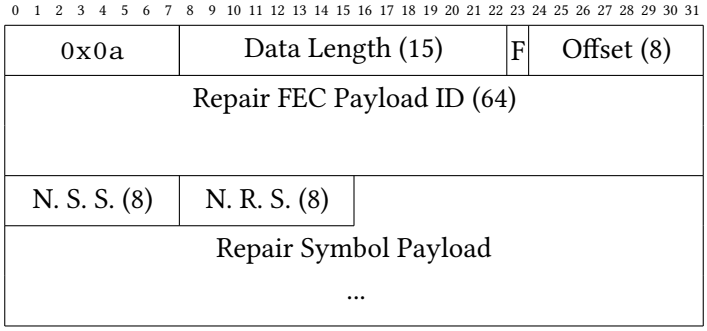


Figure 2.1: Wire format of a REPAIR frame. The Repair FEC Payload ID field is opaque to the protocol and is populated by the underlying FEC scheme.

We introduce a new QUIC frame, the *REPAIR frame*, depicted in Figure 2.1. The REPAIR frame contains the repair symbols payload as well as the *Repair FEC Payload ID* 64-bits field containing FEC scheme-specific values. These FEC scheme-specific values are handled by the underlying FEC scheme and are opaque to the QUIC protocol. It allows easily developing new FEC schemes independently of the behaviour and core functionalities of the QUIC protocol. The *REPAIR frame* also advertises the number of source (*N.S.S.*) and repair symbols (*N.R.S.*) in the current FEC Block for block codes. For convolutional codes, these fields advertise the number of source symbols in the sliding window and the number of repair symbols generated at each window step. They allow the sender to dynamically change the code rate during the connection and adapt its use of FEC to the changing network conditions. As a single repair symbol could be too large to fit into a single REPAIR frame, the latter contains an *Offset* field indicating the offset of the repair symbol chunk transported in the frame and a FIN bit (*F*) indicating if the frame contains the last chunk for this repair symbol.

While QUIC packets are sent encrypted and authenticated, the repair symbols are generated from their cleartext payload, avoiding the CPU overhead of deciphering the recovered symbols. The confidentiality and integrity properties of the recovered symbols are still ensured since the REPAIR frame is a classical QUIC frame sent inside the QUIC encrypted payload. No REPAIR frame is sent before the end of the TLS handshake and 0-RTT QUIC packets do not contain REPAIR frames.

2.2.2 The FEC Framework

The IETF has already developed solutions to add erasure-correcting codes to several protocols. The most recent solution is the FECFRAME framework [RFC6363] which has notably been applied to RTP and supports different

FEC schemes [RFC6865; RFC6816]. Inspired by FECFRAME, we define a FEC Framework implementing the common behaviour of these FEC schemes in order to further simplify their implementation. It provides a structure for the different FEC scheme-specific values exchanged by the peers to proceed to erasure correction. Its design is described in details in a technical report [MDB18]. Our FEC Framework is designed to handle both block and convolutional codes. For block codes, it uses the 32-bits field added in the packet header to encode the FEC block number and the offset of this packet in the FEC block. For convolutional FEC schemes, it uses these 32 bits to encode the offset of this packet in the protected packets sequence. The framework also leverages 32 bits of the *Repair FEC Payload ID* field in the REPAIR frame with informations identifying the FEC Block (for block codes) and coding window (for convolutional codes) protected by the repair symbol. The 32 other bits can be populated by the underlying FEC scheme with values required to perform the encoding/decoding.

At the time of writing the article presented in this chapter, such an interfacing for FEC in QUIC also brought interest from the IETF, where the network coding research group worked on an Internet Draft [SMR19] before the publication of this work. While the core ideas were similar, the draft was recommending to only protect stream chunks while our design protects arbitrary QUIC frames of any type. Upon publicating this work, we were invited to take part to the IETF design of the FEC extension of QUIC and integrated the design proposed in this chapter in two new versions of the Internet Draft [Swe+20a; Swe+20b].

2.2.2.1 Studied FEC schemes.

Our implementation supports three different FEC schemes, each of them having different characteristics: the XOR, Reed-Solomon and Convolutional Random Linear Code (RLC) FEC schemes. The first two use block codes and the last one uses RLC as a convolutional code.

- *XOR FEC scheme.* Its principle is quite simple: the source symbols are simply XORed with each other to generate a repair symbol. It is easy to implement and to compute but can only recover the loss of one source symbol. Experiments carried out by Google showed that this is insufficient on the Internet because losses often occur in bursts [Lan+17; Swe17]. Our implementation uses interleaving to recover from burst losses with the XOR FEC scheme. Sending successive packets in different FEC Blocks enables the XOR FEC schemes to better handle burst losses at the expense of delay.

- *Reed-Solomon FEC scheme.* It can generate multiple repair symbols per

source block, allowing handling of burst losses. While it better handles burst losses than the XOR FEC scheme, it is also more computationally intensive.

- *Convolutional RLC FEC scheme.* As convolutional codes provide different properties from block codes, our FEC extension enables their use through the RLC error correcting code. Our implementation is inspired from the *FECFRAME* RLC FEC scheme design [RFC8681].

2.2.3 FEC and the congestion control

The QUIC protocol is intended to use classical congestion control algorithms, including the loss-based algorithms such as New Reno and Cubic. In the case of QUIC-FEC, there are three possible scenarios impacting the loss recovery mechanism and congestion control when it experiences a packet loss. *i*) *The packet was not FEC-protected.* In that case, it will not be recovered. The sender observes a hole in the acknowledgements and registers a loss. Packets containing only REPAIR frames fall in this category. *ii*) *The packet was FEC-protected but could not be recovered.* In that case, the sender will notice the loss and retransmit the missing STREAM frames. *iii*) *The packet was FEC-protected and recovered.* Acknowledging these recovered packets can hide the congestion signal and make the FEC-enabled protocols behave unfairly compared to TCP or regular QUIC, as they could potentially take more than their fair share of the link bandwidth. In this first FEC extension, we considered three ways of avoiding to hide the congestion notification signal due to packet losses:

1. *Not acknowledging the recovered packets.* This approach conservatively considers a recovered packet as lost. It leads to a similar congestion control behaviour to when the lost packets are not recovered with FEC. The drawback is that the sender will perform unnecessary packet retransmissions.

2. *Acknowledging part of the recovered packets depending on the origin of their loss.* Kim *et al.* [Kim+14] propose to modify the loss-based congestion controls for this purpose. They assume that a congestion-implied loss event is preceded by an increase of the Round-Trip-Time (RTT) due to the filling of the network buffers. They propose to diminish the decrease of the congestion window after a packet loss if the current RTT is close to the minimum observed RTT. Tickoo *et al.* [Tic+05] propose to only react to congestion when it is explicitly notified by the network nodes though the Explicit Congestion Notification (ECN) [RFB01] mechanism. TCP Westwood [Mas+01] estimates the link bandwidth using the acknowledged data rate. At each loss event, it adjusts its congestion window to use the estimated bandwidth instead of multiplicatively

decreasing it. In this thesis, we want a loss recovery mechanism that works with every common congestion control algorithm, including the ones that consider that every packet loss is due to congestion.

3. *Explicitly advertising a packet recovery to the sender.* This is the approach we follow in this chapter. The recovered packet is acknowledged using a regular QUIC ACK frame but the receiver also signals that this packet has been recovered with an additional frame. Upon reception of this information, the sender both adapts its congestion window according to the loss event and removes the content of the recovered packet from its retransmission queue. We implement this solution in QUIC-FEC with a *RECOVERED* frame. Its format is similar to the QUIC ACK frame: it advertises the ranges of newly recovered packets.

Once a sender receives a RECOVERED frame, it removes the recovered packets from its retransmission queue. It then signals to its congestion control that a packet has been lost for each packet listed in the RECOVERED frame. Using the RECOVERED frame conservatively adapts the congestion window of the sender as if every loss was caused by congestion. This behaviour is comparable to the utilization of ECN. Once a packet containing a RECOVERED frame is acknowledged, the recovered packets ranges are removed from the subsequent RECOVERED frames. We analyse the impact of this approach in Section 2.3.2.4.

This technique still sends ambiguous signals to the receiver by both acknowledging a packet and signalling its reception through FEC. Such a dual signal can create confusion on some loss-based congestion control algorithms. The approach is refined in the next chapters to avoid this ambiguity.

2.3 Evaluation

In this section we perform a large set of experiments to assess the performance of our implementation and analyse the benefits of this first FEC extension by porting the HTTP use-case over QUIC-FEC. We first describe our methodology, then perform experiments with parameters inspired by In-Flight Communications. We analyse the Download Completion Time (DCT), i.e. the time required to complete an HTTP transfer.

2.3.1 Methodology

We use network emulation with the Mininet tool [Han+12] to evaluate the performance of the different FEC schemes in `quic-go`. We perform experiments with different loss models. While small burst lengths or uniform losses can already provide an idea on the efficiency of a solution, we advocate for

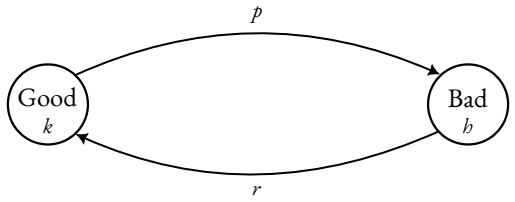


Figure 2.2: Gilbert-Elliott 2-states Markov Model

looking at longer burst lengths as well in order to evaluate our solution with different loss configurations and handle the problematic cases encountered by Google in 2017 [Lan+17]. We thus use the Gilbert-Elliott model [Ell63], a standard loss model used to represent bursts of lost packets.

The *Gilbert-Elliott* model is a two-states Markov model used to represent correlated losses. It is illustrated in Figure 2.2. The two states are the *Good* and *Bad* states. In the *Good* state, a packet is delivered with a probability k. In the *Bad* state, a packet is delivered with a probability h. p denotes the probability of transition from the *Good* to the *Bad* state, while r denotes the probability of transition from the *Bad* to the *Good* state. While this model stays fairly simple, it allows representing more complex and realistic loss patterns than a classical uniform model [HH08].

2.3.1.1 Experimental design

We use an *experimental design* approach to perform our experiments [Fis49]. This methodology consists in defining wide ranges for the values of each parameter. Numerous experiments are then performed with parameters values sampled randomly from these wide ranges. This provides a global overview of the entire parameters space. In addition to providing a general confidence concerning the performance of the tested implementation, it mitigates the bias in the parameters selection by the experimenter. We use the WSP algorithm [SCS12] to broadly sample the space of parameters with a reasonable number of experiments. Unless otherwise specified, we run the experiments with 130 different combinations of parameters. Each configuration is run 9 times and the median download completion time from these 9 runs is considered. The experiment consists in an HTTP GET request for a particular file using QUIC. Figure 2.3 shows the network topology used for our experiments. We apply the delay, losses and bandwidth limitation on the link between the two routers. Table 2.1 shows the parameters ranges chosen for our experiments. It specifies ranges for the One-Way Delay (OWD), bandwidth (BW), uniform loss rate (p) when a uniform loss model is used and the state-transition probabilities (p, r, k and h) when a Gilbert-Elliott loss model is used. The parameters values are inspired from the work of Rula *et al.* on In-Flight Communications [Rul+18].

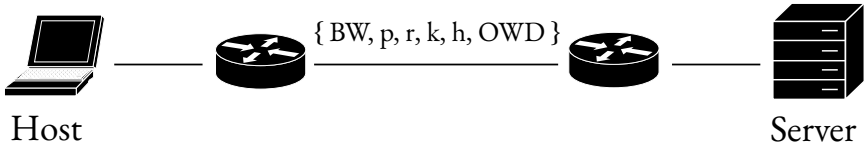


Figure 2.3: Network topology for our experiments.

Parameter	BW (Mbps)	p	r	k	h	OWD (ms)
Smallest	0.3	0.01	0.08	0.98	0	100
Highest	10	0.08	0.5	1	0.1	400

Table 2.1: Experimental design parameter ranges.

As the authors specified one representative set of parameters for *Direct Air-to-Ground Communication* (DA2GC) and one set for *Mobile Satellite Service* (MSS), we built our ranges around these values and perform experiments with many combinations of different values within these ranges, covering most of the loss conditions they experienced. For each set of parameters, the experiment uses 4 file sizes: 1kB, 10kB, 50kB and a larger file of 1MB. These sizes are intended to represent typical file sizes for the web browsing use-case.

As this is a first attempt of extending the QUIC loss recovery mechanism with FEC, the experiments are applied with a fixed code rate. Unless otherwise specified, the level of redundancy is set to $(30, 20)$ for the Reed-Solomon code and (3, 2, 20) for the RLC FEC Scheme. This ensures a code rate of $\frac{2}{3}$ and a burst recovery capability of 10 symbols per block. Adapting the code rate to the network characteristics as well as the application traffic is explored in the next chapters of this thesis.

2.3.1.2 Reproducible experiments

We strongly advocate for having reproducible experiments in order to easily analyse the results. It also enables a fair comparison between the two solutions and assess them under equal conditions since the number of lost packets as well as their exact position in the transfer have a significant impact on the download completion time. Mininet already provides tools to emulate uniform losses on a link but neither provides a Gilbert-Elliott loss model nor a way to deterministically reproduce the loss pattern of an experiment. We thus built our own tool, `ebpf_dropper` [Mic23a], allowing emulating losses in a network in a deterministic and reproducible manner. This tool, written using the *extended Berkeley Packet Filter* (eBPF) [Fle17a], can be attached to a network node via the `tc` tool. It provides a uniform and a Gilbert-Elliott deterministic loss model, which can be given a seed to exactly reproduce the sequence of lost packets for all the solutions studied here. In this thesis, all the

experiments performed using emulations or simulations leverage reproducible and deterministic loss patterns.

2.3.2 Results with uniform losses

As Rula *et al.* [Rul+18] proposed a uniform loss rate for the IFC use-case, we first perform experiments with uniform losses and investigate the benefits of FEC in QUIC in these configurations. We first study the two average cases for IFC. We then extend the parameters ranges using experimental design. We compare the regular QUIC with our QUIC-FEC implementation, using different error-correcting codes: the RLC and Reed-Solomon codes. We do not present the results for the XOR code, as it showed similar or poorer results to these two codes.

2.3.2.1 Specific IFC use-cases

In this section, we study in details the average parameters values proposed by Rula *et al.* [Rul+18] for Mobile Satellite Service and Direct Air-To-Ground Communications with different deterministic uniform loss patterns. For each case, we performed 50 experiments for each file size with a different seed for our deterministic loss generator, allowing experimenting with a high variety of loss patterns. We do not explore here the results with the Reed-Solomon code as RLC outperforms it in this uniform losses environment.

2.3.2.1.1 Direct Air-To-Ground Communication (DA2GC) We experiment with the average parameters values for the DA2GC scenario where the classical cellular network is used when the cellular antennas are in reach of the airplane [Rul+18]. This leads us to an average Round-Trip-Time of 262ms, a link bandwidth of 0.468 Mbps and a loss rate of 3.3%. The results are presented on the left graph of Figure 2.4. As we can see, even within the same set of parameters, the experiment can lead to quite different results when run with different seeds governing the loss patterns. With small file transfers, only a few packets are lost, if any.

As we can see, the 1kB file download sees only a positive impact or no impact when FEC is used. The file can be contained in a single STREAM frame and is not large enough to saturate the sender's congestion window even when FEC is used, so the redundancy overhead of FEC does impact negatively the transfer. However, when the packet containing the STREAM frame is lost, QUIC-FEC recovers it as soon as the REPAIR frame is received. The regular QUIC implementation will have to wait for at least one RTT to retransmit the lost frame. In the case of a 1kB file download, the loss of the only STREAM frame is a tail loss, that will be retransmitted once the Probe Timeout

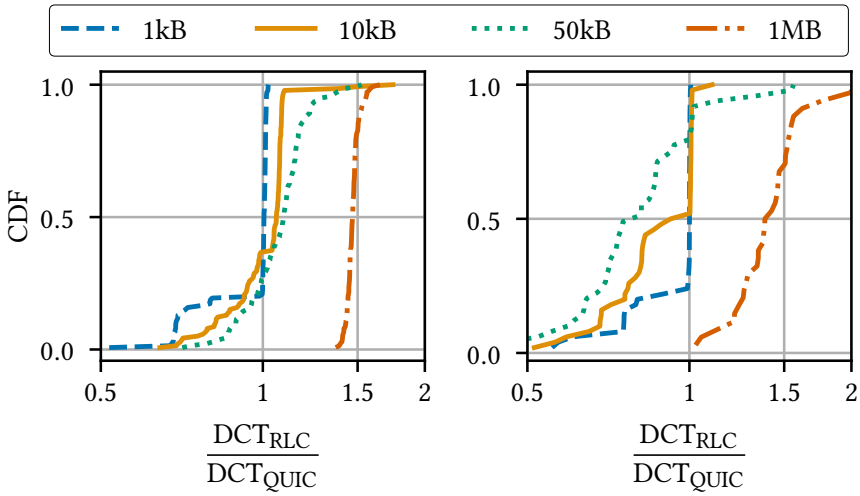


Figure 2.4: Download Completion Time (DCT) ratio between QUIC-FEC with the RLC block code and QUIC for the average DA2GC (left) and MSS (right) parameters values. A ratio below 1 means that QUIC-FEC performed better than QUIC. For DA2GC, FEC was only beneficial for 25% of the experiments for small file sizes. FEC noticeably deteriorates the performance in the other cases, due to the low bandwidth. For MSS, FEC can improve the performance in 75% of the cases for 50kB file transfers and does not deteriorate it when no loss occurs during small file transfers, due to the higher available bandwidth.

(PTO) mechanism is triggered, which takes more time than the standard acknowledgement-based loss detection mechanism of QUIC [RFC9002].

The advantages of FEC are less evident for larger files: for both 10kB and 50kB files, using FEC can deteriorate the download completion time due to the added redundancy. This result can easily be explained. With such a low bandwidth, the network forwards one packet of 1200 bytes every 20.5 milliseconds. Even in the case of a limited number of packets such as with the transfer of a 10kB file, the impact of the additional network capacity required to transfer the repair symbols will be noticed with such a limited bandwidth. This negative impact is even more visible on the 1MB curve, with an download completion time increase of 50%, directly related to the redundancy overhead sent by QUIC-FEC (recall that one repair symbol is sent every two source symbols). The advantage of FEC is especially visible when a packet loss occurs during our 1kB, 10kB and 50kB downloads: recovering the lost symbol through FEC reduces the DCT compared to a retransmission. These first results clearly show the potential of using FEC in QUIC as well as the drawbacks of the approach when redundancy is sent without care. This illustrates the complexity of the FEC mechanism and sets up the stage for the rest of this thesis. The end goal is to obtain significant latency improvements while avoiding such a negative impact due to redundancy overhead.

2.3.2.1.2 Mobile Satellite Service (MSS) We also experiment with the average parameters values for the Mobile Satellite Service scenario. In this scenario, network access is provided in the plane using a satellite connection. This is the most common scenario in airplanes and has the advantage of being available in the middle of the ocean, despite an increased delay compared to DA2GC. We set a Round-Trip-Time of 761ms, a link bandwidth of 1.89 Mbps and a loss rate of 6% following the statistics collected by Rula *et al.* [Rul+18]. The graph at the right of Figure 2.4 shows the DCT ratio between QUIC-FEC with RLC and QUIC without the FEC extension. As we can see, using FEC reduces the total DCT in the vast majority of the smaller files downloads. The loss rate is sufficiently large to have a highly negative impact on the DCT that will be withdrawn through the use of FEC. As the bandwidth is significantly higher than for the DA2GC case, the negative impact of FEC on smaller files is less present. It can be easily seen when comparing the 10kB curves for the DA2GC and MSS cases: when no loss occurred, the ratios are significantly closer to 1 for the MSS case. Finally, we can note a higher variance of the DCT ratio for the 1MB files compared to the DA2GC scenario. This is due to the higher available bandwidth and RTT: the congestion window can take larger values before encountering the first packet loss making the sender exit the slow start phase. The position of the first loss in the loss pattern has thus a higher impact than with a smaller available bandwidth and smaller RTT.

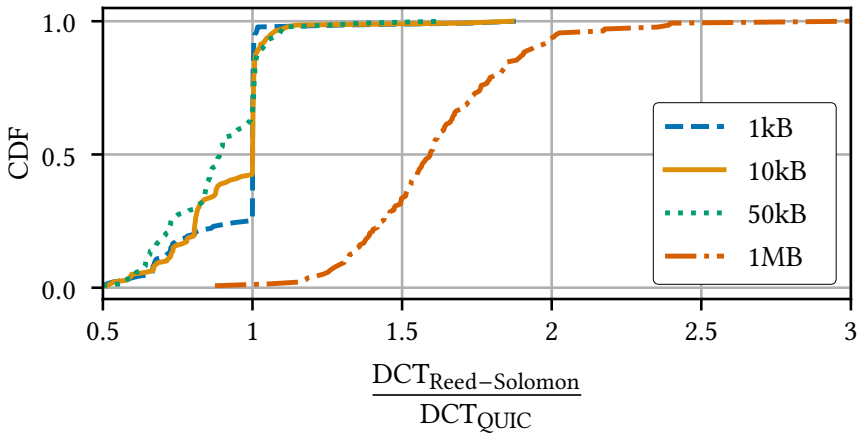


Figure 2.5: DCT ratio between QUIC-FEC with the Reed-Solomon block code and the regular QUIC.

2.3.2.2 Experimental design

We now use the experimental design approach to explore a broader set of parameters values. Figure 2.5 shows the CDF of the DCT ratio between QUIC-FEC using Reed-Solomon and the regular QUIC, with four different file sizes. Each experiment in this CDF has been performed with parameters selected from the ranges shown in Table 2.1.

2.3.2.2.1 Large files transfers We can easily see that QUIC-FEC performs badly compared to the regular QUIC with the 1MB file in nearly every scenario, showing the limits of FEC when not used adaptively. With the $\frac{2}{3}$ code rate applied in this experiment, QUIC-FEC has to transfer 1.5 times the amount of bytes transferred by regular QUIC (ignoring the potential retransmissions), which increases the overall download completion time. With longer files, the benefits brought by FEC are thus masked by the overhead needed to transmit the redundancy.

2.3.2.2.2 Small files transfers The advantage of FEC is more visible on small file transfers. Recovering a packet with FEC avoids the wait for a retransmission. A retransmission costs at least one additional round-trip time. With small files, the wait for this additional round-trip has a higher relative impact on the overall DCT. The experimental design also shows that there is a benefit in protecting the client request. For instance, our 108th test discards the client packet containing the GET request. The server then recovers it without the need for the client to retransmit it and can directly begin to serve the

request. It should however be noted that the approach becomes beneficial only when the client's request is composed of more than one packet. Otherwise, protecting the client's request is equivalent to simply duplicating it.

When looking more closely at our results, we can also see that FEC can still be harmful, depending on the loss pattern and the available bandwidth. Indeed, for experiments during which no loss occurred and with a low available bandwidth, using FEC sensibly increases the DCT, even for the 10kB and 50kB files transfers. These configurations are indeed similar to the DA2GC scenario. The additional time needed to transmit the redundancy is non negligible compared to the overall DCT. This overhead is greatly reduced for experiments with a higher available bandwidth.

2.3.2.2.3 Comparing FEC codes We now compare the impact of the FEC Scheme used for these different file sizes. Figure 2.6 shows the DCT ratio between QUIC-FEC using the RLC convolutional code and QUIC-FEC using the Reed-Solomon block code. As we can see, RLC performs significantly better than Reed-Solomon with the 1MB file transfer. This can be explained easily by the way these two codes send their source and repair symbols. These two codes provide the same code rate and a similar packet recovery capability. However, the RLC code interleaves the packets containing the REPAIR frames with the FEC-protected QUIC packets. On the other hand, the Reed-Solomon code sends all its REPAIR frames after the block containing the FEC-protected packets has been sent. In our experiments, the RLC code sends one REPAIR frame every two FEC-protected packets. The Reed-Solomon code sends 10 REPAIR frames every 20 FEC-protected packets. If the first FEC-protected packet of a Reed-Solomon block is lost, the receiver will have to wait for receiving the 19 other FEC-protected QUIC packets and one repair symbol before being able to recover it. On the other hand, the RLC code must only wait for the reception of two additional symbols: the following QUIC packet and the following REPAIR frame. With a packet-based loss detection threshold of 3 packets such as the one defined in the QUIC loss recovery document [RFC9002], using the Reed-Solomon code will trigger a spurious retransmission on the sender in most cases as the packet has been recovered too late. The spuriously retransmitted packet will occupy the congestion window of the sender, while a new packet will be sent when RLC is used.

2.3.2.3 Exploring the impact of redundancy overhead

During the previous experiments, we saw that FEC performed badly when no loss occurs and when the available bandwidth is low. We now study to which extent using FEC in the QUIC loss recovery mechanism can deteriorate the DCT. We perform experiments with the same parameters, except for the loss

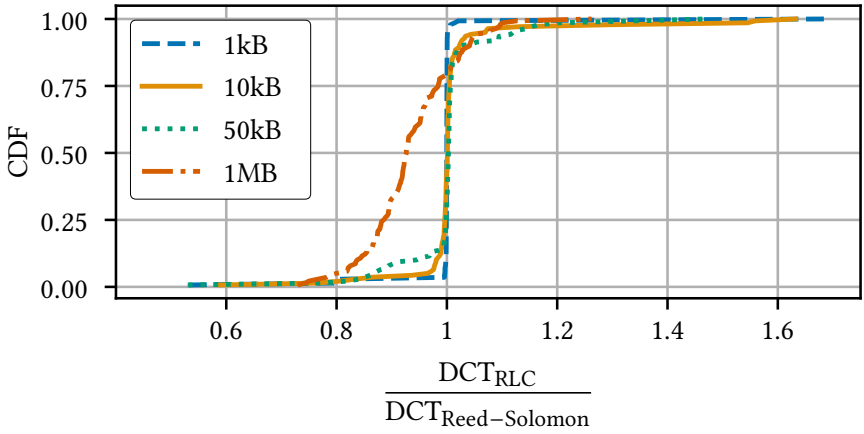


Figure 2.6: DCT ratio between QUIC-FEC with the convolutional RLC code and QUIC-FEC with the Reed-Solomon block code.

rate that we set to 0%. Figure 2.7 shows the DCT ratio comparing QUIC-FEC with RLC and regular QUIC. On the left (resp. right) of the figure, QUIC-FEC uses RLC with a code rate of $\frac{2}{3}$ (resp. $\frac{4}{5}$). As we can see, except for some results due to a slight variance in our experiments, using FEC always deteriorates the DCT for large transfers. When looking more closely at our results, we saw that the DCT for small downloads is mostly deteriorated when the available bandwidth is low. Unsurprisingly, we observe that increasing the code rate reduces the negative impact of FEC on the DCT. These experiments show that for large bulk transfers where performance is directly related to throughput, the classical SR-ARQ mechanism of QUIC is the most efficient way of delivering the data. This shows the need for adjusting the redundancy level during the lifetime of a connection.

Adaptive coding schemes are already studied in the literature at different levels in order to reduce the negative impact of over-coded transmissions [NTM08; Hou+08; ZZZ04; CP07; Zha+05; CLM15]. It is however known that ARQ offers better performance than FEC for larger bulk transfers [Zha+05]. This is also what we can observe from our experiment. The general QUIC loss recovery mechanism we aim for in this thesis should therefore limit the use of FEC and mostly rely on SR-ARQ for throughput-driven use-cases. Note however that SR-ARQ requires large buffers to store the received packets. While QUIC receive buffers can generally be large (up to several megabytes), there are cases where they remain a factor limiting the performance of SR-ARQ. We explore in Chapters 5 and 7 how FEC can help for these buffer-limited scenarios.

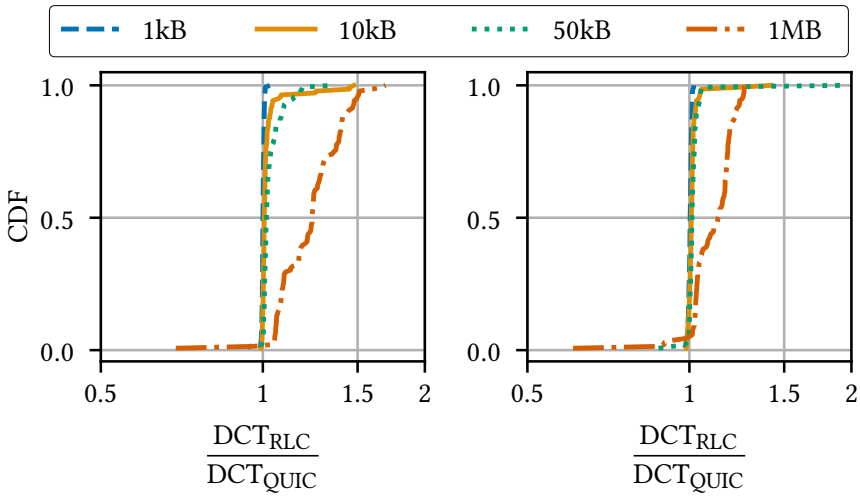


Figure 2.7: DCT ratio between QUIC-FEC with RLC and the regular QUIC with a loss rate of 0%. On the left, the used code rate is $\frac{2}{3}$, while on the right, the code rate is $\frac{4}{5}$. Increasing the code rate reduces the impact of FEC on the bandwidth.

2.3.2.4 The importance of recovery notification

In Section 2.2.3 we proposed the RECOVERED frame to notify the peer that packets have been recovered to avoid hiding the loss-based congestion signal. In this section, we analyse the impact of this notification on the fairness of QUIC-FEC.

The connection parameters used for this experiment are the parameters of the MSS scenario, except that we set a loss rate of 0% to avoid disturbing the experiment with random losses. Losses will only be caused by congestion. We use a $(7, 6, 20)$ RLC code, leading to a code rate of $\frac{6}{7}$, to better see the impact of masking the congestion signal when no RECOVERED frame is sent. Indeed, with a higher code rate such as $\frac{2}{3}$, a large part of the packet flow is composed of packets containing only a REPAIR frame. The loss of a single REPAIR frame does not lead to the transmission of a RECOVERED frame since this loss does not impact the receiver, but it is still announced in ACK frames. The sender thus still receives a frequent loss signal when many REPAIR frames are lost. With a higher code rate such a $\frac{6}{7}$, losses of packets containing only REPAIR frames and the related congestion signal will be less frequent.

Our experiment consists in performing a 10MB file transfer with the regular QUIC implementation (we call it the *foreground transfer*) while the link is already fully utilised by another transfer (we call it the *background transfer*). We consider three different candidates for the background transfer: 1) another

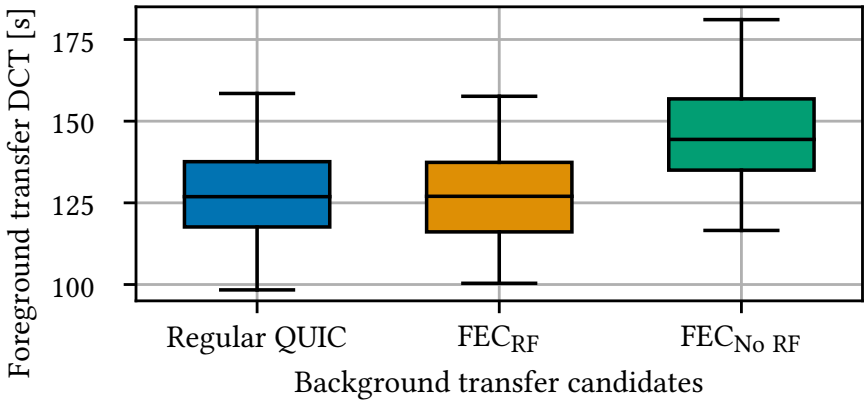


Figure 2.8: DCT of a regular QUIC transfer when competing with QUIC, QUIC-FEC with RECOVERED frames and QUIC-FEC without RECOVERED frames.

regular QUIC transfer, 2) a QUIC-FEC connection that uses RECOVERED frames (RF) and 3) a QUIC-FEC transfer that simply acknowledges the recovered packets, without sending RECOVERED frames. We compare the DCT of the foreground transfer in these three cases. The left, middle and right box plots in Figure 2.8 respectively represent the DCT of the foreground transfer for the first, second and third cases.

As we can see, the regular QUIC download takes generally longer when it competes with a QUIC-FEC transfer that does not send RECOVERED frames. This is due to the fact that in this case, the congestion signal is lost when packet losses caused by congestion are recovered and simply acknowledged by the receiver. This makes a FEC-enabled protocol unfair compared to traditional protocols such as regular QUIC that only use retransmissions. The middle box plot shows that when QUIC-FEC uses RECOVERED frames, there is no difference between a QUIC sender competing with another QUIC sender or a QUIC-FEC sender.

2.3.3 Results with bursty losses

In this section, we analyse the impact of using FEC in the case of correlated losses. We perform the same experiments with a Gilbert-Elliott loss model. The parameters of these experiments are shown in Table 2.1. We remove from our results the experiments whose intense loss patterns prevent a successful file transfer. We show the comparison of QUIC-FEC with RLC and the regular QUIC in Figure 2.9. We can see that the results are close to the results with uniform losses shown in Figure 2.5: using FEC has benefits for smaller transfers and the 1MB file transfer suffers from the added redundancy.

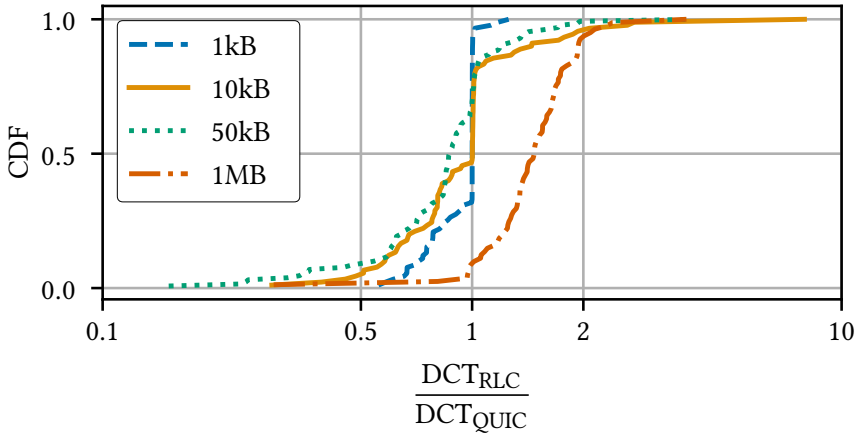


Figure 2.9: DCT ratio between QUIC-FEC with the RLC block code and QUIC, using the Gilbert-Elliott loss model.

When looking more closely at our results, it appears that using FEC performs badly when the r parameter of the Gilbert-Elliott model is low. For the download of 10kB and 50kB files, the average DCT ratio for our experiments when $r \leq 12\%$ is above 1, while it is below 1 otherwise.

2.4 Conclusion

This chapter presented a first revisit of the QUIC loss recovery mechanism. Using this extension, QUIC is able to timely recover from packet losses at the expense of additional bandwidth usage. Our design and implementation are modular and can support a variety of FEC techniques with different levels of redundancy. Using the RECOVERED frame, the sender can correctly adjust its congestion window when the receiver uses FEC to recover from packet losses.

Our evaluation over a wide range of network scenarios shows that a FEC-enabled loss recovery mechanism can bring significant benefits for small file transfers, especially when losses occur during the last packets flight. For such cases, the packet retransmission is often triggered after all application data have been sent. Sending repair symbols therefore avoids waiting for the retransmission to be sent. These experiments also showed us that FEC is not the only answer for a general QUIC loss recovery mechanism. Sending unused repair symbols can indeed significantly deteriorate the performance of troughput-driven transfers such as the bulk downloads studied in this chapter. FEC must therefore be used cautiously and in conjunction with the SR-ARQ loss recovery mechanism already provided by QUIC. Sensitivity to the network characteristics and to the application are two areas of improvement that we

explore later in this thesis.

We also compared blocks and convolutional codes and showed that the convolutional code can recover from single loss events more rapidly than block codes, leading to a better bandwidth efficiency by avoiding spurious retransmissions. For those reasons, we focus on convolutional codes in the next chapters of this thesis.

This first FEC extension also initiated a more general reflection on the interactions of network coding and loss-based congestion control mechanisms at the transport layer, by explicitly informing the peer of symbols recovery using the new RECOVERED frame. These thoughts led to discussions at the IETF, resulting to the RFC9265 document discussed in the next chapter.

The interactions between FEC and congestion control

3

Since FEC can repair lost packets, blindly applying FEC may easily lead to an implementation that also hides congestion signal from the sender, adopting an unfair behaviour towards uncoded connections. It is important to ensure that such hiding of information does not occur since packet loss may be the only congestion signal available to the sender (e.g., TCP New Reno [RFC5681] or Cubic [HRX08]). Hiding this signal may lead to the fairness problem exposed in the previous chapter. This chapter offers a discussion on how coding and congestion control should coexist. Another objective is to encourage the research community to also consider congestion control aspects when proposing and comparing FEC coding solutions in communication systems.

This chapter summarizes the personal contributions of this thesis to the IETF that led to the publication of the RFC9265 document [RFC9265]. This document represents the collaborative work and consensus of the Coding for Efficient Network Communications Research Group (NWCRG).

3.1 Symbols and packets are conceptually separate data units

We propose the model illustrated in Figure 3.1 to represent the mechanisms established on a FEC-enabled transport protocol. Classical transport protocols such as QUIC and TCP have a *Sender* and a *Receiver* component that respectively generate and consume network packets. The Sender component packs protocol information (e.g. QUIC frames) into packets that are then sent over the network. In the case of a transport protocol with FEC capabilities, it also disposes of FEC *Encoder* and *Decoder* parts.

When the application wants to send new data over the connection (left part of Figure 3.1), the Encoder encodes the application data into one or several source symbols and generates repair symbols protecting these when needed. These symbols are then packed into network packets by the Sender. In the case of QUIC-FEC discussed in Chapter 2, the Sender sets the FEC flag and adds the 32-bits header field when the packet contains a source symbol. When repair symbols must be sent, the Sender packs them inside REPAIR frames. On

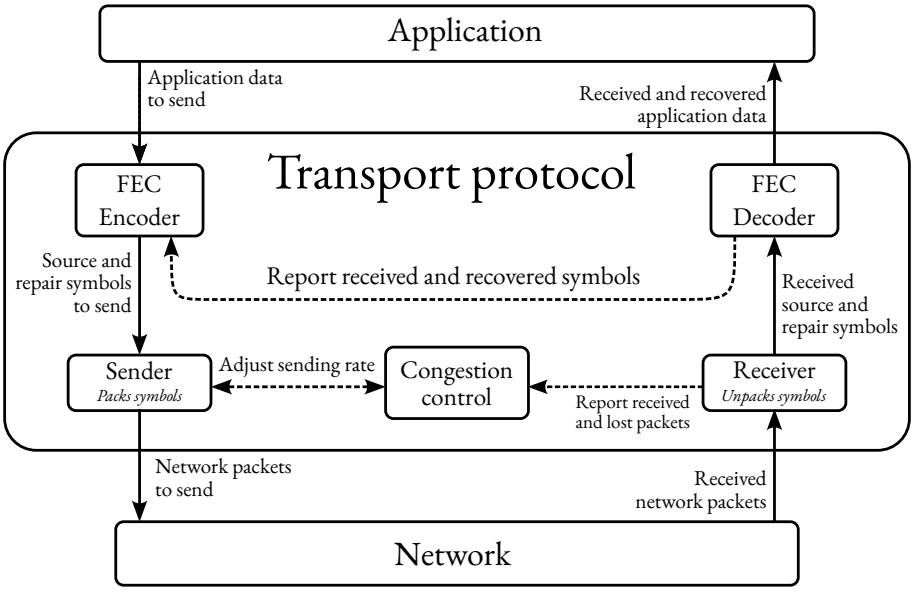


Figure 3.1: Coding-enabled transport protocol. Packets can only be received from the network while symbols can either be received as-is from the network or recovered through the FEC decoding process. Packet reception is therefore a separate signal from symbol reception.

the receiving path (right part of Figure 3.1), the Receiver consumes network packets and unpacks the symbols it contains. It provides the received symbols to the FEC Decoder that then recovers the lost source symbols if they exist. It finally passes the application data present in the newly received or recovered source symbols to the application.

As discussed in Chapter 1, the Sender has to adapt its sending rate to the network bandwidth to avoid causing congestion on the network. To do so, the congestion control component of the protocol takes feedback from the Receiver and adequately adjusts the sending rate of the Sender. Most congestion control mechanisms use packet loss as a signal of congestion in the network. This is based on the fact that congested routers drop newly arriving packets.

It is important to note that congestion control algorithms only care about network packets as they are the only data units dropped due to congestion. Network routers in this system have no notion of source or repair symbols. From Figure 3.1, we can see that only the Sender and Receiver components are directly connected to the network as they are the only two components that directly handle network packets. Recovering a missing source symbol on the decoder provides no additional information on whether the packet carrying it was actually dropped by the network.

3.2 Congestion control behaviour upon symbol recovery

Existing research work use the information that a symbol has been recovered to hide the related packet loss event from the congestion control algorithm [Sun+11; Gar+19]. Used conjointly with loss-based congestion control mechanisms such as Cubic or New Reno, this has the effect of avoiding the congestion window reduction upon packet loss, leading to a larger sending rate. The drawback of this approach is that the congestion control mechanism only receives an incomplete congestion signal, preventing it to correctly infer the congestion state of the network using its traditional algorithm. FEC-enabled connections can therefore become unfair to non-FEC-enabled ones since FEC-enabled transfers tend to ignore a significant part of the congestion signal, as discussed in Chapter 2.

This fairness problem led to several research recommendations that we included in the RFC9265 document, listed below.

1. From a congestion control point of view, a lost-and-recovered packet must be considered as a lost packet. This however does not apply to the usage of FEC on a path that is known to be lossy with non-congestion losses.

2. When a research work aims at increasing throughput by hiding the packet loss signal from congestion control (e.g., because the path between the sender and receiver is known to consist of a noisy wireless link), the authors should (*i*) discuss the advantages of using the proposed FEC solution compared to purely replacing the congestion control by one that ignores a portion of the encountered losses and (*ii*) critically discuss the impact of hiding packet loss from the congestion control mechanism.

QUIC-FEC presented in Chapter 2 follows (and led to) the first recommendation by signalling every recovered symbol using the RECOVERED frame. By doing so, the QUIC sender does not need to retransmit the recovered data while having the packet loss signal taken into account in its congestion control mechanism. The second recommendation comes from the fact that the gain in throughput brought by FEC is mainly due to the masking of losses to the congestion control algorithm. This behaviour can be accomplished without FEC by simply using a congestion controller ignoring packet losses. This is done by TCP/NC [Sun+11] that replaces loss-based congestion controls by Vegas that uses delay increase as a congestion signal [BOP94]. Kim *et al.* define a new congestion control algorithm reducing the congestion window backoff of New Reno based on the current delay increase [Kim+14]. RFC9265 formally recommends this to be the good practice when using network coding

as part of a transport protocol. QUIC-FEC does not entirely follow the second recommendation as recovered packets are acknowledged using ACK frames and then the congestion window is reduced upon receiving the RECOVERED frame. This technique adds an unneeded interaction between the congestion control algorithm and the FEC mechanism: the packet loss signal is hidden during the time between the reception of the ACK frame and the RECOVERED frame. The next chapters improve this mechanism in order to correctly follow the recommendations discussed here.

PQUIC: towards really flexible transport protocols

4

The extended loss recovery proposed by QUIC-FEC was designed and implemented using the existing extension mechanism provided by QUIC: it extends the packet header and uses new QUIC frames to define new protocol behaviours. To do so, the protocol implementation is modified and recompiled, to be finally deployed on clients and servers by updating both. Such extension mechanisms allowed transport protocols to evolve over recent decades [RFC7414]. Although the TCP specification is more than fourty years old, the protocol can be updated using extensions. A modern TCP stack supports a long list of TCP extensions (window scale [RFC1323], timestamps [RFC1323], selective acknowledgments [RFC2018], Explicit Congestion Notification [RFC3168] or multipath extensions [RFC8684]) that have been proposed along the years. However, measurements indicate that it remains difficult to deploy TCP extensions [Fuk11; Hon+14]. The window scale and selective acknowledgment options took more than a decade to be widely deployed [Fuk11]. While being part of the Linux kernel since version 5.6, Multipath TCP is only available on one major mobile OS [App18] and is not available on Android. This slow deployment of TCP extensions is caused by three main factors. First, popular stacks rarely implement TCP extensions unless they have been approved by the IETF. Second, TCP is still part of the operating system and client and server implementations are not upgraded at the same speed. Often, maintainers of client (resp. server) implementations wait until server (resp. client) implementations 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 as discussed in the previous chapters [Hon+11; Hes+13].

Google's first version of QUIC was proprietary and did not require IETF consensus to be updated. As QUIC runs above UDP it is possible to ship it as a library which can be updated as often as applications. Measurements indicate that Google updated its version of QUIC at the same pace as its Chrome browser [Rüt+18]. With this system, the real winners are companies

controlling both the server and the users' devices such as Google with Android and Apple with iPhones. As the QUIC stack does not sit in the operating system kernel anymore, these actors can update the QUIC implementation unilaterally on their servers and on the users' devices.

Other service providers only have limited control on how the transport stack can behave. While they can easily tune the QUIC stack running on the server, it is not always possible for them to customize the QUIC protocol running on the client. For instance, a web browser application can only access QUIC through HTTP/3 or through the WebTransport API [W3C23b]. Complex extensions such as FEC can easily be implemented and shipped in browsers by Google if they need it, but it is not doable by smaller Internet actors as they cannot modify the browser source code.

In this chapter, we completely revisit the extensibility of transport protocols. We model the transport protocol as a set of basic functions that can be tuned, combined and dynamically extended to support new use cases on a per-connection basis. Such an approach enables QUIC applications to adapt the underlying transport layer to their specific needs, e.g., using specialized retransmission algorithms or taking advantage of non-standard extensions. The work developed in this chapter eases the deployment of a heavy extension such as FEC and is a first step to enable the use of a loss recovery mechanism tailored to the application, explored in Chapter 5. We make four main contributions in this chapter.

- We design a technique where an extension to the QUIC protocol is broken down in a set of *protocol plugins* which can be dynamically attached to an existing implementation. These plugins interact with this implementation through code that 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` [Hui22a], one of the most complete implementations of IETF QUIC [PDB18] at the time of writing this work. We add to `picoquic` a virtual machine that allows executing the bytecode of protocol plugins in a platform independent manner while monitoring their behavior.
- In this chapter, we especially focus on the interest of PQUIC in redifining more easily the protocol's loss recovery. We design and implement our Forward Erasure Correction technique using only protocol plugins.

This work has been published and presented at the SIGCOMM'19 conference and has been done in collaboration with several colleagues [De +19].

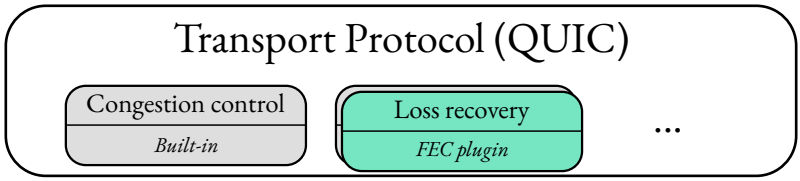


Figure 4.1: Using PQUIC, protocol behaviours such as loss recovery can be redefined on a per-connection basis by inserting protocol plugins.

This chapter focuses on the interest of PQUIC in redifining the protocol's loss recovery. We design and implement our Forward Erasure Correction technique using only protocol plugins.

4.1 Pluginizing QUIC

From the implementation viewpoint, the main difference between a PQUIC implementation and a QUIC one is that PQUIC is easily customizable on a per-connection basis. This customization relies upon a modular and 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. Figure 4.1 illustrates how we decompose the transport protocol as a set of behaviours that can be redefined. The protocol uses the built-in behaviours of the implementation by default, but applications can override these default behaviours by inserting protocol plugins. Like in Figure 4.1, an application can replace the default loss recovery behaviour by a FEC-enabled one using a FEC protocol plugin.

A PQUIC implementation provides an API for protocol plugins. Most protocol implementations are designed as black-boxes that expose 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. In PQUIC, we call these functions *protocol operations (protoop)*. 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 classical programming 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 the processing of frames, setting the retransmission timer, updating the RTT, removing acknowledged frames from the sending buffer, etc.

On-the-fly protocol plugin insertion. We use the word *pluglet* to refer to the set of bytecode instructions implementing a protocol operation. A *pluglet* consists of bytecode instructions implementing a function, e.g., computing

an RTT estimate. A protocol plugin consists of the set of all pluglets needed to implement some protocol behaviour, such as loss recovery or congestion control. Once a PQUIC connection is established, PQUIC can potentially load plugins at any time.

Isolation between connections and between plugins. Each plugin is instantiated to operate on a given connection. Our framework ensures that each instance has its own memory which is only shared among pluglets of this plugin. The plugin memory is isolated from access or sharing with any other plugin or connection. This yields strong memory safety guarantees for the plugins and the sharing of information. Interactions are still possible through the protocol operation interface or by calling the functions exposed by PQUIC. However, these are clearly defined information flows that ease reasoning about the behavior and the safety of the plugins.

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

4.1.1 Pluglet Runtime Environment (PRE)

Pluglets are the building blocks of the protocol plugins. These pieces of bytecode are independent of the PQUIC implementation itself. Therefore, 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 these two issues, PQUIC executes plugins inside a lightweight virtual machine (VM). Various VMs have been proposed for different purposes [Lin+14; Haa+17; Wan+15; Geo+10; BMG99; Fle17b]. In this paper, our *Pluglet Runtime Environment* (PRE) relies on a user-space implementation [IO 18] of the eBPF VM [Fle17b] present in the Linux kernel since 2014 where it has been used to support various services [Edg15; Gre15; Bra17]. Similarly to the Linux kernel, PQUIC allows performing Just-In-Time (JIT) compilation of eBPF bytecode into native host CPU instructions to enhance the performance of the running plugins. For kernel security reasons 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.

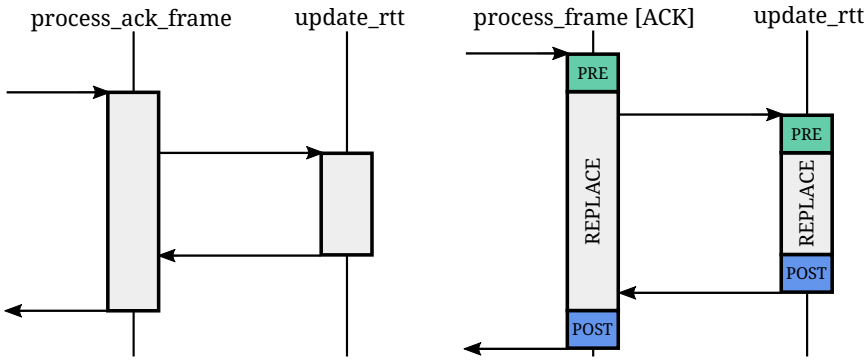
Our implementation extends a relaxed version of the eBPF verifier with additional monitoring capabilities. Those are similar to works in Software-Based Fault Isolation [Wah+94; Yee+09]. First, our PRE checks simple properties of

the bytecode to ensure its apparent validity. This includes checking that: (*i*) the bytecode contains an exit instruction, (*ii*) all instructions are valid (known opcodes and values), (*iii*) the bytecode does not contain trivially wrong operations (e.g., dividing by zero), (*iv*) all jumps are valid, and (*v*) the bytecode never writes to read-only registers. Furthermore, our PRE statically verifies the validity of stack accesses. A plugin is rejected if any of the above checks fails for one of its pluglets.

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

4.1.2 Protocol Operations

In order to attach pluglets to PQUIC, we define an API identifying the different protocol operations where the pluglets can be attached. Each protocol operation has its own name, inputs, outputs and specifications. Some protocol operations can also be parametrized so that different pluglets implementing different behaviours can be inserted for different parameter values. The most straightforward example being the protocol operations reading and writing frames: the parameter is the QUIC frame type and different pluglets can be inserted concurrently to read and write different types of frames. This allows implementing new types of QUIC frames (e.g. the REPAIR frame used in QUIC-FEC) only relying on protocol plugins and without impacting the processing of other frames. Our PQUIC implementation currently includes 72 protocol operations. Four of them take a parameter. 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, estimation, deciding which stream is to send next, etc. A third category involves QUIC packet management. It includes setting the *Spin Bit* [RFC9000], retrieving the connection IDs, etc. A fourth category concerns protocol operations that are not part of the base protocol but that are defined by protocol plugins themselves. For instance, a protocol plugin can define new protocol operations providing packet loss or one-way delay estimations that



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

Figure 4.2: Turning a monolithic design into protocol operations with the ACK processing example. It also illustrates the different anchors for the pluglets.

can be called and used by other protocol plugins.

To illustrate how an implementation can be split into protocol operations, consider the example shown in Figure 4.2a. 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 in Figure 4.2b, PQUIC functions are wrapped by a protocol operation whose name describes its goal. While the name of the protocol operation and the original function are similar, the processing of ACK frames is linked 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: `pre`, `replace` and `post`. Each anchor is a possible insertion point for a pluglet. Protocol operations with parameters propose a specific set of anchors for each parameter value. The `replace` anchor, consists of the actual implementation of the operation. A default behaviour is usually provided by the original QUIC implementation. Inserting code on the `replace` anchor enables a pluglet to override this default behavior. Because it may modify the connection context, inserting several pluglets on a `replace` anchor may cause conflicts between the pluglets. Therefore, at most one pluglet can `replace` a given protocol operation. If a second one tries to replace the same operation, it will be rejected and the plugin it belongs to will be rolled back. The two other anchors, `pre` and `post`, attach the plugin just before (resp. just after) the protocol operation invocation. These modes are similar to the eBPF kprobes in the Linux kernel [Ken+16]. By default, those are no-ops in PQUIC. Unlike the `replace` anchor, any number of `pre` and `post` pluglets can be inserted for a given protocol operation. However, they only have read access to the

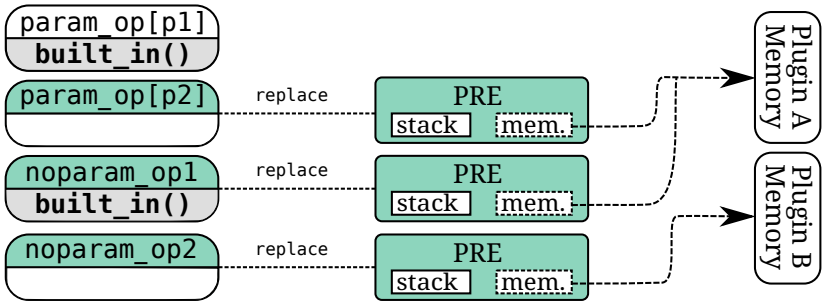


Figure 4.3: Attaching pluglets in `replace` mode to protocol operations.

connection context, protoop arguments and protoop outputs. The only write accesses they have is to their pluglet stack and their own plugin-dedicated memory. In the rest of the chapter, unless explicitly stated, we discuss pluglet insertions in `replace` mode, and refer to pluglet inserted in `pre` and `post` as passive pluglets.

4.1.3 Attaching Protocol Plugins

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

To address these needs, PQUIC is organized as illustrated in Figure 4.3. As explained in the previous section, the behavior of a protocol operation is either provided by a built-in function (e.g. `param_op[p1]` in Figure 4.3) or overridden by a pluglet (e.g. `noparam_op1`). 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 Plugin Runtime Environment (PRE) is created for each inserted pluglet. Each PRE contains its own registers and stack. The PRE heap memory points to an area common to all pluglets of a plugin, as illustrated in Figure 4.3. This link, ensured by the PQUIC implementation, provides pluglets with a communication channel through shared-memory. In addition to the isolation benefits, this architecture ensures that aggressive or ill memory management only affects the plugin itself. Thanks to our PRE, pointer dereferencing is restricted only to the pluglet stack or its plugin memory. In addition, pluglets also need to communicate with the host implementation to interact with the connection. Similarly to related work [AW18; Wir+19; Wir], PQUIC exposes

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

Table 4.1: PQUIC API exposed to pluglet bytecode.

some functions to the PRE. These functions form an API that pluglets can use (Table 4.1). We detail its six major operations below.

Exposing connection fields through getters and setters. Letting plugins directly reference the fields of PQUIC structures makes the injected code very dependent on PQUIC internals such as its memory layout. 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, this interface abstracts the implementation internals from the pluglets, making them compatible with different PQUIC versions or implementations. In addition, it allows the PQUIC host to monitor and control the fields accessed by the injected code. A host could thus reject plugins based on the fields that it wishes to access. For example, a client could refuse plugins that modify the *Spin Bit*, as it is not encrypted. Similarly, depending on its local user policies, a host could accept or deny a plugin accessing the TLS state.[1]

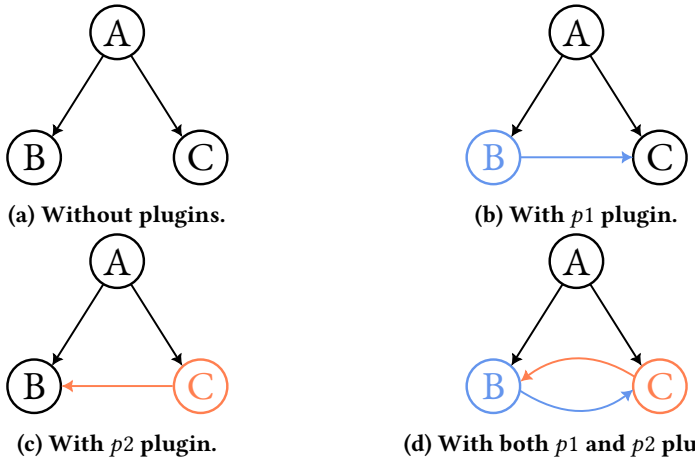
Managing plugin memory. Pluglets might need to keep persistent data across calls. Therefore, we provide `pl_malloc` and `pl_free` to allocate and free memory in the plugin dedicated area. Our framework dedicates a fixed-size memory area split into constant size blocks [Ken12]. Such approach provides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

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

Modifying connection memory area. Plugins might need to modify memory outside the PRE. For instance, a pluglet might need to write a new frame inside a buffer. This can be done by using `pl_memcpy` and `pl_memset`. The API keeps control on the plugin operations by ensuring the pluglets are allowed to access the specified memory area.

Calling other protocol operations. This is required when a protocol op-

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



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

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

Figure 4.4: 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.

eration depends on another one and is done using the `plugin_run_protoop` function. However, such a capability raises potential safety issues. As plugins can call any protocol operation, a PQUIC implementation needs to take care of possible infinite calling loops due to these calls. Keeping a loop-free protoops call graph allows preventing such call loops. However, ensuring this property for any combination of loop-free plugins is not practical to assess before executing them due to the combinatorial state explosion. Consider the example shown in Fig. 4.4. There are three protocol operations A, B and C, all guaranteed to terminate. Even if both $p1$ and $p2$ plugins are legitimate, their combination might introduce an infinite loop, as shown in Figure 4.4d. To avoid this situation, a PQUIC implementation keeps track of all the currently running protocol operations in the call stack. If a call is requested for an operation that is already running, PQUIC raises an error.

Scheduling the transmission of QUIC frames. PQUIC provides a way for pluglets to reserve a slot for sending frames using the `reserve_frames` function. However, it should enforce two rules. First, plugins must not prevent PQUIC from eventually 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 such as a FEC plugin should not be able to starve other plugins. Concurrently active plugins should have a 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 a percentage of the available congestion window. A deficit round robin scheduler then distributes the remaining budget between the plugin frames.

4.1.4 Interacting with Applications

We showed how plugins can interact within PQUIC. Plugins can also interact with the application using PQUIC. This allows them to extend the application-facing interface of PQUIC to bring new functionalities. For example, a plugin could implement a message mode for QUIC to supplement the standardized ordered byte-stream or datagrams abstractions [RFC9000; RFC9221]. This communication is established in a per-plugin bidirectional manner. First, an application can call *external* protocol operations. These are new anchor points that can be defined when injecting pluglets. The *external* mode is similar to the `replace` anchor, but it makes the protocol operation only executable by the application. This allows it to directly invoke new methods, e.g. queuing a new message to send. Second, a plugin can asynchronously push messages back to the application, so that it remains independent of the application control flow. Interactions between the application and protocol plugins are used extensively in Chapter 5.

4.2 Extending the loss recovery using protocol plugins

Several QUIC extensions were implemented with the PQUIC framework [De +19]. Adding Forward Erasure Correction to the loss recovery mechanism is probably the most substantial of them. Such an extension needs to update a good part of the protocol behaviours and access and modify most of its state. In order to encode and decode symbols, the FEC plugins additionnally need to access the packets content. Being able to meet all these requirements demonstrates the flexibility of PQUIC. Finally, this extension is also computationally-intensive and is therefore a good candidate for assessing the performance impact of using protocol plugins.

We leverage the PQUIC framework to implement a flexible framework inspired by QUIC-FEC from Chapter 2. Our plugin sends repair symbols to enable PQUIC receivers to recover lost QUIC packets without waiting for retransmissions and therefore meeting the delay constraints.

4.2.1 Design & implementation

Our FEC plugin allows plugging different FEC Frameworks: we implemented block and sliding-window-based codes. These block and window frameworks can be used interchangeably without modifying the base FEC plugin. The

plugin adds several protocol operations and can be extended to change the FEC framework and the underlying erasure-correcting code.

Our frameworks attach passive pluglets to the protocol operations that send and receive QUIC packets. Each packet containing STREAM frames will be protected by sending repair symbols later. On the receiver-side, the FEC-protected packets are added to their respective FEC encoding window. The missing packets of a window are recovered upon reception of repair symbols.

Similarly to QUIC-FEC discussed in Chapter 2, our FEC plugin defines the REPAIR frame containing a repair symbol. The plugin also defines the FEC ID frame that indicates that the packet payload is to be considered as a source symbol and is protected by FEC. This replaces the FEC flag and the 32-bits packet header added by QUIC-FEC in Chapter 2.

Using the new protocol operations added by our framework, we provide two pluglets implementing different erasure correcting codes. The first one implements a XOR code similar to the one proposed by Google [IT16]. It is thus simple to compute and can be used on lightweight clients. It can however only recover from the loss of a single packet, as only one repair symbol can be generated from the same encoding window. The second code is a Random Linear Code (RLC) [FLW06] where the repair symbols are generated by computing a random linear combination between all the packets of the encoding window. Compared to the XOR code, RLC is computationally heavier but can recover from the erasure of more than one packet. However, its recovery process, i.e. solving a system of linear equations whose unknowns are the lost packets, is more computationally intensive. Other erasure-correcting codes such as Reed-Solomon could easily be added by implementing new pluglets.

Our FEC plugin also includes other protocol operations to customize its behavior. One can choose between protecting the entire data transfer or only the last packets of a stream by using different pluglets. The first mode is similar to the behaviour of QUIC-FEC. The second mode can reduce the Download Completion Time (DCT) of a bulk data transfer by recovering erasures occurring during the last flight of packets while reducing the bandwidth usage for repair symbols. In total, the FEC plugin defines 51 pluglets to define new protocol behaviours and access specific QUIC anchor points.

4.2.2 Evaluation

In this section, we evaluate our FEC plugin in the In-Flight Communications use-case discussed in Chapter 2. This evaluation follows an experimental design approach exploring the same network parameters range as in Chapter 2, namely $\{d_1 \in [100, 400], bw_1 \in [0.3, 10], l_1 \in [1, 8]\}$, based on the experimental results of Rula et al. [Rul+18]. Figure 4.2.2 compares the performance of

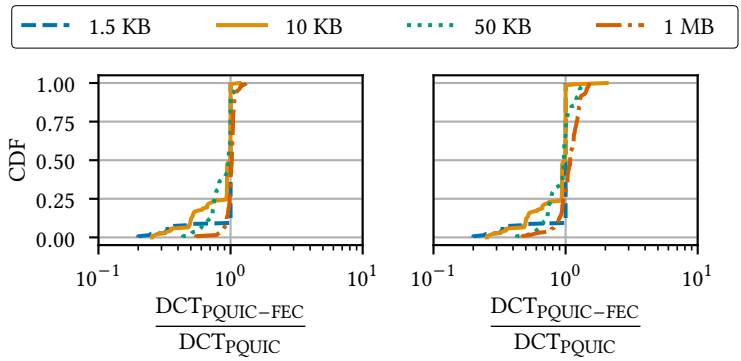


Figure 4.5: DCT ratio between PQUIC with and without the FEC plugin. Left: only the end of stream is protected. Right: The whole stream is protected.

downloading a regular HTTP object with and without the FEC plugin. On the left graph only the end of the data stream is protected (i.e., some repair symbols are only sent at the end of the connection), while on the right graph the whole data stream is protected, by sending 5 repair symbols every 25 source symbols. As we can see, there is a benefit in only protecting the end of the stream for larger file transfers. Protecting the whole transfer requires more bandwidth which negatively impacts the DCT. As discussed in Chapter 2, packets lost in the middle of the transfer can be easily recovered through classical retransmissions without significant impact on the DCT.

Only sending redundancy at the end of the transfer considerably lowers the negative impact we saw with QUIC-FEC in Chapter 2 but still provides latency improvements when last flight losses occur. This is a first example showing that although the code rate is still fixed, adapting the redundancy scheduling to the use-case is beneficial. The bulk transfers considered here do not need a constant protection like real-time media communications would. FEC mostly brings benefits here when last flight losses occur. Using protocol plugins, applications can insert the plugins that suit their needs and finely tune the loss recovery mechanism. This possibility is explored in details in the next chapter.

4.2.3 Plugin Overhead

The balance between flexibility and raw performance is a classical trade-off. Although Google measured that "*QUIC's server CPU-utilization was about 3.5 times higher than TLS/TCP*" [IS18] back in 2018, QUIC itself was chosen over TCP by Google due to its features and flexibility improvements. The performance gap between TCP and UDP has since been reduced [Jae+23], with some QUIC implementations reaching a rate of several Gigabits per second

Plugin	$\tilde{x}$ Goodput	$\sigma/\tilde{x}$	$\tilde{x}$ Load Time
PQUIC, no plugin	1104.2 Mbps	3.8%	0.0 ms
FEC XOR	516.6 Mbps	3.2%	11.71 ms
FEC RLC	187.4 Mbps	1.1%	11.21 ms
FEC XOR EOS	661.5 Mbps	3.2%	11.70 ms
FEC RLC EOS	648.0 Mbps	4.5%	11.22 ms

Table 4.2: Benchmarking plugins over 10Gbps links (20 runs).

on high-end CPUs, but TCP continues to provide significantly higher raw performance results. As PQUIC delegates the execution of the plugins to the PRE, the cost of its added flexibility is a processing overhead due to the JIT, the runtime verifications performed the PRE at each memory access and the utilization of the API to safely access PQUIC state variables.

Executing code in the PRE is less efficient than running native code. By compiling source code to eBPF instead of classical ARM or x86 CPU architectures, PQUIC renounces to most of their optimized instructions, such as SSE, AVX or ARM NEON that could be used by a plugin such as FEC to provide efficient computations. Finally, our get/set API is five times slower compared to direct memory accesses, due to memory-safety checks performed by the PRE at each call.

To observe this performance impact in bandwidth-intensive environments, we benchmark our PQUIC implementation by measuring the completion time of a 1 GB download between two servers with 10Gbps NICs running two Intel Xeon E5-2640 v3 CPUs. Note that PQUIC is single-threaded and thus does not benefit from the additional CPU cores.

Table 4.2 reports the median achieved goodput, its relative variance and the plugin loading time for several variants of the FEC plugin. *FEC XOR* (resp. *FEC RLC*) corresponds in the FEC plugins using the XOR (resp. RLC) erasure correcting code. FEC XOR EOS and FEC RLC EOS are the variants of the plugin only sending repair symbols protecting the end of the sent stream. PQUIC alone achieves a median goodput of 1104.2 Mbps.

As can be seen on the table, the FEC plugin causes a significant drop of performance. Two major factors affect this result. First, the specific FEC scheme used impacts the computational cost, as RLC is more expensive than XOR. On the RLC encoder, source symbols need to be multiplied by a coefficient before they get XORed. Decoding is performed through gaussian elimination which is a costly operation taking significantly more time as the number of symbols to recover increases. Second, FEC introduces a bandwidth cost by generating repair symbols over the network with the $\frac{5}{6}$ code rate, limiting the achieved goodput. Restricting their generation to the end of the stream (EOS) reduces this cost but still nearly divides by two the achieved goodput.

This performance degradation is the price of flexibility induced by the PQUIC approach. Reducing the performance overead of FEC for QUIC can be done with a native implementation using optimized CPU instructions. This is explored in Chapter 7.

Inserting plugins takes time, as described in the last column of Table 4.2. This time is proportional to the number of inserted pluglets and their complexity. The instantiation of PREs is the major contributor to this loading time. While not presented here, we developed a caching system for the PQUIC plugins. If the host previously loaded the plugin in a completed connection, caching allows it to reuse its PREs and load the plugin in less than 30μs instead of 11ms.

4.3 Validating Plugins

PQUIC allows peers to directly echange the bytecode of the protocol plugins to insert. Inserting arbitrary bytecode into the protocol implementation raises security issues. Although not detailed in this thesis, the original PQUIC article defines ways for peers to request proofs of the validity of protocol plugins when receiving them over a QUIC connection [De +19; Ryb+21]. This validation is carried out by agents called *Plugin Validators*. Validators can apply a range of techniques, from manual inspection, privacy checks [Ege+11; Li+15], to fuzzing [PAJ18] or using formal methods 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 during a PQUIC connection. To demonstrate the possibility of using formal techniques to validate protocol plugins, we have used the state-of-the-art T2 [BK23; CPR06] automated termination checker. This tool checks termination of programs written in the T2 language implementing a counter example-guided abstraction refinement procedure. This procedure builds on the seminal works on transition invariants (used to characterize termination) [PR04] and predicate abstraction (used to simplify the representation) [PR05] to build a proof of termination, or to disprove it. It is a counter-example based approach starting from an abstracted version of the system, and refining it until either no counter example to termination can be found, or there is a clear proof that the system does not terminate.

Using the appropriate tools [Khl+15; LLV23], we checked the termination of PQUIC pluglets by compiling their C source code to T2 programs. While proving termination is often difficult and sometimes undecidable, we succeeded to prove the termination of a good part of the pluglets written for PQUIC. The T2 prover assumes the termination of external functions, i.e., functions of the

PQUIC implementation available through the PRE. We proved the termination of most of the pluglets of the FEC plugin (37 pluglets out of 51 were proven to be terminating), and the XOR erasure correcting code was proven to be terminating in its entirety. To obtain those proofs, we had to slightly modify the source code of some pluglets to ease the verification process of T2. For example, we added an explicit size to null-terminated linked lists and used it to bound the loops iterating over the lists. Since T2 can export its termination proofs in files, these could be attached to the plugins to provide proof-carrying code as proposed by Necula [Nec02].

With its use of formal verification in addition to running the code in a sandboxed environnment, PQUIC improves its safety model one step further than what is done by web browsers nowadays with their unverified execution of Javascript and WebAssembly [Haa+17]. Recent research works have proposed to follow a similar approach with formal verification of WebAssembly code for the browser [Pro+19; Rao+23].

Our approach to PQUIC's verification has however some limitations. First, we limited ourselves to termination verification. Timeliness properties (i.e. the fact that a plugin has to terminate in a reasonable time) were not explored. In the kernel, eBPF ensures both these properties by limiting the size of a program and the number of instructions an execution path can take. These limits were low by the time of writing the PQUIC article (programs could not exceed 4096 instructions) and we made the choice of loading arbtrarily-sized programs, verifying their termination through formal verification. Aside timeliness, the correctness property was also left unexplored in the PQUIC article due to a lack of time and resources. Some programming languages such as F* [Swa+16] are explicitly designed to easily define and verify the programs specifications, while others are extended with verification frameworks such as the recent `prusti` for the Rust language [Ast+22]. PQUIC plugins could be written using these tools before being compiled to eBPF, performing the verification on the source code at compile-time, which is ofter easier than verifying the bytecode itself.

4.4 Conclusion

After having explored a first time the extensibility of the QUIC loss recovery with QUIC-FEC, we proposed in this chapter a general approach for extending transport protocols. We presented Pluginized QUIC (PQUIC) a new extensibility model composed of a set of *protocol operations* which can be enriched or replaced by *protocol plugins*. These plugins are bytecodes executed by a Protocol Runtime Environment that ensures their safety and portability. The plugins can be dynamically loaded by an application that uses PQUIC or received from a remote host thanks to PQUIC's secure plugin management

system. We showed in this chapter that a simple FEC extension can then be built using this new extensibility model. PQUIC can be the starting point for a new, modular 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, similarly to what has been done when *x*BGP plugins framework to several BGP implementations [Wir]. This set should be simple enough to allow very different implementations, having possibly very specific internal architectures such as zero-copy support, to inter-operate. Instead of adding more and more features to a monolithic implementation, developers could leverage the inherent extensibility of a pluginized protocol to develop a simple set of kernel features that are easy to extend.

FlEC: application-tailored loss recovery using protocol plugins

5

With PQUIC in the previous chapter, we introduced a new way to extend the QUIC protocol and showed the flexibility of the approach with a simple FEC plugin. With this plugin, we showed that there is an interest in adapting the redundancy scheduling to the underlying application. HTTP transfers do not need a constant FEC protection throughout the whole download. Sending repair symbols at the end of the transfers provides similar benefits to the ones obtained with QUIC-FEC in Chapter 2 without the increased overhead. Yet, this first plugin was simple and focused on bulk file transfers. This chapter addresses other kinds of network transfers and explores further the possibilities brought by the PQUIC paradigm. We propose a novel approach for providing a truly flexible QUIC loss recovery mechanism. We adapt the redundancy scheduling to the application's needs and traffic pattern instead of limiting ourselves to loss rate adaptation.

We design, implement and evaluate our Flexible Erasure Correction (FlEC) framework built atop PQUIC. With FlEC, an application can easily select the reliability mechanism that meets its requirements, from pure retransmissions to various forms of FEC. We consider three different use cases: (*i*) bulk data transfers, (*ii*) transfers with restricted buffers and (*iii*) delay-constrained messages such as real-time video communications. We demonstrate that a modern transport protocol such as QUIC may benefit from application knowledge by leveraging this knowledge in FlEC to provide better loss recovery and stream scheduling. The article discussed in this chapter was published in the April 2023 issue of IEEE/ACM Transactions on Networking [Mic+23a].

5.1 Introduction

While retransmissions currently remain the prevalent technique to recover from packet losses in QUIC, we showed that coding techniques added to the QUIC loss recovery mechanism could bring benefits for small bulk transfers. QUIC being initially designed for the Web, bulk transfers constitute a straightforward and representative example to experiment with. The previ-

ous chapters did not experiment with latency-sensitive applications such as video-conferencing, where latency and responsiveness are more valued than pure throughput. Traditionally, applications tend to use separate transport protocols for their different requirements. TCP is generally preferred when throughput and reliability are required while latency-sensitive applications generally rely on UDP or RTP. For instance, while most browser applications and websites rely on regular TCP and HTTP, web browsers provide the WebRTC protocol for real-time applications to meet their latency requirements. WebRTC [RFC8825] itself relies on UDP, RTP and SCTP underneath. With its large set of features, QUIC has nearly all the necessary mechanisms to act as a multi-purpose protocol and welcome this class of latency-sensitive applications, the exception being its loss recovery mechanism still fully based on SR-ARQ. Supporting different applications having competing needs with a FEC loss recovery mechanism complexifies the redundancy scheduling. Video-conferencing applications may benefit from regularly sent repair symbols while they deteriorate the quality of experience of bulk transfers due to goodput reduction. The flexibility of PQUIC and the possibility to insert different plugins for different applications opens new ways to ensure good transfer performance for different applications. Our main contribution in this chapter is our FlEC framework enabling applications to finely tune the transport loss recovery mechanism to closely fit their needs. We implement our solution entirely using QUIC and protocol plugins, but our ideas are generic and can be applied to other protocols as well. We evaluate the flexibility of our techniques through reproducible simulations and go beyond the simple bulk transfer use-case considered until now. We show that our application-tailored loss recovery mechanism outperforms one-size-fits-all SR-ARQ and FEC solutions.

This chapter is organised as follows. We first discuss how flexibility is currently provided by existing transport solutions by tracking the network loss characteristics (Section 5.2). We then present FlEC and its design. (Section 5.3). We implement FlEC atop PQUIC (Section 5.4) and demonstrate the benefits of this approach by studying three different use-cases (Sections 5.5-5.7) with competing needs that can all improve their quality of experience using FlEC.

5.2 Adaptive Forward Erasure Correction

Sending repair symbols for delay-sensitive applications is done at the cost of bandwidth when there is no loss to recover. This is why repair symbols should be sent carefully to avoid consuming bandwidth with no or low additional benefit. Some adaptive FEC mechanisms have been proposed to adjust the redundancy overhead to the measured network loss rate. Both CTCP [Kim+12] and TCP/NC [Sun+11] adjust the level of redundancy according to the estimated average loss rate. rQUIC [Zve+21] proposes a similar idea. While these

approaches can show significant benefits compared to classical retransmission mechanisms, they still increase the overhead compared to the more efficient selective-repeat mechanisms in bulk transfer use-cases: every unused repair symbol is a waste of bandwidth, similarly to spurious retransmissions. With FlEC, we want to both react to the current channel conditions and adopt similar behaviours to SR-ARQ when it is needed by the application requirements.

QUIC stacks are mostly implemented as libraries that can be used by a wide range of applications. While QUIC can easily be tuned on the server-side to better fit the use-case, obtaining such a flexibility on the end-user devices is more complicated, as the application needs to tune the underlying stack to meet its requirements. In the TCP/IP stack, this tuning is mainly done by using socket options or system-wide parameters. Socket Intents [Sch+13] and ongoing work [Pau+23] within the IETF TAPS working group show that there is an interest to transfer some knowledge from the application to the transport protocol. The QUIC specification [RFC9000] does not currently define a specific API between the application and the transport protocol but specifies a set of actions that could be performed by the application on the streams (e.g. reading and writing data on streams) and on the connection itself (e.g. switching on/off 0-RTT connection establishment or terminating the connection). The QUIC specification allows the application to pass information about the relative priority of the streams. However, it is still unclear how an application could for instance express timeliness constraints.

On the other side, the IETF FEC specifications for QUIC we developed during the course of this thesis [Swe+20b; MB22] do not guide the application to choose a code rate nor which parts of the application data should be FEC-protected and when coded symbols should be sent. Furthermore, different applications may require different strategies to send redundancy. In a video-conferencing application, repair symbols could protect a whole video frame. An IoT application [Egg20] with limited buffers may want to protect the data incrementally to ensure a fast in-order delivery despite losses.

In 2020, Cohen *et al.* proposed AC-RLNC [Coh+20]. This theoretical work provided a joint coding scheme and algorithm in which one can manage the delay-throughput tradeoff expressed by an application to get the required QoS. However, this work was theoretical and not designed for a transport protocol such as QUIC. Therefore, it did not cope with the complex and various requirements of real applications, more specifically their traffic pattern or exact time constraints. In this chapter, we start from the theoretical algorithms of AC-RLNC. We join our forces with its authors and push further the idea of adapting the redundancy scheduling to the application. We leverage the flexibility of protocol plugins and implement our reliability mechanism as a framework exposing two simple anchors points for applications to attach plugins. Applications can redefine these anchors according to their delay-

sensitivity and traffic pattern. We explore three different use-cases and show that adapting the reliability mechanism to the use-case can drastically improve the quality of the transmission. The three next sections discuss the use-cases considered in this chapter.

5.2.1 Bulk file transfer

We consider here the transfer of files under the assumption that the receive buffer is large compared to the bandwidth-delay product of the connection. This is the classical use case for many transport protocols. Current open-source QUIC implementations use default receive window sizes that support such a use-case. The receive window starts at 2 MBytes for locally-initiated streams in `picoquic` [Hui+21]. The Chromium browser's implementation [21] starts with an initial receive window of 6MB per stream and 16MB for the whole connection. Some QUIC clients such as curl [Curl] start with a small receive window but allow it to grow throughout the connection. The metric that we minimize in this scenario is the transfer completion time. This scenario also includes REST API messages that often need to be completely transferred in order to be processed correctly by the application. As already pointed out [Fla+13; MDB19; Lan+17] and discussed in the previous chapters, a packet loss during the last round-trip can have a high impact on the transfer completion time. The latter may indeed be more than doubled for small files due to the loss of a single packet. Protecting these last-flight packets can drastically improve the total transfer time at a cost significantly smaller than the cost of simply duplicating all these packets. On the other hand, protecting other packets than the tail ones with FEC can be harmful for the transfer completion time as shown in Chapter 2. The packet losses in the middle of the transfer can be recovered without FEC without any quiescence period provided that the receiver uses sufficiently large receive buffers.

5.2.2 Buffer-limited file transfers

In numerous network configurations, the available memory on the end devices is a limiting factor. It is common to see delays longer than 500 milliseconds in satellite communications, while their bandwidth is in the order of several dozens of Megabits per second [Tho+19; Kuh+20]. Furthermore, with the expansion of the optical fiber and 5G networks, some devices will have access to bandwidth up to 10Gbps [Rap+13; ARS16]. While the edge latency of 5G infrastructures is intended to be in the order of a few milliseconds [3GP19], the network towards the other host during an end-to-end transport connection may be significantly higher. For instance, this can be due to the physical distance between two hosts (transatlantic communications typically imply a latency above a hundred of milliseconds) or bufferbloat on the network when

proper AQM solutions [RFC8290] are still not deployed. Packet losses occurring on those high Bandwidth-Delay Product (BDP) network configurations imply a significant memory pressure on reliable transport protocols running on the end devices. To ensure in-order delivery, the transport protocol needs to keep the data received out-of-order during at least one round-trip-time, requiring receive buffer sizes to grow significantly for each connection. At the same time, QUIC is also considered for securing connections on IoT devices [Egg20; KD19]. Those embedded devices cannot dedicate large buffers for their network connections. Receive buffers that cannot bear the bandwidth-delay product of the network they are attached to are unable to fully utilize its capacity, even without losses. This typically occurs when the receive window is smaller than the sender's congestion window. Measurements show that TCP receivers frequently suffer from such limitations [Lan+17]. The problem gets even worse in case of packet losses as they prevent the receiver to deliver the data received out-of-order to the application. Keeping the out-of-order data in memory reduces the amount of new data that can be sent until the lost data is retransmitted. Sacrifying a few bytes of the receiver memory in order to handle repair symbols and protect the receive window from being blocked upon packet losses can drastically improve the transfer time. In such cases, FEC can be sent periodically along with non-coded packets during the transfer and not just at the end of the transfer.

5.2.3 Delay-constrained messaging

Finally, we consider applications with real-time constraints such as video conferencing. Those applications send messages (e.g. video frames) that need to be successfully delivered within a short amount of time. The metric to optimize is the number of messages delivered on-time at the destination.

FEC can significantly improve the quality of such transfers by recovering from packet losses without retransmissions, at the expense of using more bandwidth. Researchers have already applied FEC to video applications [CSA03; Pur+01]. Some [CSA03] take a redundancy rate as input and allocate the repair symbols given the importance of the video frame. Others [Pur+01] propose a congestion control scheme that reduces the impact of isolated losses on the sending rate. They then use this congestion control to gather knowledge from the transport layer to the application in order to adapt the transmission to the current congestion. We propose to directly transfer the application knowledge in the transport protocol to automatically adjust its stream scheduler and redundancy rate given the application's requirements.

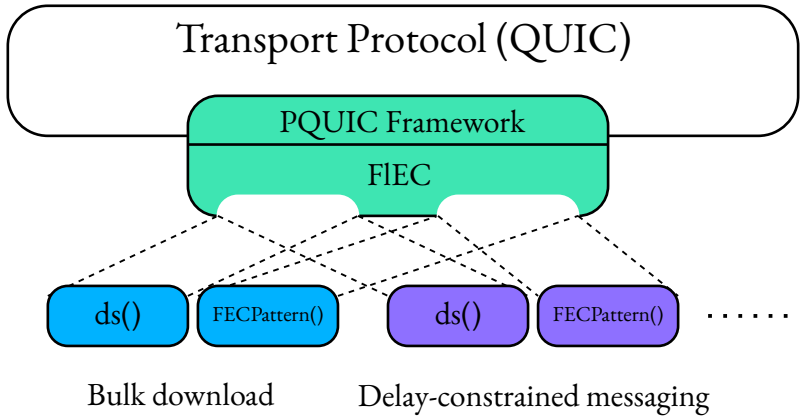


Figure 5.1: Design of the solution: a general framework with only two pluggable anchors to redefine the reliability mechanism given the use-case.

5.3 FlEC

In this section, we present the Flexible Erasure Correction (FlEC) framework. FlEC starts from a theoretical work, AC-RLNC [Coh+20]. This previous work proposes a decision mechanism to schedule repair symbols depending on the network conditions and the feedback received from the receiver. In this approach, repair symbols are sent in reaction to two thresholds: the first is triggered as a function of the number of missing degrees of freedom by the receiver (i.e. the number of symbols missing to perform FEC decoding), and the second threshold sends repair symbols *once every RTT*. The original goal of the proposed algorithm [Coh+20] is to trade bandwidth for minimizing the in-order delivery delay of data packets.

We start from this idea of tracking the sent, seen and received degrees of freedom as a first step to propose a redundancy scheduler for the transport layer. However, while this first idea provides a general behaviour, this may be insufficient for real applications with tight constraints that cannot be expressed with AC-RLNC's parameters. For example, a video-conferencing application may prefer to maximize bandwidth over low-delay links and therefore rely on retransmissions only, while FEC is needed over high-delay links as such retransmissions cannot meet the application's delay constraints. The principles of FlEC are illustrated in Figure 5.1. Instead of proposing configurable constant thresholds to tune the algorithm, we make it dynamic by proposing two redefinable functions: $ds()$ (for "*delay-sensitivity*") and $FECPattern()$. These two functions can be completely redefined to instantiate a reliability mechanism closely corresponding to the use-case. This allows having completely different FEC behaviours for use-cases with distinct needs such as HTTP versus video-conferencing. The $ds()$ threshold represents the sensitivity of the application

$\hat{l}$	the estimated uniform loss rate
$\hat{r}$	the estimated receive rate
$\hat{P}_{GB}$, (resp. $\hat{P}_{BG}$)	the estimated transition probability from the GOOD to the BAD (resp. BAD to GOOD) state of a Gilbert loss model [Ell63]
md	missing degrees of freedom
ad	added degrees of freedom
$ds()$	customizable threshold eliciting repair symbols given the application's delay sensitivity
$FECPattern()$	customizable condition to send FEC using the application's traffic pattern

Table 5.1: Definition of the different symbols.

to the in-order delivery delay of the sent data. In AC-RLNC, the FEC scheduler sends redundancy once per RTT. In FlEC, the $FECPattern()$ dynamic function allows triggering the sending of FEC at specific moments of the transfer depending on the application's traffic pattern. Sending FEC for every RTT may deteriorate the application performance, especially when the delay is low enough to rely on retransmissions only. Having a dynamic $FECPattern()$ function avoids this problem. For instance, in a bulk transfer scenario, it can trigger FEC at the end of the transfer only and rely on retransmissions otherwise. For video transfer, it can trigger FEC after each video frame is sent. In FlEC, the SR-ARQ mechanism used by QUIC is a particular case among many other possibilities. Algorithm 1 shows our generic framework and Table 5.1 defines the variables used by our algorithms. We implement FlEC using PQUIC and define $FECPattern()$ and $ds()$ as protocol operations. However, the same principles can be applied without PQUIC with the application redefining the operations natively thanks to the user-space nature of QUIC. This will be explored in Chapter 7.

The *computeMd* function computes the number of missing degrees (*md*) of freedom (i.e. missing source symbols) in the current coding window. The *computeAd* function computes the number of added degrees (*ad*) of freedom (i.e. repair symbols) that protect at least one packet in the current coding window. Compared to AC-RLNC, we only consider in-flight repair symbols in *ad* to support retransmissions when repair symbols are lost. The higher the value returned by $ds()$, the more likely it is to send repair symbols prior to the detection of a lost source symbol and the more robust is the delay between the sending of the source symbols and their delivery to the application. The extra cost is the bandwidth utilization. Sending repair symbols *a priori* occupies slots in the congestion window and is likely to increase the delay between the

Algorithm 1 Generic redundancy scheduler algorithm. The $ds()$ and $FECPattern()$ thresholds are redefined by the underlying application. The algorithm is called at each new available slot in the congestion window.

Require: $\hat{l}$, the estimated loss rate.
Require: $feedback$, the most recent feedback received from the peer.
Require: W, the current coding window.

$\hat{r} \leftarrow 1 - \hat{l}$
$ad \leftarrow computeAd(W)$
$md \leftarrow computeMd(W)$
if $feedback = \emptyset$ **then**
  if FECPattern() **then**
    return $NewRepairSymbol$
  else
    return $NewData$
  end if
else
  updateLossEstimations(feedback)
  if FECPattern() **then**
    return $NewRepairSymbol$
  else if $\hat{r} - \frac{md}{ad} < ds()$ **then**
    return $NewRepairSymbol$
  else
    return $NewData$
  end if
end if

Use-case	$ds()$	$FECPattern()$
Bulk transfer (SR-ARQ)	$-\hat{l}$	$false$
AC-RLNC [Coh+20]	$c \cdot \hat{l}$	*true* every RTT
Bulk transfer	$-\hat{l}$	$allStreamsFinished()$
Buffer-limited bulk	$c \cdot \hat{l}$	Algorithm 2
Messaging	$-\hat{l}$	Algorithm 4

Table 5.2: Definition of $ds()$ and $FECPattern()$ for the considered use-cases.

generation of data by the application and their actual transmission. Setting $ds()$ to $-\hat{l}$ in Algorithm 1 triggers the transmission of repair symbols only in reaction to a newly lost source symbol, implementing thus a behaviour similar to regular QUIC retransmissions. In this chapter, retransmissions are done using repair symbols to illustrate that the approach is generic, but classical uncoded retransmissions can be used for better performance without loss of generality. $FECPattern()$ allows regulating the transmission of *a priori* repair symbols regardless of the channel state, in contrast with AC-RLNC [Coh+20] where this threshold is triggered once per RTT.

Table 5.2 describes how $ds()$ and $FECPattern()$ can be redefined to represent reliability mechanisms that fit the studied use-cases. The first row of the table shows how to implement the classical default QUIC SR-ARQ mechanism using FlEC. The second one implements the behaviour of AC-RLNC [Coh+20], with $FECPattern()$ being triggered once every RTT according to the original alogrithm. The third one is tailored for the bulk use-case: $ds()$ is set to send repair symbols only when there are missing symbols at the receiver and $FECPattern()$ sends repair symbols when there is no more stream data to send. The two other rows are explained in details in the next sections. In this Table, c is a non-negative user-defined constant. The higher c is, the more sensitive we are to the loss rate deviation.

5.3.1 Comparing FlEC and previous work

The origin of FlEC comes from both the shortcomings of AC-RLNC [Coh+20] and QUIC-FEC. FlEC shares with AC-RLNC the idea of tracking the state of the communication in terms of received, seen and lost symbols. However, it adds the tight and diverse application requirements to the loop in order to adopt a correct behaviour for use-cases where FEC can be beneficial. It also adds all the transport-layer considerations such as being fair to the congestion control of the protocol upon loss recovery.

FlEC also builds upon QUIC-FEC as it integrates similar transport layer considerations. For instance, FlEC uses a similar wire format as well as the concept of RECOVERED frame in order to differentiate packet acknowledge-

ments from symbol recoveries. However, QUIC-FEC was designed without consideration for the application traffic pattern or channel condition: the packet redundancy was not adaptive at all.

5.4 Implementation

FlEC is composed of two parts. First, the general FlEC framework allows defining reliability mechanisms in a flexible way. This part is generic and is not intended to vary. The second part contains the $FECPattern()$ and $ds()$ operations. These operations are designed to vary depending on the use-case, so the app can redefine them based on their requirements.

We implement our FlEC framework inside PQUIC. We address the three use-cases discussed in this article by redefining $FECPattern()$ and $ds()$ to support the adequate reliability mechanism for each of them. Similarly to previous chapters, we rely on Random Linear Codes for the encoding and decoding of the symbols. This choice is made out of implementation convenience although other error correcting codes can be used as encoding/decoding tools with only little adaptation. Simpler codes such as Reed-Solomon can provide benefits for the considered use-cases although the benefit may be lower: simple block codes cannot mix the repair symbols of different generations conversely to random linear codes.

Our implementation of FlEC starts from the FEC plugin proposed in the previous chapter. We enhanced the $GF(2^8)$ RLC implementation to use dedicated CPU instructions and added an online system solver for faster symbols recovery. Most of the FlEC protocol operations consist in providing a shim layer between the PQUIC packet loop and the FlEC symbols scheduling algorithm.

The whole FlEC framework implementation spans 8200 lines of code. It adds a complete FEC extension to the QUIC protocol with the RLC error correcting code using PQUIC protocol plugins. This code is generic and does not have to be redefined by any application. The codes needed to define $ds()$ and $FECPattern()$ for the bulk and buffer-limited use-cases have been written with respectively 57 and 97 lines of C code while the code for the messaging use-case takes 335 lines of C code. These two small functions are the parts that can be redefined by each application to stick to its use-case.

5.5 Bulk file transfers

We here present the implementation and evaluation of the loss recovery mechanism proposed for bulk file transfers. The metric to minimize is the total transfer completion time. Sending unneeded repair symbols reduces the goodput and increases the transfer completion time. The expected behaviour is

therefore similar to SR-ARQ with last flight protection. The repair symbols are always sent within what is allowed by the congestion window, meaning that FlEC does not induce any additional link pressure.

5.5.1 Bulk loss recovery mechanism

For a file transfer, we set the delay-sensitivity threshold to be equal to $-\hat{l}$. The formula triggering the sending of repair symbols (line 17 of Algorithm 1) thus becomes the following.

$$\hat{r} - \frac{md}{ad} < -\hat{l} \rightarrow sendRepairSymbol() \tag{5.1}$$

Substituting $\hat{l}$ by $(1 - \hat{r})$ in Equation 5.1, we can rewrite it as

$$\frac{md}{ad} > 1 \rightarrow sendRepairSymbol() \tag{5.2}$$

so that we send repair symbols only when a symbol is missing and has not been protected yet. The transmission of a repair symbol triggered by this threshold increases *ad* by 1 until *ad* becomes equal to *md*. Using the threshold defined in Equation 5.1 ensures a reliable delivery of the data but does not improve the transfer completion time in the case of tail losses. *md* only increases after a packet is marked as lost by the QUIC loss detection mechanism. The *FECPattern*() operation controls the *a priori* transmission of repair symbols. In contrast with the previous solution [Coh+20], we redefine *FECPattern*() and set it to *true* only when all the application data has been sent instead of setting it to *true* once per RTT. This implies that only the last flight of packets will be protected. All the previous flights will be recovered through retransmissions triggered by Equation 5.2. We track the loss conditions throughout the transfer and trigger the *FECPattern*() threshold according to the observed loss pattern. This loss-rate-adaptive approach is especially beneficial when enough packets are exchanged to accurately estimate the loss pattern. This occurs when the file is long or when loss information is shared among connections with the same peer. When a sufficient number of repair symbols are sent to protect the expected number of lost source symbols, the algorithm keeps slots in its congestion window to transmit new data. Another approach would be to define *FECPattern*() to use all the remaining space in the congestion window to send repair symbols, with the drawback of potentially consuming more bandwidth than needed.

5.5.2 Evaluation FlEC for the bulk scenario

We evaluate FlEC with the *ds*() and *FECPattern*() protocol operations defined for this specific use-case. We base our implementation on PQUIC on commit

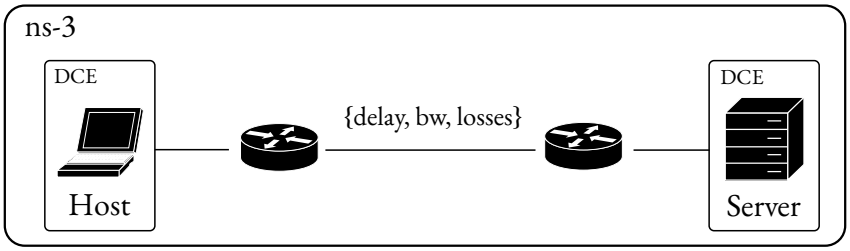


Figure 5.2: Experimental topology using NS-3 with Direct Code Execution.

68e61c5 [PquicRepo]. We perform a large number of experiments and compare it with the regular QUIC loss recovery mechanism without plugins. We use ns-3 [RH10] version 3.33 along with the Direct Code Execution (DCE) [Cam+14] module. The DCE module allows using ns-3 with the code of a real implementation in a discrete time environment. This means that the actual code of the QUIC and FlEC implementations is running and that the underlying network used by the implementation is simulated by ns-3, making the experiments fully reproducible while running real code. Figure 5.2 shows the experimental setup. We use ns-3's packet erasures models to generate reproducible loss patterns with different seeds, representing both uniform and bursty losses. We configure the network queues to 1.5 times the bandwidth-delay product.

Although congestion control is orthogonal to our proposed loss recovery mechanism, the Reno [RFC6582] and Cubic [HRX08] congestion control algorithms supported by PQUIC suffer from bandwidth underestimation under severe loss conditions. We thus perform experiments using the BBRv1 [Car+17] congestion control algorithm. BBR avoids underestimating the network bandwidth upon packet losses as it directly looks at the receive rate and delay variation during the transfer. As discussed earlier, other congestion control algorithms [Car+16b; BOP94] use other signals than packet losses to detect congestion. They are not explored in this chapter as it would require to implement them from the ground-up and validate their behaviour, which is out of scope for this thesis.

5.5.2.1 Experimental design

We evaluate the bulk use-case by sending files of several sizes and first see how FlEC compares with QUIC using its regular loss recovery mechanism. For this evaluation, we rely on an experimental design similarly to what we did in the evaluation of Chapters 2 and 4. We use the WSP [SCS12] space-filling algorithm to cover the parameters space with 95 points. One experiment is run for each point in the parameters space.

Figure 5.3 shows the CDF of the Transfer Completion Time (TCT) ratio

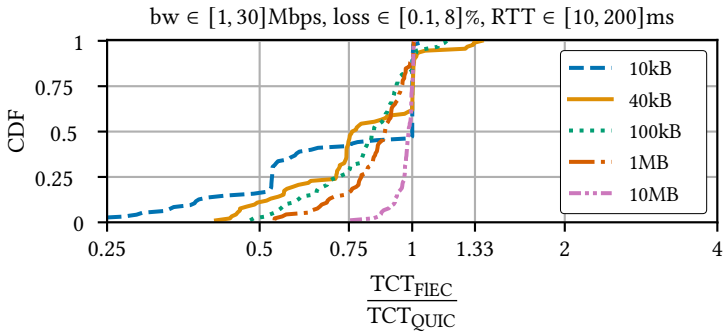


Figure 5.3: TCT ratio for bulk use-case using BBR. *FECPattern*() **and** *ds*() **ensure that repair symbols only protect the tail of the file.**

between FlEC and `picoquic` [Hui22a] used as our reference QUIC implementation. The experiments consist in the transfer files of size 10kB, 40kB, 100kB, 1MB and 10MB. For each file size, 95 experiments are run using experimental design. 40kB and 100kB are the average response sizes for Google Search on mobile and desktop devices in 2017 [Lan+17]. The explored parameters space is described on top of the Figure. The loss rate varies between 0.1% and 8% to cover both small loss rates and loss rates experienced under intense network conditions such as In-Flight Communications [RBC16]. The round-trip-time varies between 10ms and 200ms. As shown in the Figure, *ds*() and *FECPattern*() implement here a bulk-friendly loss recovery mechanism. By automatically protecting the tail of the transferred file, we obtain the beneficial results of QUIC-FEC (Chapter 2) without the overhead drawbacks, similarly to Chapter 4. A few of the experiments with 40 and 100kB files provided poorer results compared to QUIC. With those file sizes, FlEC uses one more stream frame to transmit the data, needing in some rare cases one additional round-trip to transmit this additional packet. While not shown graphically in this chapter, replacing BBR by Cubic [HRX08] provides similar results.

Figure 5.4 compares FlEC with an implementation of AC-RLNC [Coh+20] following Table 5.2. We observe that FlEC still outperforms AC-RLNC as sending repair symbols every RTT consumes too much bandwidth for the bulk use-case. FlEC only sends repair symbols *a priori* for the last flight of packets, relying on retransmissions for all the other packets as their retransmission arrives before the end of the transfer. This shows the benefits of using the configurable *FECPattern*() operation compared to AC-RLNC's default behaviour, since sending repair symbols regularly still brings benefits for the following studied use-cases.

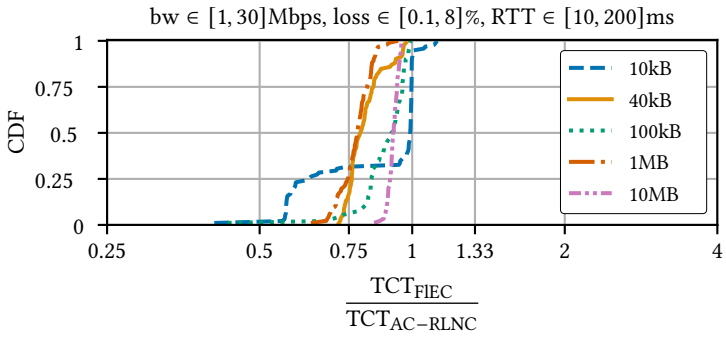


Figure 5.4: TCT ratio between FlEC and AC-RLNC [Coh+20] for regular bulk use-case using the BBR congestion control.

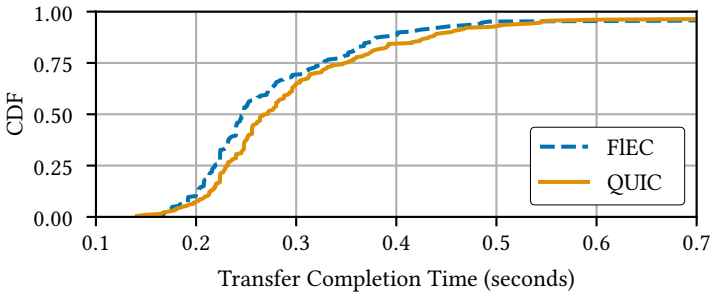


Figure 5.5: TCT comparing FlEC and the regular QUIC for transfers with at least one packet loss, performed on a real Starlink network access.

5.5.2.2 Experimenting with a real network

We now extend our study and analyze the benefits of FlEC over a real network between a regular QUIC and FlEC server on a Ubuntu 18.04 server located at UCLouvain and a client wired to a Starlink access point located in Louvain-la-Neuve (Belgium). We performed a total of 20150 uploads of 50kB from the client to the server. Among those 20150 uploads, 430 encountered at least one packet loss during the transfer. Figure 5.5 shows the CDF of the transfer completion time for these 430 uploads. The median transfer completion time is 247ms for FlEC and 272ms for regular QUIC. The average transfer completion time is 340ms for FlEC and 393ms for QUIC. Unsurprisingly, FlEC improves the transfer completion time for the transfers where the loss events occur during the RTT.

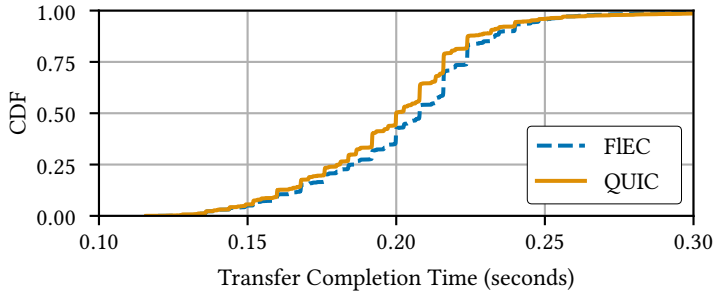


Figure 5.6: TCT comparing FlEC and the regular QUIC for all transfers, performed on a real Starlink network access.

5.5.2.3 CPU performance

While it has been demonstrated that PQUIC protocol plugins deteriorate noticeably the performance [De +19], we analyze the CPU impact of the FlEC framework by first transferring 1GB files on the loopback interface. Without FlEC, we achieved a throughput of 650 Mbps. With FlEC configured for the bulk use-case (i.e. sending repair symbols at the end of the transfer only), it dropped to 300 Mbps. We also analyzed the throughput when sending one repair symbol every ten source symbols and obtained a throughput (i.e. not goodput) of 280 Mbps, meaning that the encoding and decoding of repair symbols implies only a small overhead compared to the framework in itself. This performance drop has an impact on transfers that were not affected by last flight losses. To illustrate this impact, we can look at the transfers we performed using our Starlink access point.

Figure 5.6 shows the CDF of the TCT for all the transfers we performed, with and without losses. We can see that when we consider all transfers, FlEC increases the transfer completion time outside simulations, due to that performance overhead. This is a current limitation of the PQUIC paradigm: the price for this flexibility is a computational overhead due to the protocol plugins system. These observations are inline with earlier discussions on PQUIC performance. We believe that with a native implementation, the impact of the FlEC framework would be barely noticeable. We will see in Chapter 7 how we can get rid of this performance drop using a native and efficient implementation.

5.6 Buffer-limited file transfers

We now discuss the loss recovery mechanism for buffer-limited file transfers. In this setup, the receive window (*rwin*) is relatively small compared to the

congestion window (*cwin*) of the sender, making every loss event potentially blocking and increasing the transfer completion time. In addition to protecting the transfer from tail losses, we protect every window of packets to avoid stalling due to lost packets blocking the stream flow-control window.

5.6.1 Loss recovery mechanism

For this use-case, $ds()$ returns $\hat{l}$ to ensure that *ad* stays larger than *md*, according to the estimated loss rate. $FECPattern()$ behaves as shown in Algorithm 2. We spread the repair symbols along the sent source symbols in order to periodically allow the receiver to unblock its receive window by recovering the lost source symbols and deliver the stream data in-order to the application. More precisely, the $FECPattern()$ operation sends one repair symbol every $\frac{1}{\hat{l}}$ source symbols. The algorithm needs three loss statistics. The first is the estimated uniform loss rate $\hat{l}$. The two others are the $\hat{P}_{GB}$ and $\hat{P}_{BG}$ parameters of the Gilbert loss model. The Gilbert model [Ell63] is a simplification of the two-states Markov model already used in the evaluation of Chapter 2. In the *Good* state of the Gilbert model, all the packets are received while all the packets are dropped in the *Bad* state. The model only contains state-transition parameters. The packet drop probabilities for the Good and Bad state are respectively fixed to 1 and 0. $\hat{P}_{GB}$ is the transition probability from the Good to the Bad state while $\hat{P}_{BG}$ is the transition probability from the Bad to the Good state. In order to estimate the loss statistics $\hat{l}$, $\hat{P}_{GB}$ and $\hat{P}_{BG}$, we implement a *loss monitor* protocol plugin that estimates the loss rate and Gilbert model parameters over a QUIC connection.

When the sender is blocked by the QUIC flow control, $FECPattern()$ sends more repair symbols to recover from the remaining potentially lost source symbols. Spreading the repair symbols along the coding window helps to recover the lost source symbols more rapidly compared to a block approach.

5.6.2 Evaluation

We now evaluate our generic mechanism under a buffer-limited file transfer use-case. We first study a specific network configuration that could benefit from FlEC. We then evaluate its overall performance using experimental design.

5.6.2.1 FlEC for SATCOM

We choose the satellite communications (SATCOM) use-case where the delay can easily reach several hundreds of milliseconds [Tho+19; Kuh+20]. In those cases, end-hosts need a large receive buffer in order to reach the channel

Algorithm 2 FECPattern for buffer-limited use-case

Require: $last$, the ID of the last symbol present in the coding window when $FECPattern()$ was triggered the last time
Require: $nTriggered$, the number of times FECPattern() has already been triggered since no new symbol was added to the window.
Require: $maxTrigger$, the maximum number of times we can trigger this threshold for the same window
Require: $nRSInFlight$, the number of repair symbols currently in flight
Require: W, the current coding window.
Require: $FCBlocked()$, telling us if we are currently blocked by flow control.
Require: $\hat{l}$, $\hat{P}_{GB}$, $\hat{P}_{BG}$, see Table 5.1.

1: **if** $nRSInFlight \geq 2 * \lceil |W| * \hat{l} \rceil$ **then**
2: **return** $false$ {Wait for feedback before sending new RS}
3: **end if**
4: $nUnprotected \leftarrow W.last - last$
5: $n \leftarrow min(\frac{1}{\hat{P}_{GB}}, |W|)$
6: $protect \leftarrow nUnprotected = 0 \vee nUnprotected \geq n \vee FCBlocked()$
7: **if** $protect \wedge nUnprotected \neq 0$ **then**
8: {*Start repair symbols sequence*}
9: $nTriggered \leftarrow 1$
10: $last \leftarrow W.last$
11: $maxTrigger \leftarrow \lceil max(\hat{l} * nUnprotected, \frac{1}{\hat{P}_{BG}}) \rceil$
12: **else if** $protect$ **then**
13: **if** $FCBlocked() \vee nTriggered < maxTrigger$ **then**
14: $nTriggered \leftarrow nTriggered + 1$ {*Continue sending symbols*}
15: **else**
16: $protect \leftarrow false$ {*Enough symbols have been sent*}
17: **end if**
18: **end if**
19: **return** $protect$

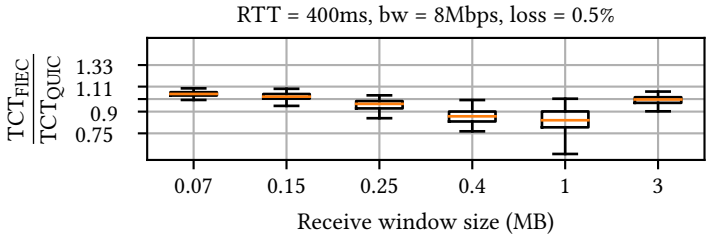


Figure 5.7: TCT ratio, 0.5% losses.

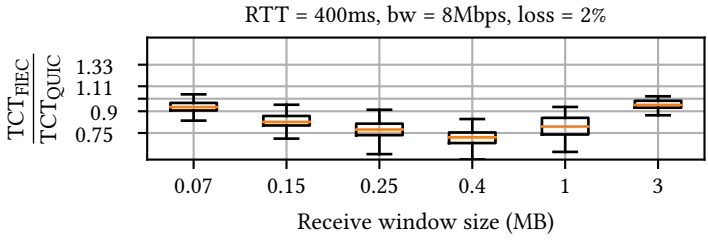


Figure 5.8: TCT ratio, 2% losses.

capacity. If they do not use a sufficiently large buffer, packet losses can have a significant impact on the throughput, preventing the sender to send new data as long as the data at the head of the receive buffer have not been correctly delivered to the application. The studied network configuration has a round-trip-time of 400 milliseconds and a bandwidth of 8 Mbps. Those are lower-bound values compared to current deployments [Kuh+20; Tho+19]. The bandwidth-delay product is thus 400kB. Higher BDP configurations are studied in the experimental design analysis of Section 5.6.2.2. We study the impact of FlEC with several receive window sizes.

5.6.2.1.1 Transfer completion time and throughput. Figure 5.7 shows the transfer completion time ratio between FlEC and regular QUIC with a 5 MB file transfer and 0.5% of packet loss. Each box in the graph is computed from 95 runs with different seeds for the ns-3 rate error model. The bandwidth is set to 8 Mbps and the congestion control is BBR. For each transfer using FlEC, we decrease the receive window by 5% at the receiver in order to store the received repair symbols in the remaining space. With receive windows smaller than the BDP (ranging from 70 kB to 400 kB), the sender is flow-control-blocked once per RTT during a time proportional to the $\frac{rwin}{cwin}$ ratio. This implies that the transfer completion time with small receive windows is large even without any packet loss. Furthermore, the earlier the loss occurs during the round-trip, the longer the sender will be blocked by the flow control for the next round-trip,

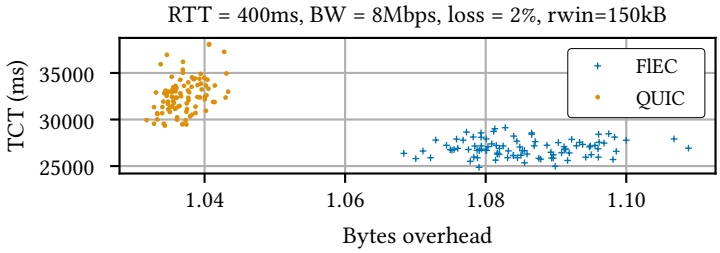


Figure 5.9: Time-bandwidth tradeoff, 2% losses.

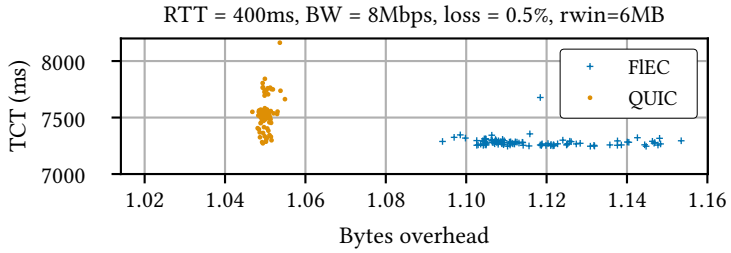


Figure 5.10: Time-bandwidth tradeoff, 0.5% losses.

since it needs to retransmit the data to unblock the receive window. Sending *a priori* repair symbols for these configurations allows reducing or completely avoiding those blocking situations, at the price of a small reduction in goodput. For the 70 kB receive window, the 5% reduction to store the repair symbols is significant compared to the benefits of FEC and has a negative impact on the goodput. With the 400 kB receive window and above, the sender only blocks in the presence of losses during the round-trip. For the 3MB receive window, sending repair symbols *a priori* does not unblock the window but still helps to recover from tail losses. With such a high RTT, the impact of a tail loss relative to the transfer completion time is still significant.

Figure 5.8 shows the results of our experiments with a 2% packet loss rate. It is thus more common that the sender becomes flow-control blocked due to the increased loss rate. This makes the approach worth even for smaller receive window sizes such as 70kB as the sender will be slowed down a lot more often.

5.6.2.1.2 Delay-bandwidth tradeoff. The graph on Figure 5.9 illustrates the delay-bandwidth tradeoff operated when using FlEC instead of regular QUIC. Each point on the Figure concerns a single experiment and represents the transfer completion time and the bytes overhead of the solution. The bytes overhead is computed by dividing the total amount of bytes of UDP payload

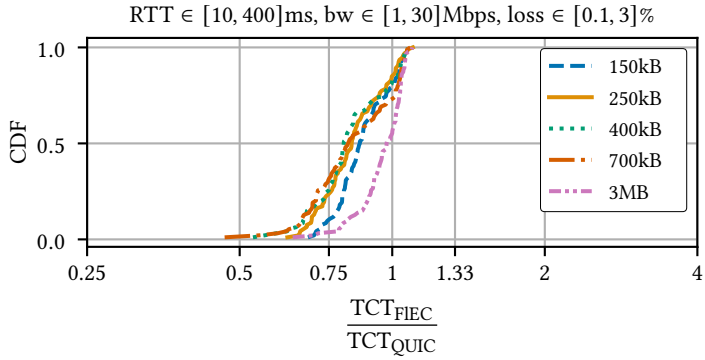


Figure 5.11: Experimental design analysis for several receive window configurations.

sent by the server by the size of the file transferred (5MB). For this graph, the experiments use a small receive window of 150kB and the loss rate is 2%. As the receive window is small, sending FEC unblocks the receive window upon losses and allows drastically lowering the transfer completion time. The price to pay is an increased bytes overhead compared to the regular QUIC solution. In this rwin-limited scenario, the available bandwidth is generally larger than what is used due to the rwin restriction.

Figure 5.10 shows experimental results with the opposite scenario: the receive window is 6MB large, which is larger than both the file to transfer and the bandwidth-delay product of the link. This case is similar to the bulk use-case of section 5.5. We can see that FlEC leads to stable latency results at the expense of a larger bytes overhead than QUIC. As the receive window is larger than the file to transfer, the sender will never be flow-control blocked during the transfer. In this case, FlEC minimizes the latency essentially by recovering from tail losses.

5.6.2.2 Experimental design analysis

Figure 5.11 shows the aggregated results of simulations using experimental design. We show the CDF of the transfer completion time ratio between FlEC and QUIC. Each CDF on the Figure is built from 95 experiments with parameters selected from the ranges depicted on the top of Figure 5.11. Each CDF curve corresponds to transfers using the receive window size specified in the legend. The congestion control used is still BBR. We observe positive results using FlEC for the majority (75%) of the network configurations, especially for smaller receive window sizes (80% positive results for windows smaller or equal to 400kB). Some configurations still expose negative results using FlEC,

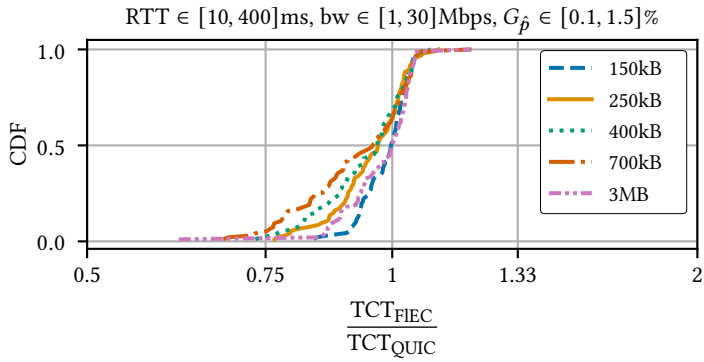


Figure 5.12: Experimental design analysis using Gilbert model with bursts of 1 to 5 packets.

even for smaller receive window sizes. These configurations are those whose bandwidth-delay product is small compared to the receive window. To verify this, we computed the average $\frac{BDP}{rwin}$ ratio on all the experiments for which FlEC took more time to complete than `picoquic`, whose value is 0.48. For the experiments where the FlEC transfer was faster, the average value of this ratio is 1.53.

Let us now assess the performance of our solution using a bursty loss model in order to see whether FlEC remains robust even in the presence of loss bursts. Figure 5.12 shows the results of an experimental design analysis with a Gilbert loss model with $\hat{P}_{GB}$ ranging from 0.1% to 1.5% and $\hat{P}_{BG}$ set to 33% (i.e. an expected burst size of 3 packets) and a maximum burst size of 5 packets. Loss events thus occur less often compared to Figure 5.11, leading to fewer blocking periods for QUIC during the experiments but with a higher probability of loosing several packets in a row. We can see that Algorithm 2 still offers benefits in the presence of bursty losses. Similarly to Figure 5.11, FlEC especially improves the results for experiments with a large $\frac{cwin}{rwin}$ ratio. We also observe a higher variance in the results due to bursty loss events being more difficult to repair on average than isolated losses.

5.7 Delay-constrained messaging

In this section, we present the implementation and evaluation of FlEC tailored for delay-constrained messaging. The goal is to protect entire messages instead of naively interleaving repair and source symbols. Using application knowledge, FlEC protects as much frames as possible at once.

5.7.1 Reliability mechanism

We consider an application sending variable-sized messages, each having its own delivery deadline. To convey these deadlines, we extend the transport API using protocol plugins (Section 5.7.1.1). Furthermore, we replace the QUIC stream scheduler to leverage application information (Section 5.7.1.2). We then discuss and evaluate FlEC under this scenario in Section 5.7.2.

5.7.1.1 Application-specific API

We propose the following API enabling an application to send deadline-constrained messages.

5.7.1.1.1 SEND_FEC_PROTECTED_MSG(MSG, DEADLINE) This function allows the application to submit its deadline-constrained messages. The QUIC protocol already supports the stream abstraction as an elastic message service. However, the stream priority mechanism proposed by QUIC, while being dynamic, is not sufficient to support message deadlines. The protocol operation attached to this function inserts the bytes submitted by the application in a new QUIC stream, closes the stream, and attaches the application-defined delivery deadline to it. The message must be delivered at the receiver within this amount of time to be considered useful. If the network conditions prevent an on-time delivery of the message, the message may be cancelled, possibly before being sent, and the underlying stream be reset.

5.7.1.1.2 NEXT_MESSAGE_ARRIVAL(TIME) This function allows the application to indicate when it plans to submit the next message. While this API function is not useful for all kinds of unreliable messaging applications, applications having a constant message sending rate such as video transfers might benefit from providing such information.

5.7.1.2 Application-tailored stream scheduler

The knowledge provided by the application to the transport layer is not only useful for redundancy scheduling. It is also valuable for the QUIC stream scheduler. Without this information, the PQUIC stream scheduler schedules high priority streams first and has two different ways to handle the scheduling of streams with the same priority: *i*) round-robin or *ii*) FIFO. We let the application define its own scheduler to schedule its streams more accurately. Algorithm 3 describes our QUIC stream scheduler for deadline-constrained messaging applications.

The *closestDeadlineStream*() function searches among all the available streams attached to a deadline to find the stream having the closest expiration

Algorithm 3 Application-tailored scheduler for delay-constrained messaging.

Require: $\mathcal{S}$, the set of available QUIC streams
Require: $\hat{OWD}$, the estimated one-way delay of the connection
Require: *now*, the timestamp representing the current time
Require: $FCBlocked(stream)$, telling if the specified stream is flow control-blocked.
Require: $closestDeadlineStream(\mathcal{S}, deadline)$, returning the non-expired stream with the closest delivery deadline to the specified deadline

$scheduledStream \leftarrow \emptyset$
$currentDeadline \leftarrow now + \hat{OWD}$ {Initialization}
while $scheduledStream = \emptyset$ **do**
  $candidate \leftarrow closestDeadlineStream(\mathcal{S}, currentDeadline)$
  if $candidate = \emptyset$ **then**
    break
  end if
  if $\neg FCBlocked(candidate)$ **then**
    $scheduledStream \leftarrow candidate$
  else
    $\mathcal{S} \leftarrow \mathcal{S} \setminus \{candidate\}$
    $currentDeadline \leftarrow candidate.deadline$
  end if
end while
if $scheduledStream = \emptyset$ **then**
  return $defaultStreamScheduling(\mathcal{S})$ {Fallback}
end if

deadline while still having chances to arrive on-time given the current one-way delay. The scheduler chooses the non-flow-control blocked stream that is the closest to expire while still having a chance to be delivered on-time to the destination. Our implementation estimates the one-way delay as $\frac{RTT}{2}$. Other methods exist [FHK18; Hui22b]. Recent versions of `picoquic` include a mechanism for estimating the one-way delay [Hui22a] when the hosts clocks are synchronized. In the absence of clock synchronization, the estimated one-way delay can only be interpreted relatively, which helps to estimate the one-way delay variation but not for decision thresholds such as the one used in Algorithm 3.

5.7.1.3 FECPattern() and ds() for delay-constrained messaging

We now describe how our application redefines the two FlEC anchors. Our application is sensitive to the delivery delay of entire messages more than the in-order delivery delay of individual source symbols. We set the $ds()$ threshold to $-\hat{l}$ as it is useful to retransmit lost symbols that can still arrive on-time. $FECPattern()$ is described in Algorithm 4. The algorithm triggers the sending of repair symbols to protect as many messages as possible according to the messages deadline and the next expected message timestamp if provided by the application. The rationale is the following. If the unprotected messages can wait for new messages to arrive before being protected, $FECPattern()$ does not send repair symbols and waits for the arrival of new messages. Otherwise, repair symbols are sent to protect the entire window until it is considered fully protected. This idea of waiting for new messages before protecting comes from the fact that the messages can be small and sending repair symbols for each sent message can lead to a high overhead. By doing so, $FECPattern()$ adapts the code rate according to the application needs.

5.7.2 Evaluation

We evaluate FlEC under the messaging use-case using an application sending video frames as messages. We set the deadline to 250 milliseconds, meaning that each frame must be delivered within this time. We use 86 seconds of the video recording from the Tixeo video-conference application [22j]. The frame and bit rates are adjusted by the application. This video recording starts at 15 frames per second during the first 6 seconds and runs at 30 images per second afterwards. For each frame, we record its delivery delay between when the application sends it and when it is delivered at the receiver. We send each video frame in a different QUIC stream to avoid head-of-line blocking across frames upon packet losses. The regular QUIC solution uses the default round-robin scheduler provided by PQUIC. In the first set of experiments, we set the bandwidth to 8 Mbps and observe the performance of FlEC in the

Algorithm 4 *FECPattern*() for delay-constrained messaging.

Require: $\mathcal{S}$, the set of available QUIC streams
Require: $\hat{OWD}$, the estimated one-way delay of the connection
Require: now, the timestamp representing the current time
Require: $closestDL(\mathcal{S}, deadline)$, returning the message deadline that will expire the sooner from the specified deadline
Require: $last$, the last protected message.
Require: $nTriggered$, the number of times FECPattern has already been triggered since no symbol was added to the window.
Require: $maxTrigger$, the maximum number of times we can trigger this threshold for the same window
Require: $nextMsg$ (is $+\infty$ if the message API is not plugged), the maximum amount of time to wait before a new message arrives.
Require: $cwin$, bif, the congestion window and bytes in flight.
Require: θ space to save in cwin for directly upcoming messages.
Require: $\hat{l}$, $\hat{P}_{BG}$, see Table 5.1.

$nextDL \leftarrow closestDL(\mathcal{S}, max(now + \hat{OWD}, last.deadline))$
$protect \leftarrow (nextDL = \emptyset \vee now + \hat{OWD} + nextMsg + \epsilon \geq nextDL)$
$nUnprotected \leftarrow W.last - last$
if $protect \wedge nUnprotected \neq 0$ **then**
  {*Start repair symbols sequence*}
  $nTriggered \leftarrow 1$
  $last \leftarrow W.last$
  $maxTrigger \leftarrow \lceil max(\hat{l} * nUnprotected, \frac{1}{\hat{P}_{BG}}) \rceil$
else if $protect$ **then**
  if $nTriggered < maxTrigger$ **then**
    $nTriggered \leftarrow nTriggered + 1$ {*Continue sending*}
  else
    $protect \leftarrow false$ {*Enough symbols have been sent*}
  end if
end if
return $appLimited() \wedge protect \wedge \frac{cwin}{bif} > \theta$

presence of losses. For each experiment, the one-way delay is sampled in the $[50, 200]ms$ range. We next perform an experimental design analysis over a wider parameters space.

Figures 5.13 and 5.14 show the 95^{th} and 98^{th} percentiles of the message delivery times for each experiment. We can see that while 95% of the video frames are delivered successfully in every experiment, regular QUIC struggles to deliver 98% of the submitted frames on time (i.e. before 250 milliseconds) with a one-way delay above 75 milliseconds. Indeed, with a one-way delay above 75 milliseconds, the lost frames are retransmitted after more than 150 milliseconds and take more than 75 milliseconds to reach the receiver. Note that QUIC's loss detection mechanism takes a bit more than one RTT to consider a packet as lost using the time-based loss detection threshold [RFC9002]. These retransmitted frames thus arrive a few milliseconds before the deadline in the best case. As we can see on Figure 5.14, only a few experiments without FlEC have more than 98% of the frames arriving on-time while FlEC can cope with one-way delays up to 200 milliseconds without problem. Figure 5.15 shows that no experiment with regular QUIC succeeded to deliver 99% of the video frames on time with a one-way delay above 75ms, while FlEC succeeded in every performed experiment. Note that the $FECPattern()$ algorithm plugged in this scenario tries to protect as many messages as possible with the same number of repair symbols by delaying the sending of repair symbols when new messages are expected soon. This lazy repair symbol scheduling explains why the frame delivery time is larger than the one-way delay. In order to send as few repair symbols as possible, FlEC delays the sending of repair symbols to the last possible moment while ensuring that lost data can be recovered before the deadline.

Figure 5.16 shows the ratio between the number of messages received on-time by FlEC ($FlEC_{API}$) and by the regular QUIC implementation. In order to isolate the effects of the FlEC API, the Figure also shows FlEC results without leveraging the application knowledge brought by the API functions ($FlEC_{NO-API}$). It thus uses PQUIC's default stream scheduler and sends repair symbols for each newly sent message. As it is only a simplified version of Algorithm 4, we do not show the $FECPattern()$ algorithm of this second solution. As we can see, nearly every experiment ended with more messages received on-time using the API-enabled $FlEC_{API}$ compared to QUIC. A similar gain compared to the regular QUIC is present for both $FlEC_{API}$ and $FlEC_{No-API}$. However, the interest of the application-defined API resides in the redundancy it needs to obtain those results.

We now analyze the redundancy overhead of our solution. Figure 5.17 shows the ratio of bytes sent by the server between regular QUIC and FlEC with and without the API defined in Section 5.7.1.1. The results of $FlEC_{No-API}$ show that protecting every message blindly is very costly in terms of bandwidth.

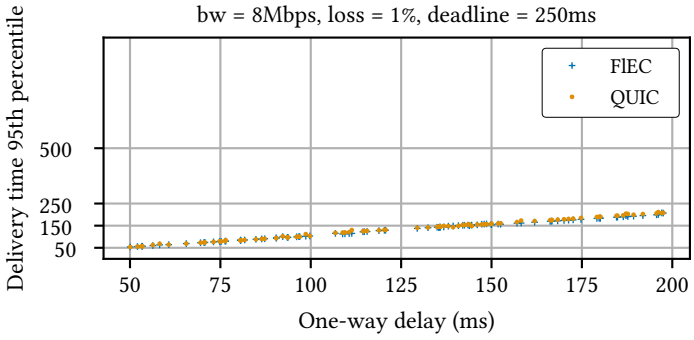


Figure 5.13: Message delivery time 95th percentile, comparing FlEC using the application-defined API and the regular QUIC.

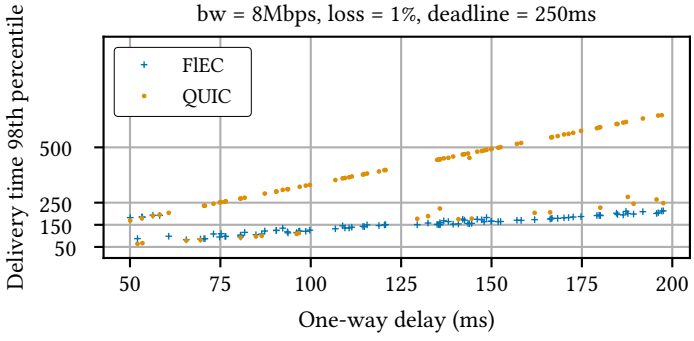


Figure 5.14: Message delivery time 98th percentile, comparing FlEC using the application-defined API and the regular QUIC.

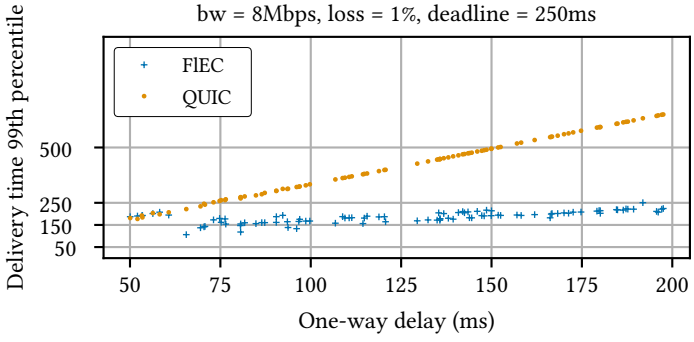


Figure 5.15: Message delivery time 99th percentile, comparing FlEC using the application-defined API and the regular QUIC.

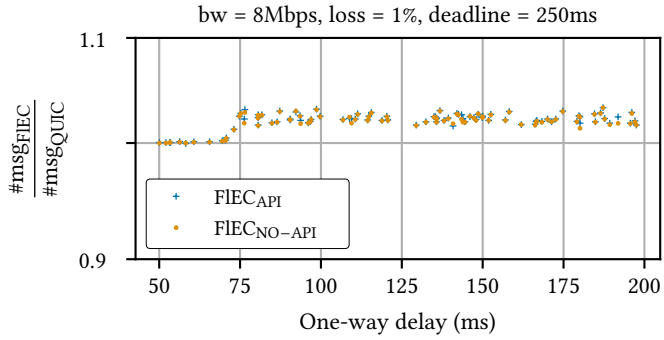


Figure 5.16: Messages received on-time comparing QUIC and FlEC with and without the application-defined API.

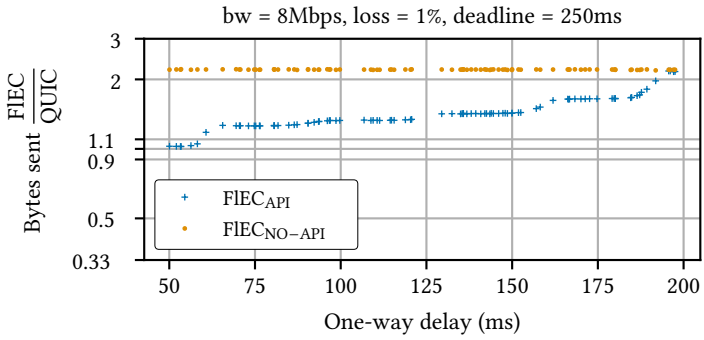


Figure 5.17: Bytes sent by the server, comparing QUIC and FlEC with and without API, using BBR.

Indeed, for this video-conferencing transfer, many video frames sent by the application are smaller than the size of a full QUIC packet. The REPAIR frames sent by FlEC contain additional metadata. In this case where the application traffic is thin, protecting every message may double the volume of sent data as shown for $FlEC_{No-API}$ on Figure 5.17. Using FlEC with the message-based API can spare bandwidth significantly by using application-aware stream and redundancy schedulers. Note the portion of the graph between 5ms and 70ms one-way delays. For those configurations, no repair symbol is sent by $FlEC_{API}$. Indeed, the messages are acknowledged by the peer before $FECPattern()$ triggers the sending of repair symbols. FlEC thus naturally uses SR-ARQ when redundancy is not needed to meet the message deadlines. The amount of redundancy sent then gradually increases along with the one-way delay. Indeed, when the delay increases, the number of messages that can be protected at the same time by the $FECPattern()$ algorithm decreases.

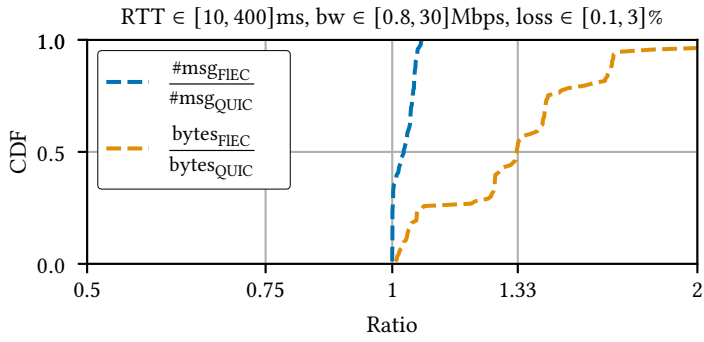


Figure 5.18: Experimental design analysis for the delay-constrained messaging use-case using BBR.

Experimental design analysis

Figure 5.18 shows the results of an experimental design analysis using the parameters depicted on the top of the Figure. The Figure shows CDFs for the amount of bytes sent by the server and the number of messages received within the deadline. In a few cases, the FlEC solution using the application-tailored API sends a similar amount of bytes to regular QUIC. This is due to the fact that for some configurations, the delay was sufficiently low to send no or only a few repair symbols. We can also see that none of the experiments revealed a lower amount of on-time received messages compared to regular QUIC, showing the robustness of FlEC under various network conditions.

Improvements

Other information from the application could have been taken in addition to the message deadlines. For example, information concerning the video frames type could have an impact on the stream scheduling: H264 I-frames are more important than P ones as the latter depend on the first to be decoded. The stream scheduling could even be further improved by looking at the dependence between each frames in a group of H264 frames. Given the flexibility of FlEC, the messaging API can be extended for the application to transfer this kind of knowledge to the transport stack.

5.8 Conclusion

In this chapter, we redefined the QUIC loss recovery mechanism and enabled a per-use-case customization. Flexible Erasure Correction (FlEC) allows efficiently combining retransmissions and Forward Erasure Correction. Appli-

cations can either use a standard Selective-Repeat ARQ mechanism or tailor a Forward Erasure Correction mechanism that fits their own traffic pattern and sensitivity to delays. Our FlEC implementation leverages the PQUIC protocol plugins to enable the application to insert its own algorithms to select the level of redundancy and perform the stream scheduling decisions. We customized FlEC for three different use-cases. We evaluated and demonstrated that FlEC can be configured with small to no effort by applications to significantly enhance the quality of experience compared to the existing QUIC loss recovery mechanisms in simulations. Yet, we still have to overcome a few limitations of our approach. First, the computational overhead of the PQUIC framework is important and we saw in Section 5.5.2.2 that it can lead to bad results on real networks when loss events are rare. Second, the scenarios explored until now were generic and mostly evaluated in a simulated environment. Finally, we did not evaluate yet the performance improvements of our loss recovery extension on real applications.

Chapter 6 explores in more details the loss characteristics of the Starlink network we installed at UCLouvain to see how a FEC-enabled loss recovery mechanism can bring benefits on real networks. The findings of Chapters 5 and 6 are then leveraged in Chapter 7 where an efficient loss-recovery mechanism inspired by FlEC is implemented to improve real applications on actual networks without the computational overhead of PQUIC and FlEC.

Starlink: analyzing a new access network

6

The evaluation parts of the previous chapters were focusing on either emulated or simulated environments. In the last chapter, we saw that FlEC could bring significant benefits in different scenarios by adapting the loss recovery to the application. However, we observed that FlEC could cause a performance degradation when applied outside simulations for the bulk scenario. In order to experiment with real network accesses where FEC can bring latency improvements, we purchased a standard Starlink access and installed it on the roof of our laboratory building in Louvain-la-Neuve. In this chapter, we analyze the different characteristics of this network access. This analysis includes throughput and delay measurements, but also provides a detailed study of packet losses, observing the rate and patterns of loss events as well as their duration. This study shows that QUIC applications may encounter numerous loss events when using a Starlink network access and may benefit from a FEC-enabled loss recovery mechanism.

This work was done in collaboration with Martino Trevisan (University of Trieste) and Danilo Giordano (Politechnico di Torino). It has been published and presented at the IMC '22 conference [Mic+22]. This chapter reflects the state of our Starlink access by the time of writing that article, although the results are still relevant today.

6.1 A new wireless access network

Internet access technologies and Internet protocols are constantly evolving. Broadband technologies such as xDSL and cable modems are prevalent today, but they are being replaced by optical fibers. In densely populated areas such as cities, fiber deployment can be profitable, while it can be much more expensive in rural or mountainous areas. For this reason, network operators have been working on other Internet access technologies for considerable time. Some propose Fixed Wireless Access (FWA) technologies [ATT; CM19]. Others are deploying hybrid networks that combine cellular and xDSL [KHB20; For16]. Given the opportunities offered by these rural areas, several companies nowadays offer satellite-based Internet access solutions.

Classical Satellite Communications (SatCom) use geostationary satellites at

an orbit of 22 236 miles. A single satellite can cover a large portion of the Earth at the price of a latency of several hundreds of milliseconds due to their high elevation [CGK10; Kod+20]. Such a communication technology may provide connectivity to thousands of customers with connections easily reaching up to 100Mbps, with the drawback of a latency above 500 ms [Per+22]. These geostationnary satellite services are the ones providing the Mobile Satellite Service (MSS) for the In-Flight Communications (IFC) scenario explored in Chapters 2 and 4.

A new approach is to use a constellation of Low Earth Orbit (LEO) satellites to dramatically reduce communications latency. The first large-scale deployment of this kind is the Starlink constellation, currently operating more than two thousand satellites. The commercial service started in beta version in October 2020 in the United States and from 2021 in European countries. It promises Internet access with latency on the order of 20 ms and bandwidth between 100 and 200 Mbps [22i]. Being a newborn service, its operation and performance have not been fully investigated yet. The only comparable work has been proposed by Kassem *et al.* [M K+22], which shows how Starlink performance changes from different vantage points. We here focus on how the network characteristics of a single Starlink vantage point changes when accessing globally distributed resources, under high and heavy network loads, with the TCP and QUIC transport protocols.

For many years, TCP has been the dominant protocol for Internet services [TMW97; LCB10]. SatCom operators therefore widely adopt TCP Performance Enhancing Proxies [RFC3135] (PEP) to mitigate the impact of increased latency on TCP performance. In contrast with TCP, QUIC cannot be optimized by using PEPs in satellite networks since QUIC packets are encrypted and authenticated. Given the current growth of QUIC traffic, it is important to evaluate new access networks using both QUIC and TCP.

In the next sections, we benchmark the Starlink service and compare it to traditional SatCom networks. We measure the performance in terms of throughput for QUIC and TCP, latency, and packet loss, and find that Starlink delivers its performance promises and enables the use of demanding services such as high-definition video streaming or cloud gaming. We also find that Quality of Experience (QoE) for Web browsing with Starlink is far better than with traditional SatCom and comes close to wired access.

6.2 Testbed and Measurements

For our experimental campaign, we use three off-the-shelf PCs equipped with 8 cores and 16 GB of memory running Ubuntu 20.04 and Linux kernel version 5.0.4. The first two PCs are located in the UCLouvain campus in Louvain-la-Neuve, Belgium. The first PC (PC-Starlink) is connected to the Internet via

Starlink with a regular subscription. The second PC (PC-Wired) is connected to the UCLouvain campus network via a 1 Gbit/s Ethernet adapter. The third PC (PC-SatCom) is connected to the Internet via a traditional SatCom equipment for which we have purchased a regular plan offering up to 100 Mbit/s in downlink and 10 Mbit/s in uplink. The SatCom operator is a reseller and relies on a major European provider that uses geostationary satellites to provide Internet access. Our user equipment consists of a dish antenna and a modem that connects the PC to the network. For each setup, the TCP receive window is the kernel default, i.e. 131072 bytes with a maximum of 6291456 bytes through automatic buffer tuning. The congestion control is Cubic. We use the three PCs to run the experiments that we describe in detail below and summarize in Table 6.1.

QUIC measurements. Some of the performance metrics of this article are gathered using QUIC. We assess the network performance with two kinds of transfers: *(i) bulk* HTTP/3 (H3) [RFC9114] 100MB transfers and *(ii) light* QUIC transfers with regularly sent messages, similar to a real-time video traffic. The latter sends 25 variable length messages per second during 2 minutes. Each message has a size sampled randomly in the 5-25kB range. The average bitrate of this transfer is 3 Mbit/s, far below both downlink and uplink capacities announced by Starlink. The QUIC client runs on PC-Starlink while the server is located in the UCLouvain university campus. Half of the experiments are transfers from the server to the client (download) and the other half are from the client to the server (upload). Using QUIC instead of TCP ensures that we measure the end-to-end latency as it forbids the use of middleboxes and proxies interfering with the traffic at the transport layer as it can be done for TCP with PEPs. The way QUIC identifies and retransmits packets also allows us to exactly point every lost packet and disambiguating original packets from retransmissions. The QUIC H3 server is able to provide more than 400Mbps of QUIC traffic to other servers outside our campus. The QUIC implementation used is `quiche` [Clo22] compiled in *release* mode from commit `ba87786`. Its initial `max_data` and `max_stream_data` transport parameters are set to 10MB and the receive window varies through automatic buffer tuning. The congestion control used is Cubic.

Latency. We measure the latency of Starlink by probing a set of 11 anchors using `ping`. Our set of anchors includes 7 servers used inside the RIPE Atlas project [22e]. The servers are located in Europe (Amsterdam ×2, Nuremberg ×2), North America (New York, Fremont) and Asia (Singapore). We also include 4 nodes of the RIPE Atlas project hosted by volunteers in the same country as our Starlink connection (Belgium). Every five minutes, we measure the latency towards the anchors running 3 pings. We also measure the link latency under light and heavy network load by studying the evolution of the Round-Trip Time (RTT) measured by QUIC with our messages and H3 transfers.

Table 6.1: Overview of the datasets.

Measure	Network	Duration	Target
Latency	Starlink	5 Months	11 Anchors
Throughput	Starlink	4 Months	Ookla Servers
	SatCom	2 Weeks	
Web Browsing	Starlink	4 Months	120 Websites
	SatCom	2 Weeks	
QUIC H3	Starlink	5 Months	Our server
QUIC messages	Starlink	5 Months	Our server

Packet loss. Starlink provides a new kind of wireless network access. In general, packet losses come from two causes: congestion or medium imperfection (e.g., electromagnetic interferences). We study the packet losses under light and heavy network load using our QUIC setup with both bulk H3 transfers and messages variants.

Throughput. We measure Starlink download and upload throughputs using the command-line version of the Ookla SpeedTest service [22h]. The application selects the closest test server and probes download and upload capacities by opening several parallel TCP connections. We perform a speed test every half hour using PC-Wired from December 20 2021 to April 7 2022. We compare Starlink with SatCom using PC-SatCom, on which we run identical measurements, scheduling them at the same pace. Finally, we also measure the Starlink throughput using our QUIC H3 setup.

Web Browsing. We measure the performance of Starlink for Web browsing by running on PC-Starlink automatic visits to websites and collect metrics that can be used as proxy for users' perceived QoE. We rely on BrowserTime, a tool performing automated visits to websites [22a]. We rely on the rank provided by SimilarWeb [22k], an online ranking service out of which we pick the top-120 websites for Belgium. Among the statistics collected with BrowserTime, we focus on two metrics that have been shown to be correlated with users' QoE [Hor+18]: *(i)* ***onLoad***: the time when the browser fires the onLoad event – i.e., when all elements of the page have been downloaded and parsed; *(ii)* ***SpeedIndex***: proposed by Google [22g], it represents the time at which the visible parts of the page are displayed. It is computed by recording the video of the browser screen and tracking the visual progress of the page during rendering. Every half an hour, we test 30 websites chosen at random and ensure they do not overlap with the speed test experiments. We collect data from December 20 2021 to April 7 2022. We compare the browsing experience offered by Starlink with the SatCom link by running the same experiments on PC-SatCom and collect the resulting metrics.

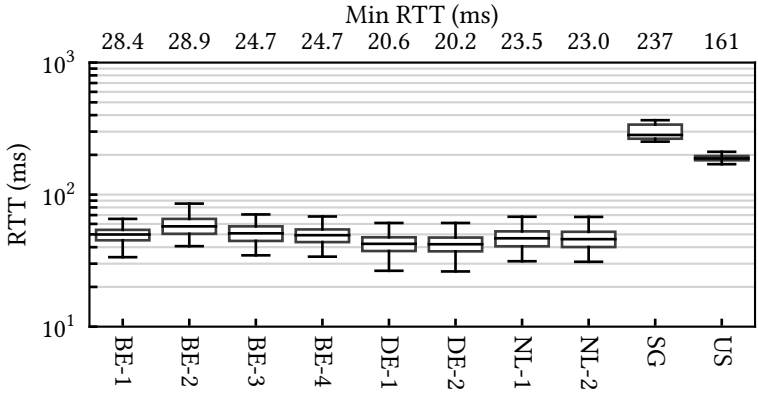


Figure 6.1: Distribution of the RTT to the anchors. The top x axis reports the distribution minimum. Notice the logarithmic y axis.

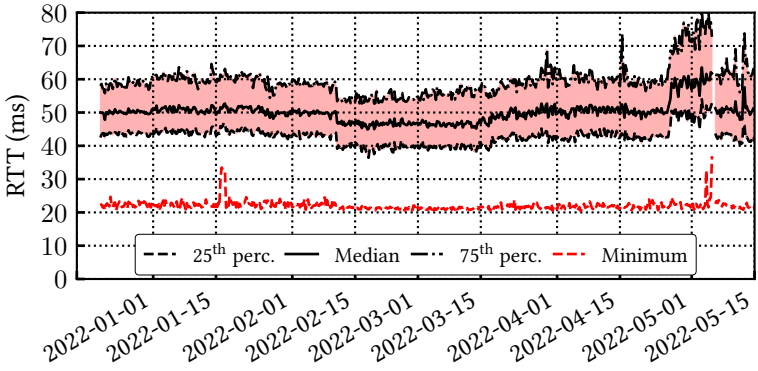


Figure 6.2: RTT towards the European anchors.

6.3 Results

In this section, we report our results and findings. We first discuss the measured latency and then focus on packet loss and throughput, comparing Starlink with traditional SatCom. Finally, we discuss QoE-related metrics for web browsing and the presence of middleboxes.

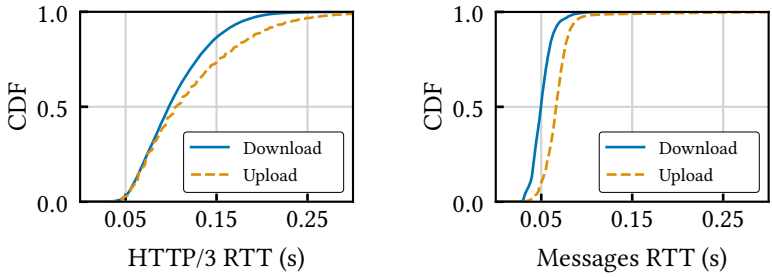
6.3.1 Latency

We begin our analysis by looking at the RTT. We first measure the latency without load on the link, which is the best latency Starlink subscribers could achieve. We then perform QUIC downloads and uploads, thus generating bandwidth pressure and study how the RTT evolves under load.

6.3.1.1 Latency during inactivity

Figure 6.1 shows the distribution of the measured latencies towards our set of anchors. The y-axis (in logarithmic scale) represents the distribution of RTT measured by `ping` in the form of a boxplot: boxes range from the 25^{th} to the 75^{th} percentile, while whiskers range from the 5^{th} to the 95^{th}. The black central stroke represents the median, while on the upper x axis we indicate the absolute minimum of the distribution. The 4 left-most boxes are the four local anchors. In the median case, the RTT is in $[46, 52]$ms and exceeds 70ms in less than 5% of cases. The minimum observed RTT for these anchors is $[24, 28]$ms. Similar considerations apply to the two Dutch anchors. The lowest RTT we observe is for the two German probes, which PC-Starlink reaches in only 42ms in median. The lowest RTT we observe is 20.2ms, confirming Starlink's 20ms latency promise. We observe that these values allow high QoE for voice calls [ITU03] and are compatible with latency-sensitive services such as cloud gaming. Indeed, GeForce Now, one of the leading platforms, mandates a latency below 80ms [22d]. To reach the most distant anchor points in the U.S. (San Francisco) and Asia (Singapore), the RTT is necessarily much higher, but not more than the distance between the endpoints would suggest. San Francisco and Singapore are reached in a median of 184 and 270ms respectively. Using traceroute, we verified the path taken by packets towards San Francisco and Singapour and the exit nodes from the Starlink network were the same as for the European anchors (i.e. one exit in the Netherlands and the other in Germany). This suggests that inter-satellite links (ISL) were not enabled by the time, although ISL-capable satellites were already launched [Twi22a] and ISL activation was planned by the end of 2022 [Twi22b]. As of writing this thesis, traceroutes towards american anchors still exit the Starlink network through European service providers and do not seem to leverage inter-satellite links.

To investigate how latency evolves over time, we depict in Figure 6.2 various percentiles and the minimum values, focusing on European anchors. The x-axis spans the five months of measurements, and we compute our statistics using 6-hours bins. The picture is fairly flat, indicating stable performance and no particular changes in Starlink infrastructure over this period. The RTT to the European anchors remains constant around 50 ms in median and ranges from 40 ms (25^{th} percentile) to 60ms (75^{th}). The minimum measured latency is on the order of 20 ms. Interestingly, we observe that the distribution takes on slightly smaller values of a few milliseconds from February 11 onwards - see the small step in the middle of the figure. We suspect that this improvement is related to new satellites joining the constellation in early 2022, although we have no direct evidence [22f]. Moreover, we observe an increase in RTT during the last week of April and the first week of May. Since, at that time, we did not



(a) H3 bulk traffic. **(b) Messages traffic.**

Figure 6.3: Measured per-packet RTT distribution.

run other experiments concurrently, we speculate that in this period Starlink was more loaded or going through reorganization, but we cannot confirm this. Finally, we observe that distribution of RTT is rather flat over the hours of the day. The median RTT is around 50 ms and a Mood's test suggests the samples are drawn from distributions with the same median. Similar considerations hold for throughput measurements as well, and this can hint low utilization of the infrastructure as it does not seem impacted by diurnal patterns.

6.3.1.2 Latency under load

We now study the latency evolution under link pressure. We perform HTTP/3 downloads and uploads towards our server and study the evolution of the RTT during the file transfer. Figure 6.3 shows the distribution of the RTT for every acknowledged packet during the experiments. We compute the downloads curve by running an additional one-week experiments session with packets captures on the server as they were too few RTTs samples coming from the client capture for download transfers. Each curve contains more than 2 millions RTT samples. We note a median, 95^{th} and 99^{th} percentiles RTT of 95 (resp. 104), 175 (resp 237) and 210 (resp 310) ms for downloads (resp. uploads). We can see that the RTT increases more for uploads than download. This difference may be explained by the larger available bandwidth for downloads allowing emptying the router queues faster than for uploads, having thus a smaller impact on queuing delay for equally-sized queues.

We finally study the RTT evolution with the QUIC message transfers. Compared to the H3 traffic, the RTT stays mostly under 100ms, similarly to the values we obtain for pinging European anchors. The downloads (resp. uploads) have 50 (resp. 66) ms median RTT, 71 (resp 87) ms 95^{th} percentile and 87 (resp. 143) ms 99^{th} percentile RTT. The larger RTT for uploads relates to quiche not implementing packet pacing for their default HTTP/3 client.

The largest messages (25 kB) are thus stacked in the network's buffers making the RTT increase lightly. To sum up these latency measurements, we observe that the minimum latency of Starlink is on the order of 20 ms for nearby destinations, as publicly advertised. Under traffic load, it may increase to a few hundreds of milliseconds, probably due to bufferbloat on the access link.

6.3.2 Characterizing packet losses

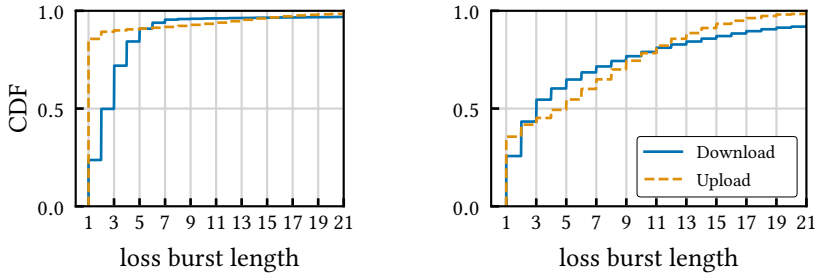
Packet losses can be caused by congestion or imperfections on the medium. For downloads, we determine losses by looking at QUIC received packet numbers on the client. As in QUIC retransmitted data have different packet numbers from the original data and as the quiche implementation does not introduce packet number gaps, every missing packet number means the packet has been lost. For uploads, we determine the received packets by looking at the ACK frames returned by the server.

6.3.2.1 Packet losses during HTTP/3 transfers

We first study the packet losses encountered during HTTP/3 bulk transfers. In this case, losses can be due to *both* congestion and medium imperfections. Note that the UDP buffers of each QUIC endpoint were configured to be large enough to prevent any packet loss due to UDP buffers being filled. The first two columns of Table 6.2 show the packet loss rates recorded during the H3 transfers. We can see that uploads suffered from more loss events than downloads. Nearly 2% of the packets were lost during uploads while a bit more than 1.5% were lost during downloads. Figure 6.4a shows the measured distribution of the loss burst lengths during H3 transfers. The loss burst length is the number of consecutively lost packets for each loss event. As we can see, the majority of the loss events during uploads concerned only one packet at a time, while more than 75% of loss events during downloads concerned several consecutive packets. We also look at the duration of a loss event. Indeed, some wireless technologies such as 802.11 implement retransmission mechanisms that may delay the arrival of subsequent packets, resulting in small silent periods during the transfer [MB21a]. As packets are captured on the client, we can compute the duration of loss events during downloads. We identified 244 008 loss events. The median loss event duration is 49 microseconds. The 75^{th} and 90^{th} percentiles are respectively 58 and 113 microseconds. The 95^{th} and 99^{th} percentiles are 1.5 and 7.5 milliseconds. We also identified a small number of longer loss periods lasting more than 1 second identifying a possible loss of connectivity.

Table 6.2: QUIC packet loss ratios

H3 ↓	H3 ↑	Messages ↓	Messages ↑
1.56%	1.96%	0.40%	0.45%



(a) H3 transfers **(b) Messaging transfers**

Figure 6.4: Measured loss bursts distribution. Note that Table 6.2 shows that packet losses are far less common for messages transfers. The loss bursts during messages are thus larger but less frequent.

6.3.2.2 Packet losses during low bitrate transfers

We now focus on the low bitrate messaging use-case. The two last columns of Table 6.2 show the packet loss ratios measured during those transfers. Conversely to the H3 experiments, the computed loss ratio is only slightly smaller for downloads than for uploads. The loss rate is also significantly lower compared to H3. Given the low bitrate of the messaging use-case and the overall low RTT previously measured, we can expect that fewer packet losses were caused by congestion here. Note however that from the transport viewpoint, there is no known way to distinguish between congestion and medium-induced losses. This is why loss-based congestion control algorithms such as Cubic [HRX08] interpret every loss event as a congestion signal.

Figure 6.4b shows the loss bursts lengths distribution for the messaging experiments. While packet losses are a lot less frequent compared to the H3 experiments (see Figure 6.2), the loss bursts are in general longer when occurring. We conjecture that most of the loss events occurring during the H3 experiments are due to congestion: they are more frequent and only concern a few packets, while the loss events encountered during the message transfers may be mostly related to the medium, sometimes being even comparable to small network outages with some loss bursts spanning more than 100 packets (also present for H3 transfers). Concerning the loss events duration, most events for message transfers were shorter than 1ms. However, we noted 95^{th} and 99^{th} percentiles of 104 and 127 ms which are larger than the same

percentiles for H3 downloads (note that the loss events for message transfers are a lot more rare than H3 loss events and that H3 transfers probably mostly encounter congestion-induced losses). Such long loss events would be more difficult to recover using FEC as the repair symbols would arrive more than 100 milliseconds after the start of the event. Similarly to H3, we also detected small network outages with loss events lasting more than 1 second.

Finally, we checked that those losses were neither caused by our network nor our server by running downloads for both H3 and messages transfers from a machine in Amsterdam (i.e., close to an exit point of the Starlink network) towards our H3 server. For H3 (resp. messages) downloads, over more than 5.8 M (resp. 2.8 M) packets sent by our QUIC server, only 10 (resp. 8) were lost, making loss events nearly absent when the client is outside Starlink.

To sum up these packet loss experiments, we can say that the loss events occurring when the link is loaded are more frequent and only affect a few consecutive packets. Without link pressure, the loss events are more rare, concern overall more consecutive packets and last longer.

6.3.3 Throughput

Figure 6.5 shows the throughput distribution for three experiments: Ookla Speedtest on Starlink, H3 bulk download on Starlink and Ookla Speedtest on the regular SatCom access. We first discuss the Ookla speed tests and then the H3 results.

6.3.3.1 Speed test results

By looking at the left graph of Figure 6.5, we can see that Starlink's download throughput ranges between 100 and 250 Mbps. The median value is 178 Mbps, while the maximum is 386 Mbps. This maximum is surprisingly high given the company's public statements, i.e., download speeds between 100 Mbps and 200 Mbps. We note that they enable the use of bandwidth-intensive services, such as High-Definition video streaming. Netflix's 4K videos require a download bandwidth of 15 Mbps [22c], while Disney+ recommends 25 Mbps [22b].

The upload throughput, on the right of Figure 6.5, is significantly lower, reaching a median of 17 Mbps. Fewer than 5% of the cases exceed 30 Mbps and the highest observed rate is 64 Mbps. For both metrics, we cannot find a seasonality in the measurements. Looking at the different hours of the day, the median throughput varies by less than ±10% with no apparent day-night cycle. Furthermore, we have not observed any increasing or decreasing trend in the measurements over our three months of experiments, and the distributions assume approximately the same average values and variability.

Comparing with traditional SatCom, we find that Starlink provides higher throughput in both scenarios. Considering download, with a median value

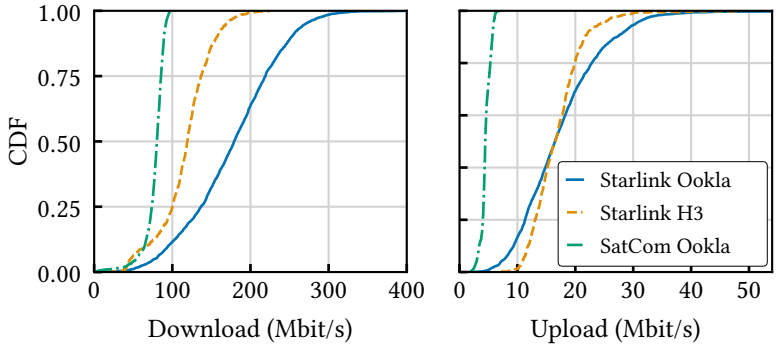


Figure 6.5: Measured throughput distribution.

of 178 Mbps, Starlink is more than twice as fast as SatCom (82 Mbps). The situation is similar for upload: the traditional SatCom connection inherently offers lower upload throughput (4.5 Mbps in median), as it is limited to a bitrate of 10 Mbps.

We can briefly compare these values with mobile networks looking at recent related work. Safari *et al.* [Kha+17; TKG20] conducted a large-scale measurement campaign in 2018 involving 4 European mobile network operators in 2 countries. For download, they found that in the best case (4G with good signal quality), mobile networks provide a median throughput of 29.5 Mbps. For upload, the authors found a median bitrate of 14 Mbps, comparable to Starlink's 17 Mbps. However, keep in mind that these throughput measurements [Kha+17; TKG20] are already 5 years old at the time of writing and thus possibly outdated.

6.3.3.2 HTTP/3 transfers

We now measure throughput using HTTP/3 with our server located in Belgium, the same country as the Starlink access. We report the measured throughput distribution for the download and upload of 100MB of data in Figure 6.5. We ran two experiment sessions, one until the 7^{th} of April and one starting from the 25^{th} of April. We observed a difference of download throughput during the two sessions but the upload throughput stayed the same. All the parameters are the same for the two sessions but we observed an increase of download capacity for QUIC. Figure 6.5 thus shows the results for the second session as they represent the most up-to-date results for Starlink.[1] The download bitrate sits mostly between 100 and 150 Mbps which is in line with what

[1]While not present on the graph, all packet captures for the first session are provided in the artifacts of this thesis.

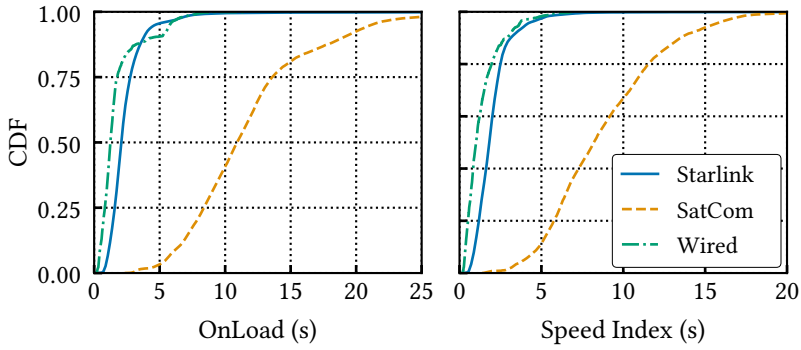


Figure 6.6: Web browsing performance.

is announced by Starlink but lower than the best results obtained with the Ookla TCP speed tests and lower than what our QUIC server can deliver to other wired endpoints. We also excluded the possibility of an incorrect receive window tuning of the quiche implementation by running additional experiments with a 150MB receive window, leading to similar results. The difference in download throughput may be due to the fact that regular speed tests use at least four concurrent TCP connections while the QUIC download uses one single connection, reacting more strongly to losses [FL20; Mac+]. It is also possible that Ookla speed tests are prioritized by the operator. The measured upload throughput is similar for the two sessions and is in-line with the Ookla speed test results: they have the same median, although the QUIC results are more stable. In summary, Starlink outperforms traditional SatCom for both download and upload. The measured throughput with QUIC is lower compared to TCP speed tests for downloads but similar to TCP speed tests for uploads.

6.3.4 Browsing Performance

We now quantify the Starlink performance for Web browsing. We compare the *user experience* of Starlink users against other access technologies. We resort to the *onLoad* and *Speed Index* metrics that have been shown to correlate with it [Hor+18]. We continuously visit a set of 120 popular websites in our country, using PC-Starlink, PC-SatCom and PC-Wired.

In Figure 6.6, we show the CDF of QoE-related metrics. Starting from *onLoad* (left of Figure 6.6), we find that it generally ranges from a few to 15-20 seconds, depending on the website and conditions. Starlink provides a median *onLoad* of 2.12s and an interquantile range between 1.60s and 2.78s. Experiments with the SatCom equipment show that *onLoad* is substantially larger, 10.91s on median. The distribution ranges from 8.36s (25^{th} percentile) to 13.59s

(75^{th} percentile). It is likely that this performance is due to the high latency of the SatCom connections, which affects the operation of TCP and HTTP. Note that rendering a web page requires opening multiple connections to different servers to retrieve all page objects. In our dataset, a single visit results in 15 connections on average. On SatCom, opening a connection (including the TLS handshake) takes an average of 2 030ms, while Starlink requires only 167ms. Concerning the performance of the wired network, the median *onLoad* is 1.24s, still considerably lower than the other two cases. Although we do not run experiments on mobile networks, we mention that Rajiullah *et al.* [Raj+19] used a large testbed of mobile nodes to visit a number of popular websites. They measure *onLoad* times on the order of $2-5$s, thus moderately higher than what we measure on Starlink. Similar considerations apply to the *SpeedIndex* (right of Figure 6.6). Starlink shows a median performance of 1.82s, outperforming SatCom with a 8.19s median *SpeedIndex*. Starlink performance is closer to the wired setup.

To sum up, Starlink outperforms SatCom for web browsing and has close performance to regular wired access. Looking at QoE-related metrics, Starlink is $75-80\%$ faster than traditional SatCom.

6.3.5 Middleboxes and traffic discrimination

SatCom solutions often deploy PEPs to alleviate the problems due to the high link latency. Some operators also apply traffic discrimination to control the bandwidth used by applications on their network. In this section, we analyze the presence of middleboxes and traffic discrimination on the Starlink network.

PEPs and middleboxes: We first use Traceroute and Tracebox [Det+13] to detect PEPs and middleboxes. Traceroute shows us the presence of two levels of NAT at the two first nodes: the Starlink access point (`192.168.1.1`) and a carrier-grade NAT node (`100.64.0.1`) at the exit of the satellite link. Tracebox does not show the presence of any PEP: the TCP handshake is correctly performed in the destination network. Only the TCP and UDP checksums are altered by the NATs.

Traffic discrimination: We employ Wehe [Li+19], a state-of-the-art tool to detect traffic discrimination. It replays packet traces of 22 popular services including video streaming (e.g., Netflix, YouTube) and video calls (e.g., Zoom, Skype). It then replays the same traces with randomized bytes to prevent the operator from correlating this traffic to the original service. In the case of Starlink, we launched ten times the complete Wehe tests but could not find any traffic discrimination policy in place, at least for these popular services.

6.4 Conclusion

This study presents a tour of our Starlink access point. Our TCP and QUIC measurements show that Starlink delivers its promised low latency and high throughput. It enables the use of latency-sensitive services that struggle with traditional SatCom. Interestingly, early simulations of LEO constellations (see Hypatia [Kas+20], among others) predicted similarly low values for RTT, especially in this first phase of low utilization. Our QUIC measurements reveal additional details about the RTTs and packet losses under load. During HTTP/3 bulk transfers, RTTs increase more than when applications exchange messages at a low rate.

At the application level, we have studied QoE for web browsing and found it to be radically better than traditional SatCom. Note, however, that we only studied a limited number of websites because we wanted to visit them hourly. We did not account for differences in experiences that could be due to different browsers, different devices, or other factors. Also, we only visited landing pages, while a more realistic campaign should include internal pages [Aqe+20].

The most interesting part of this study for this thesis is the presence of packet loss even at low network utilization. Thanks to QUIC's precise acknowledgments, our measurements show that packet losses are more frequent during bulk transfers and provide some characterization of the loss patterns. While a few loss events span several dozens of packets and last more than 100 milliseconds, most of then seem easily recoverable using Forward Erasure Correction. The large available bandwidth also leaves room for the sending of redundancy without diminishing the performance of latency-sensitive traffic. The next chapter focuses on providing actual performance improvements in such a real lossy network for real applications without the performance drawbacks caused by FlEC.

QUIRL: improvements for real applications on real networks

7

Until now, our solutions were focusing on emulated (Chapter 2 and 4) or simulated (Chapter 5) lossy environments. Furthermore, the studied scenarios were synthetic, with self-implemented bulk and message transfers. The latter was studied using traces from a video conferencing software without knowing if our solution actually improved the video quality of experience. In this chapter, we study the actual improvements that can be obtained from the ideas explored in this thesis. We apply the lessons learned from the previous chapters on real applications running over real networks. Chapter 6 introduced a network having both congestion-induced and medium-induced packet losses that is a good real-network candidate to experiment with QUIC and FEC.

We perform a second real-network experiment with FlEC and confirm that due to the heaviness of the PQUIC framework, FlEC can provide bad experimental results outside simulations. We start from the ideas brought by FlEC and adapt them in real-world scenarios. We then develop QUIRL (short term for "quirrell", derived from the words QUIC, REdundancy and Low-Latency), the resulting solution in a production QUIC implementation. We evaluate its performance for popular video streaming and web-based applications over a real lossy network. Our evaluation shows that for video streaming QUIRL improves the video quality while meeting strict delay requirements. For web transfers, QUIRL efficiently reduces the tail latency when packet losses occur without causing harm when there are no losses.

This chapter is organised as follows. We describe QUIRL and its design in Section 7.2. Section 7.3 addresses QUIRL implementation details. Section 7.4 and 7.5 then describe and evaluate how QUIRL can be used with curl and GStreamer to improve their quality of experience.

7.1 Existing FEC extensions for QUIC

During the writing of this thesis, rQUIC [Gar+19; Zve+21] proposed another integration of FEC into QUIC and studied other practical applications such as web page load time and DASH [ISO22]. In contrast with QUIC-FEC and FlEC,

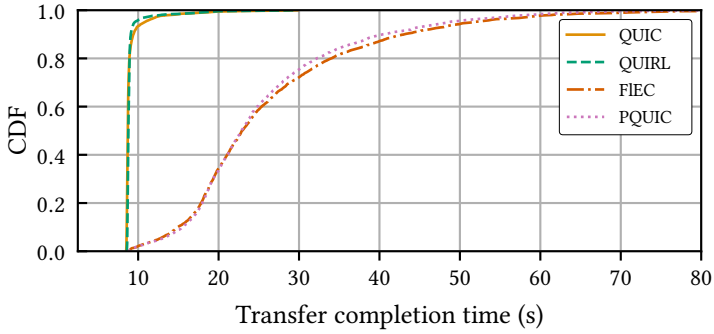


Figure 7.1: Transfer completion times for 100MB files transfers

rQUIC hides the packet loss signal from the congestion control, potentially resulting in unfair behaviour towards non-coded connections. All the rQUIC experiments [Zve+21] were performed using a simple XOR code unable to handle loss bursts and rQUIC has been evaluated using ns-3 simulations applying uniformly distributed losses exclusively. On the other side, while FlEC obtained good simulation results with bursty losses, we observed in Chapter 5 that it could lead to bad results in a real network. To confirm this observation, we performed real-network experiments using a wired network instead of our Starlink access point. We obtained a similar performance diminution as Chapter 5. This is illustrated by Figure 7.1 as a preliminary experiment. In this Figure, we ran 100MB downloads from an OVH Virtual Private Server connected at 100Mbps. Downloads were performed using Cloudflare's quiche implementation, QUIRL (the solution proposed in this chapter) and PQUIC with and without the FlEC plugin, all four used the Cubic congestion control. FlEC was configured for the bulk transfer use-case. The kernel UDP receive buffers were configured such that no packet loss was due to them being too small. Not only PQUIC struggled to saturate the link using Cubic, but enabling FlEC led to performance degradation, with the 90th percentile download taking nearly three seconds longer with FlEC. This degradation is confirmed with a Welch hypothesis test providing a p-value of 0.01 when comparing the PQUIC and FlEC curves. On the other side, both quiche and QUIRL can saturate the 100Mbps link with a median completion time of 8.7 seconds for both, enabling their use for bulk scenarios in real networks. The improvements of QUIRL over regular QUIC are explored in more details in this chapter.

The motivation behind the design of QUIRL is the fact that the existing works were only providing benefits in controlled simulated or emulated environments. QUIC-FEC showed bad results for large transfers, rQUIC was not evaluated with real, non-uniform loss patterns and FlEC shows bad experimental results in the case of low-loss rate transfers. The PQUIC implementation

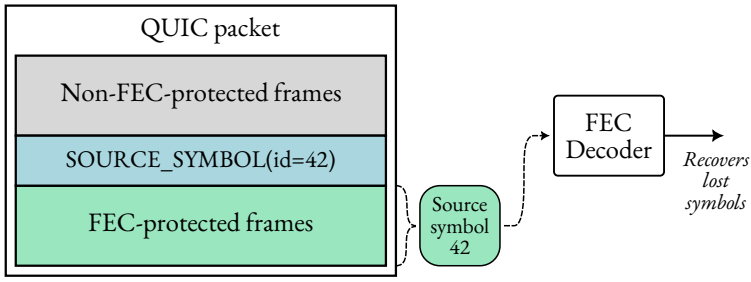


Figure 7.2: Receiving a QUIC packet containing a SOURCE_SYMBOL frame.

used by FlEC is also not evolving and maintained as much as production QUIC implementations. The goal of this chapter is therefore to propose a FEC extension providing latency improvements for different QUIC applications in real networks with real loss patterns.

7.2 QUIRL Design principles

We use the acronym QUIRL to refer to both the FEC extension proposed in this paper and its implementation. QUIRL introduces a few design changes compared to FlEC and QUIC-FEC. We first describe how QUIRL identifies the FEC-protected parts of a QUIC payload. We then explain how QUIRL encodes repair symbols in QUIC packets. Next, we discuss how QUIRL interacts with congestion control. We finally address how repair symbols are scheduled depending on the use-case.

7.2.1 Identifying FEC-protected payloads

Similarly to FlEC, we use a QUIC frame to identify the packet payloads to be considered as source symbols. We call it the SOURCE_SYMBOL frame. The difference we introduce with FlEC is that we do not consider anymore the whole packet payload to be part of the source symbol. Only the frames coming after the SOURCE_SYMBOL frame in the QUIC payload are part of a source symbol and are then FEC-protected. The SOURCE_SYMBOL frame assigns a unique identifier to each source symbol, allowing the receiver to determine unambiguously which symbols need to be recovered. The reception of a packet containing a SOURCE_SYMBOL frame is illustrated in Figure 7.2. The QUIC receiver considers all the FEC-protected frames as part of source symbol 42. This source symbol is then passed to the FEC decoder that will recover lost source symbols when possible using the already received repair symbols. Conversely to QUIC-FEC that needs a specific header field, QUIRL is fully compatible with QUIC version 1 [RFC9000].

7.2.2 Serializing the repair symbols

Similarly to previous work, QUIRL uses a REPAIR frame to carry the repair symbols in a QUIC packet. To support different erasure correcting codes, the payload of a REPAIR frame is an opaque payload that is passed as-is to the FEC decoder. The format of the repair symbols themselves depends on the used erasure correcting code.

7.2.3 QUIRL and congestion control

QUIRL does not hide packet losses to the congestion control algorithm, even when lost source symbols have been recovered. QUIRL generates QUIC ACK frames only for packets received from the network. QUIRL is thus fully in-line with the recommendations of RFC9265 [RFC9265]. The REPAIR frames are congestion-controlled, meaning that they are subject to congestion control as regular application data. QUIRL does not send REPAIR frames when the congestion window is full.

QUIRL defines the SOURCE_SYMBOL_ACK frame to explicitly signal that a source symbol has been received, either from the network or through FEC recovery. In practice, SOURCE_SYMBOL_ACK frames are only sent for recovered source symbols, as their reception through the network can also be deduced from regular ACK frames. When receiving a SOURCE_SYMBOL_ACK frame, the QUIC sender can remove the recovered data from its retransmission queue, but the original QUIC packet containing these data is not considered as received until a regular ACK frame acknowledges it. This frame shares similarities with QUIC-FEC's RECOVERED frame. However, QUIC-FEC uses ACK frames to directly acknowledge recovered packets and the RECOVERED frame is used to reduce the congestion window afterwards. QUIRL does not acknowledge packets that were not received from the network in conformance to RFC9265.

Figure 7.3 shows an example of transfer using QUIRL. The three STREAM frames sent by the sender are protected by FEC but it is not the case for the PING frame of Pkt_1 since it is placed before the SOURCE_SYMBOL frame in the packet payload. The repair symbols are transmitted through the two REPAIR frames carried by Pkt_4 and Pkt_5. Once both repair symbols are received, the STREAM frames of Pkt_1 and Pkt_3 can be recomputed using the erasure correcting code and their stream data can be directly delivered to the application. The PING frame of Pkt_1 remains lost as it was not part of the source symbol and has therefore not been recovered. Pkt_2 is acknowledged using a classical ACK frame as it has been received normally through the network, but Pkt_1 and Pkt_3 are not. Source symbols 1 and 3 are acknowledged using a SOURCE_SYMBOL_ACK frame to prevent the sender from considering that Pkt_1 and Pkt_3 have been received from the network. These packets will

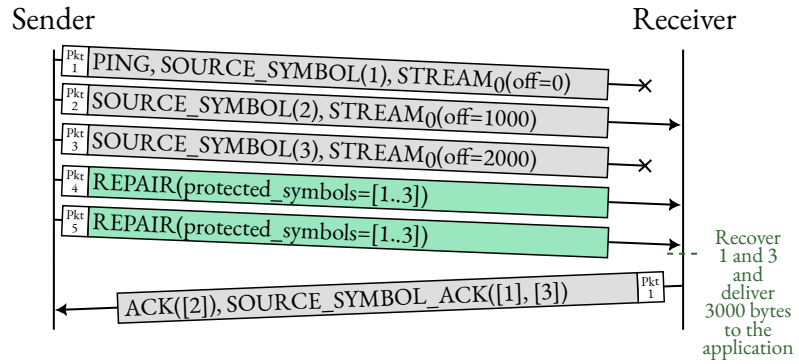


Figure 7.3: Example of FEC-protected transfer using QUIRL, with the first and third QUIC packets being lost. The FEC-protected data are the three STREAM frames. They can be recovered without waiting for retransmissions using the two repair symbols contained in the fourth and fifth packets.

be marked as lost by the QUIC loss detection mechanism but the two lost STREAM frames they contain will not be retransmitted. The PING frame of Pkt_1 will be retransmitted later.

7.2.4 Scheduling the repair symbols

As discussed in Chapter 5, the best moment to send repair symbols can strongly vary depending on the application and its traffic pattern. FlEC uses protocol plugins to define application-tailored redundancy scheduling, inserting different plugins depending on the requirements. However, the drawback is that protocol plugins are significantly more CPU-costly than a native implementation.

In this chapter, we propose two QUIRL redundancy schedulers that are adapted respectively to bulk transfers and real-time bursty traffic such as low-latency video transfers. The schedulers are implemented natively inside the quiche implementation. This approach is less flexible than FlEC with protocol plugins. Indeed, QUIRL applications cannot insert their own API functions unless they recompile quiche and tune it for their use-case. This is not doable for some applications such as the ones running in web browsers. However, the native implementation of QUIRL allows us to benefit from better performances compared to the PQUIC framework. The two schedulers developed here are articulated around the same *no harm* principle: avoid sending repair symbols when new application data can be sent, unless required to meet the latency objectives of the application. Prioritizing source symbols this way ensures no harm is caused to bandwidth-demanding applications, only sending repair symbols during quiescence. The scheduling algorithms for bulk and video transfers are presented in more details in Sections 7.4.1 and 7.5.

7.3 Implementing QUIRL

QUIRL is built on top of Cloudflare's quiche implementation [Clo22]. Written in Rust, quiche is a production QUIC implementation that is already used on Cloudflare's edge servers, the `curl` command-line tool [Curl] and the DNS resolver of Android version 10 and onwards. Our goal was to make as little changes to quiche as possible to keep QUIRL maintainable and up-to-date. In total, QUIRL adds about 1500 lines to quiche. QUIRL adds the encoding of the new frames and the use of FEC encoders and decoders in the quiche packet processing loop. We also implemented packet loss estimators to estimate the network loss conditions during a QUIC connection and integrate this knowledge in our redundancy scheduler. We provide the FEC encoders and decoders separately from QUIRL in dedicated Rust crates outside quiche so that they can be reused by other protocols and other QUIC implementations. In total, these external network coding crates consist in 12000 lines of Rust code available for everyone.

7.3.1 Erasure Correcting Codes

In addition to RLC, we also provide QUIRL with the Tetrys erasure correction code. Tetrys is presented as a patent-free yet powerful fountain error correcting code [Tou+11]. It adapts the Reed-Solomon encoding algorithm to be used in an online manner. While the recovery capabilities are more limited than other fountain codes (cycles can occur in the generation of repair symbols), they are sufficient for the usually low loss rates encountered on the Internet. The Rust implementation of both RLC and Tetrys-inspired erasure correcting codes are part of the code we release publicly and can be used interchangeably without modifying the application or the protocol implementation. The experiments described in this chapter rely on the Tetrys-inspired code.

7.3.2 WebTransport

Web browser applications might not be able to directly use QUIC streams and datagrams as they are limited to the HTTP/3 API [RFC9113]. WebTransport aims at providing these applications with a stream and datagram API that is later mapped to QUIC streams and datagrams [FKV23; VJA23]. WebTransport may rapidly become one of the main usages for QUIC.

To enable QUIRL to be used by a large number of applications, we provide an implementation of WebTransport and release it publicly, also as an external crate. The video experiments performed in this article use this WebTransport crate and show that QUIRL can also deliver benefits in these scenarios. A promising future work is to use QUIRL together with WebCodecs [W3C23a] for

BurstSize	The minimum burst size needed to be sent to trigger the sending of repair symbols (default: 0 byte).
MaxJitter	The maximum loss-induced jitter allowed by the application (default: 0 ms).

Table 7.1: Parameters for the bursty redundancy scheduler.

video applications and rely on WebAssembly to propose application-defined QUIRL redundancy schedulers running in the browser.

7.4 Latency-sensitive video streams

Applications doing real-time media transfers such as video conferencing tend to regularly send data bursts whose size depends on the type of video frame. Key video frames (i.e. I-frames for the H264 codec [Wie+03]) often span from several dozens to several hundreds of kilobytes, while others are significantly smaller. In the case of a real-time video transfer, it makes sense to send FEC regularly to ensure a low-latency delivery of each video frame. Depending on the latency requirements, it might be interesting to protect several video frames with the same set of repair symbols, similarly to the FlEC scheduler for delay-constrained messaging.

7.4.1 Redundancy scheduler

We provide a specific redundancy scheduler tailored for this kind of applications. We allow the application to parametrize the redundancy scheduler by choosing values for the parameters described in Table 7.1, but they can already achieve good performance with the default values. These parameters replace the application-defined API of FlEC that used protocol-plugins.

Using the *BurstSize* parameter, the application has control over the FEC-eliciting data, i.e. the data causing the sending of repair symbols. This allows reducing the redundancy overhead by not protecting small video frames for instance. The loss of small video frames generally has a lower visual impact on the video playback than larger frames. The *MaxJitter* parameter can be used by the application to specify the maximum affordable data delivery delay. Low-latency video streaming applications regularly use a *playback buffer* that delays the video playback by a few milliseconds to smoothly handle jitter in the data delivery. Specifying the maximum affordable jitter allows the redundancy scheduler to adjust the number of video frames protected with the same set of repair symbols, therefore reducing the FEC overhead. These two parameters are not mandatory for the application as they do not impact significantly the recovery capabilities of QUIRL. They only aim at optimizing the overhead

over the connection if the application configures them. The generation of repair symbols depending on these parameters is depicted in Algorithm 5. When enough data are sent and need to be protected, the algorithm waits for MaxJitter before sending the repair symbols. The number of repair symbols is determined in function of the average number of lost symbols during a loss event ($\mathcal{L}$) and its standard deviation ($\sigma_{\mathcal{L}}$). This amount is capped by a fraction of the bytes in flight (β) that we set to 0.3 for the experiments.

Algorithm 5 Repair symbols scheduler for bursty traffic

Require: BustSize, MaxJitter from Table 7.1.
Require: LastBurst, the last sent application data burst.
Require: BIF, the current amount of bytes in flight.
Require: $\mathcal{L}$, the average number of lost symbols during a loss event and $\sigma_{\mathcal{L}}$ its standard deviation.
  if LastBurst.length $\geq$ BurstSize $\wedge$ $isAppLimited()$ **then**
    if Now < LastBurst.sentTime + MaxJitter **then**
      $WakeUpAt$(LastBurst.sentTime + MaxJitter)
    else
      $n \leftarrow \min(\beta * \text{BIF}, \mathcal{L} + 2 * \sigma_{\mathcal{L}})$
      sendNRepairSymbols(n)
    end if
  end if

7.4.2 Reducing the latency of GStreamer RTP flows

The goal of this work is to improve the performance of real applications over actual networks. Baltaci *et al.* recently showed an interest at using RTP over QUIC to transmit H264-encoded aerial vehicule videos to perform remote piloting [Bal+22]. The H.264 video codec defines three types of video frames [Wie+03]: I, P and B frames. I-frames are the largest as they contain a full picture. P-frames contain motion vectors to recompute the picture from one or more previous frames. B-frames contain motion vectors to reconstruct the picture from previous *and* future frames. In their work, the videos are encoded using low-latency settings. Only I-frames and P-frames are sent since B-frames require an additional latency to be decoded. I-frames are also regularly sent to frequently refresh the video and cope more easily with frame losses. We reuse the videos used by Baltaci *et al.* in our evaluation and assess how QUIRL can impact the transfer quality. In this context, we analyze the benefits that QUIRL can bring to an existing video application using *GStreamer*. GStreamer is a flexible open source suite used to transfer videos over a network that was also used by Baltaci *et al.* for their evaluation. Low-latency streaming

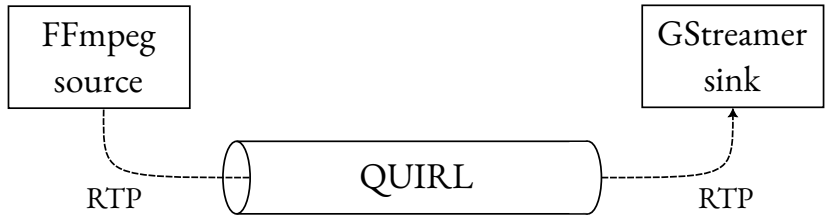


Figure 7.4: FEC-protected relay forwarding GStreamer packets using QUIRL.

is generally achieved by sending the video over RTP. We use FFmpeg [23] to extract the video frames from the file without re-encoding it and send it over RTP. The GStreamer suite provides the `gst-launch` command-line application allowing receiving a video stream from an RTP *source* and display it to the user (GStreamer *sink*). We implement an RTP relay using QUIRL as illustrated in Figure 7.4. The relay intercepts packets sent by the FFmpeg source at one end and sends them through a QUIC connection protected by FEC using the redundancy scheduler described in Section 7.4.1. The relay decapsulates the RTP packets at the other end and forwards them to the GStreamer sink.

The RTP packets are parsed at the relay entry point. Each RTP packet is placed in its own QUIC stream to avoid head-of-line blocking between packets. We do not use QUIC datagrams as the RTP packets could be too large to fit in a single QUIC datagram with some network MTUs. Using several QUIC streams allows packets to be independently delivered to the RTP receiver and skipped when they cannot be received on-time. One step further could be made by sending separate H264 slices [RFC6184] on separate QUIC streams to further reduce the possible head-of-line blocking inside a single packet. However, doing so would need a deeper payload parsing on the RTP relay. A better solution would be a video application totally adapted to the QUIC API, but this is left as future work.

We study two metrics for the video transfers. The first metric is the video frames and packets *lateness*. Video frames have a presentation timestamp encoded in the RTP packet header. The lateness of an RTP packet is the difference between the moment it was received and the moment the video frame would have been shown to the user if no playback buffer was used by the application. The lateness of a video frame is equal to the lateness of its latest packet. Network packets can arrive late for several reasons: they can be lost and retransmitted or they can be delayed by the protocol's congestion control and pacing mechanisms, without any packet loss. Applying a playback buffer of x milliseconds (e.g. using GStreamer's *rtpjitterbuffer* plugin [GStrDoc]) allows smoothly displaying video frames with a lateness up to x milliseconds when these events occur. The frames with a larger lateness will likely be skipped by

the receiver. Measuring the frame and packets lateness allows directly studying the latency impact of QUIRL independently of the application behaviour. The second metric we analyze is the *Structured Similarity (SSIM)* [Wan+04]. This state-of-the-art metric also used by Baltaci *et al.* gives a 0 to 1 similarity score to assess the visual similarity between the frames displayed by the GStreamer sink and the frames of the original video. An SSIM of 1 means that the displayed frame is identical to the original one. SSIM therefore provides a way to quantify the visual impact of QUIRL to the streaming application. In these experiments, we set the BurstSize parameter to 5000 bytes, meaning that video frames smaller than 5000 bytes will not explicitly elicit the sending of FEC. The rationale is that smaller video frames carry less information, loosing them has therefore a smaller impact. The MaxJitter parameter is set to 40 milliseconds. With a video displaying 30 frames per second, this means that after the sending of a FEC-eliciting video frame, the FEC scheduler will wait for another frame to be sent before sending the repair symbols. This adds a recovery delay for the first frame but allows protecting several video frames within a single round of repair symbols.

Adapting quiche for real-time media transfer

We implemented packet pacing on the sender in order to reduce as much as possible the potential losses caused by sending a large burst of packets on the network. We carefully configured the UDP buffer sizes such that no packet was dropped due to the UDP receive buffer capacity. Concerning the congestion control algorithm, the two algorithms provided by quiche are Cubic and BBRv1. BBRv1 can be problematic for real-time video transfer as it regularly decreases its congestion window down to 4 packets to empty the network buffers. This problem is solved in BBRv2 but it is not yet supported by quiche. We therefore use Cubic. Since Cubic was not designed for real-time video transfer either, we added a one-line change to Cubic to adopt a behaviour similar to real-time congestion controllers such as SCReAM [RFC8298]. It allows increasing the congestion window upon receiving acknowledgements when the bytes in flight exceed 66% of the congestion window instead of requiring the congestion window to be fully utilized. This behaviour is not uncommon and is especially useful for real-time flows that send bursts of varying size but that rarely utilize the full capacity of their congestion window. This behaviour has no impact for bulk, bandwidth-intensive flows as their congestion window is always fully utilized.

7.4.3 Starlink setup

Since we observed in Chapter 6 that losses are common on Starlink, especially on the upload path [Mic+22; Ma+22; Kas+22], we reuse this Starlink vantage

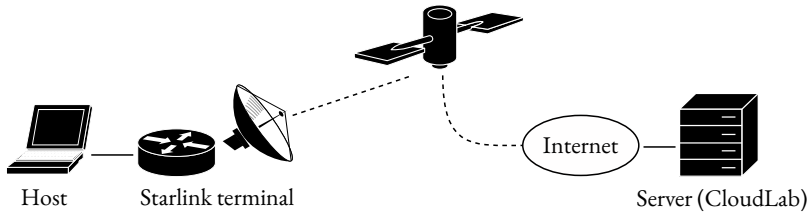


Figure 7.5: Starlink setup used in the real-network experiments.

	50ms	75ms	100ms	200ms	500ms
QUIC	86	61	64	68	69
QUIRL	69	55	65	62	63
RTP	69	57	78	71	63

Table 7.2: Number of video transfers performed per solution and per GStreamer playback buffer size.

point for our experiments. We connect a laptop to the Starlink terminal and setup a Cloudlab server in Utah [Dup+19], as illustrated in Figure 7.5. The round-trip time between our Starlink-connected laptop and the cloudlab server is around 150 milliseconds.

7.4.4 Real network experiments results

This section analyzes the results obtained from video transfers performed over our Starlink access. In total, we performed 1000 video transfers. These transfers were made using three different solutions: QUIC, QUIRL, and the classical RTP used by GStreamer and FFmpeg. We also perform experiments with different playback buffer sizes for the GStreamer sink using GStreamer's *rtpjitterbuffer* plugin. A larger playback buffer induces a higher latency between the source and the sink which is often undesirable. Since the solution used was drawn at random for each experiment, Table 7.2 summarizes the number of experiments performed for each solution and for each playback buffer.

Figure 7.6 shows the experimental Cumulative Distribution Function (CDF) of the RTP packets and video frames latenesses aggregated over all video transfers. As frames and packets lateness are computed on our relay, they cannot be computed for regular RTP transfers which do not pass through the relay.

Similarly to Chapter 6, we observed approximately 0.4% of packet loss on our Starlink access point on the upload path when there is no load on the link. By looking at the left graph, QUIRL keeps around 0.4% more packets below a 150 milliseconds lateness compared to QUIC. While this seems a small amount of packets, the graph on the right illustrates that only a small proportion of

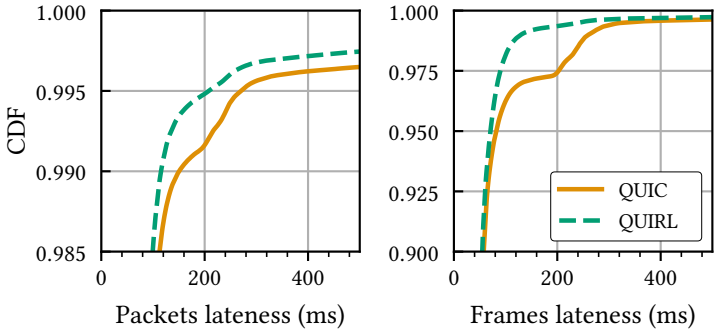


Figure 7.6: RTP packets and video frames lateness distribution for video transfers over Starlink.

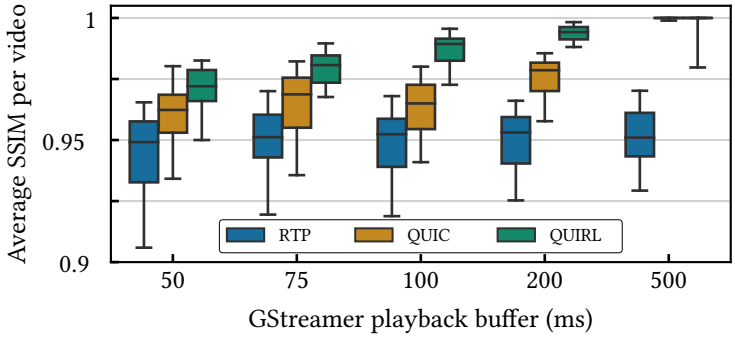


Figure 7.7: Average SSIM per transferred video for different playback buffer sizes.

late packets can have a significantly larger proportional impact when it comes to the lateness of the video frames themselves. Indeed, some video frames are composed of several of dozens of packets and losing a single packet of these frames impacts the lateness of the entire video frame. On the right graph, more than 98% of the frames have a lateness under 100ms using QUIRL, while the 98th percentile for QUIC is 221ms, more than doubling the playback buffer required to deliver the same amount of frames on-time.

Although frames and packets latenesses are valuable metrics to quantify the latency improvements brought by QUIRL, it is also interesting to look at the actual impact on the application. Figure 7.7 shows the distribution of the average SSIM for all video transfers using different GStreamer playback buffer sizes. The average SSIM of a video transfer is obtained by averaging the SSIM of every video frame sent during this transfer. A higher average SSIM corresponds to a higher visual fidelity of the video transfer on average. We observe that

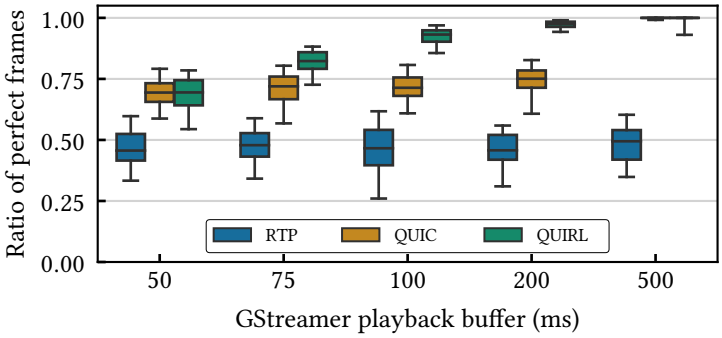


Figure 7.8: Per transfer ratio of perfect frames for different playback buffer sizes.

QUIRL provides better average SSIM scores for every studied playback buffer size. Gradually enlarging the playback buffer allows QUIRL to improve the average SSIM further, with a median average SSIM score of respectively 0.97, 0.98, 0.99, 0.995 and 1 for the 50ms, 75ms, 100ms 200ms and 500ms playback buffer sizes. Conversely, QUIC only improves the average SSIM when the playback buffer size exceeds the RTT of the connection, allowing the protocol to recover from losses using retransmissions. RTP provides similar average SSIM results regardless of the playback buffer size as the default RTP sender performs no loss recovery whatsoever.

We also computed the ratio of *perfect frames* (frames with an SSIM of 1) for each video transfer. Figure 7.8 shows the distribution of the ratio of perfect frames across the different video transfers. As H264 P-frames depend on other video frames, a missing frame can have a negative impact on the frames that follow, causing a cascading effect. In median, a video had more than 70% of perfect frames when using QUIRL and QUIC with the 50ms playback buffer. This median ratio falls down to 45% with RTP that does not pace the sent packets as QUIC does (note that RTP still achieves an average SSIM of 0.95 as illustrated in Figure 7.7). Comparing Figure 7.7 and Figure 7.8, we can see that with the 50ms playback buffer, QUIRL obtains a similar perfect frames ratio to QUIC but a better average SSIM. One reason is that although lost symbols could be recovered using FEC (improving the average SSIM), the 50ms playback buffer is too small for large frames (>70kB) to be fully received on-time even without packet losses due to the available upload bandwidth that varied between 8 and 15Mbps. Those partially-received large frames in turn have an impact on the following frames that may be fully received but depend on the larger frame to be perfectly decoded. With the 75ms and 100ms playback buffers, video transfers using QUIRL had respectively more than 82% and 93% of perfect frames in median, which is 12% and 23% better than

with the 50 ms buffer. With this additional time, QUIRL can deliver more video-frames on-time when more repair symbols are needed to perform loss recovery. On the other hand, transfers using QUIC and RTP obtain similar performance than with the 50ms playback buffer. The price to pay for these latency improvements is an increase in the required bandwidth. With the BurstSize and MaxJitter parameters respectively set to 5000 bytes and 40 milliseconds, the video transfers using QUIRL in these experiments require slightly under 4Mbps (including QUIC headers and control information) while they require 3.2Mbps with QUIC.

With the 500 ms playback buffer, both QUIC and QUIRL can obtain 100% of unaltered frames in median, at the price of an important delay between the GStreamer source and sink. With such a latency, the video transfer is arguably not real-time anymore. The performance of RTP stays unchanged as no retransmission is performed whatsoever.

7.5 HTTP/3 objects

We now focus on HTTP/3 and the exchange of web objects of different sizes. It is worth remembering that in this scenario, every unused repair symbol results in a waste of network resource, as a new source symbol could have been sent instead. This has the effect of increasing the completion time of that kind of transfers. As packet losses are varying and unpredictable, there is a high probability of sending unneeded repair symbols even when the amount of repair symbols matches the estimated loss rate of the network. Inspired by the results of FlEC, we here focus on recovering from last-flight losses and losses occurring when the sender is flow control-blocked. Our redundancy scheduler for HTTP/3 objects, depicted in Algorithm 6, is thus straightforward. We send the REPAIR frames during quiescence periods, i.e. when no application data can be sent. This happens either when the sender is blocked by the stream flow control or when all the application data have been sent (but not necessarily been acknowledged yet). The repair symbols protect every source symbol currently in flight.

As quiche can be used as a QUIC backend for curl, we built a recent version of curl and linked it with QUIRL. A tiny code change was required on the curl client to ensure it correctly sends the packets containing REPAIR frames during uploads.

Curl is used for HTTP/3 transfers of various sizes. One way to improve curl's performance is to reduce the Tranfer Completion Time (TCT): the time between the start of the HTTP/3 request and the complete reception of the response. The amount of redundancy is also determined by the loss characteristics of the link ($\mathcal{L}$ and $\sigma_{\mathcal{L}}$) and capped by a proportion of the bytes in flight (β), set to 0.3 in the experiments.

Algorithm 6 Repair symbols scheduler for HTTP/3 objects

Require: BIF, the current amount of bytes in flight.
Require: $\mathcal{L}$, the average number of lost symbols during a loss event and $\sigma_{\mathcal{L}}$ its standard deviation.
  if $isAppLimited()$ **then**
    $n \leftarrow \min(\beta * \text{BIF}, \mathcal{L} + 2 * \sigma_{\mathcal{L}})$
    sendNRepairSymbols(n)
  end if

In Section 7.5.1, we evaluate QUIRL on our Starlink access. We then validate our results on 200 different network configurations in Section 7.5.2 using Mininet [Han+12].

7.5.1 Improving curl's TCT over Starlink

We analyze the TCT of curl GET and POST requests of short (50kB) and long (10MB) files, with and without FEC. We use the same network setup as described in the beginning of Section 7.4.3. We captured all the packets exchanged during every transfer to analyze the results in details. Ideally, QUIRL should provide significant performance improvement (in the order of magnitude of one RTT) for transfers experiencing losses during their last flight of packets and small to no performance degradation for loss-free transfers. By analyzing the packet captures, we therefore classified the 50kB transfers into three categories: *No loss*, *Losses* and *Last flight losses*. The *Losses* category contains every transfer that experienced at least one packet loss. *Last flight losses* only contains the transfers experiencing at least one packet loss during the last round-trip of packets containing application data.

Figure 7.9 shows the distribution of the transfer completion time for short file transfers. The top graph shows the upload results and the bottom one the download results. We can quickly verify the no-harm approach of QUIRL: there is no noticeable difference for downloads that experienced no loss. The benefits appear along with packet losses, especially when they happen during the last flight. The median upload (resp. download) completion time in the *Last flight losses* category is 600 (resp. 378) milliseconds for QUIRL and 756 (resp. 528) milliseconds for QUIC. The median difference in this category is closely related to the connection's round-trip time for both downloads and uploads. It is therefore easy to see the interest of using QUIRL for short transfers especially for long delay communications as it can significantly reduce the transfer completion time by saving a precious round-trip at the cost of sending a few more packets. On average, QUIRL sent 68 packets for uploads and 63 packets for downloads. QUIC sent 53 packets for uploads and 49 packets for

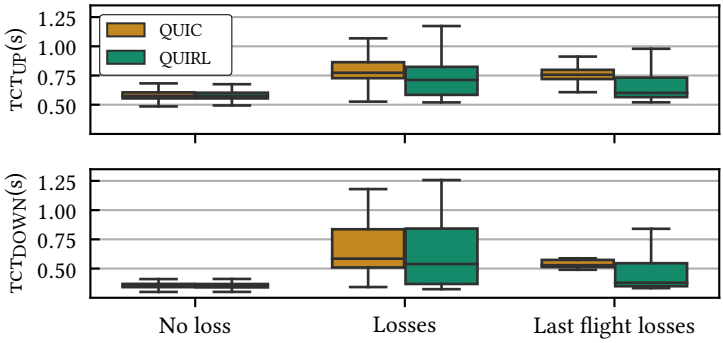


Figure 7.9: Transfer completion times for 50kB files transfers

downloads.

As discussed earlier, the relative impact of tail losses on longer transfers is significantly lower. Figure 7.10 shows the CDF of the TCT of 10MB downloads and uploads using QUIRL and QUIC on our Starlink setup. Since repair symbols are mostly sent at the end of the transfer, the redundancy does not impact negatively the TCT of downloads. On the upload side, we observe a slight improvement of QUIRL with a median upload completion time of 13.012 seconds versus 13.295 seconds for QUIC. A Welch hypothesis test comparing the two upload curves results in a p-value of 0.0054, confirming the difference between these two distributions. This can be explained by the default receive window size of the quiche HTTP/3 server implementation. While the `initial_max_data` QUIC transport parameter is set to 10MB by default, `initial_max_stream_data_uni` governating the amount of bytes that can be sent over an unidirectional stream is set to 1MB. This is not large enough to carry the entire file and requires regular sending of MAX_STREAM_DATA frames by the receiver to update the stream flow control limits. Packet losses may prevent the receiver to empty its receive buffer and subsequently send the flow control updates soon enough to avoid the sender from being flow control-blocked. Sending REPAIR frames when being flow control-blocked helps the receiver to quickly recover lost packets and unblocks the sender sooner. This case is analogous to the buffer-limited transfer scenario explored in FlEC, proving that this scenario can happen in practice. We then performed other experiments manually setting the `initial_max_stream_data_uni` parameter to 10 MB on the server. This solved this flow control problem for QUIC uploads. Since the transfer size and the number of used streams is not known by QUIC at connection establishment, being flow-control blocked on a single stream while still having buffer space on the global connection can happen in the wild, especially for applications relying on the default settings

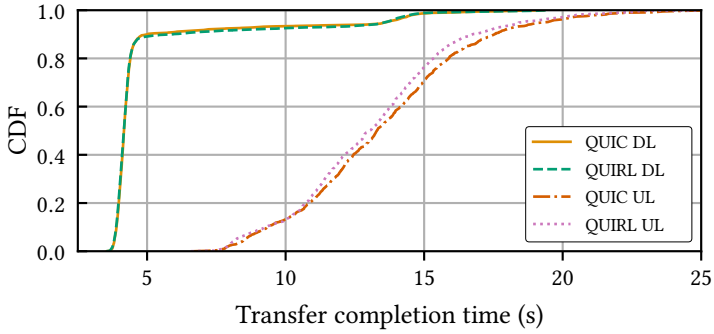


Figure 7.10: TCT distribution for 10MB downloads on our Starlink setup.

of the QUIC implementation. Sending a few REPAIR frames when being flow control-blocked on a single stream can help solving this seamlessly without fine-tuning the per-stream flow control limits of QUIC. On the download side, we do not see much difference between the two distributions, and this is confirmed by the hypothesis test providing a p-value of 0.47 when comparing them. This is because packet losses are less frequent on the download path (we experienced 0.2% of losses during downloads while 1.2% during uploads), making the sender nearly never blocked by flow control. In total, the QUIRL sender added a 3% (resp. 5%) redundancy overhead for 10MB downloads (resp. uploads).

7.5.2 Exploring diverse network configurations with Mininet

After these positive latency results with our Starlink network access, we now explore a wide variety of network configurations using the Mininet network emulation framework. The objective is to study the performance with several delay and bandwidth setups. We also explicitly evaluate QUIRL in the presence of AQM solutions such as FQ-CoDel, as sending more data might cause pressure and cause packet loss when done uncautiously. We do so by following the experimental design approach [Fis49] already used in Chapters 2, 4 and 5. The experimental setup is illustrated by Figure 7.11. The network parameters are randomly chosen from wide ranges of values, defined in Table 7.3. The P_{GB} and P_{BG} are the parameters of the Gilbert loss markov model, already used in Chapter 5. P_{GB} (resp. P_{BG}) is the transition probability from the Good to Bad (resp. Bad to Good) states. The network delay and packet losses are applied using `netem`. AQM is performed using Linux's FQ-CoDel [RFC8290] to prevent abnormal bufferbloat on the network.

To ensure sampling evenly the parameters space with a tractable number of experiments, we rely on the WSP algorithm [SCS12] that samples evenly

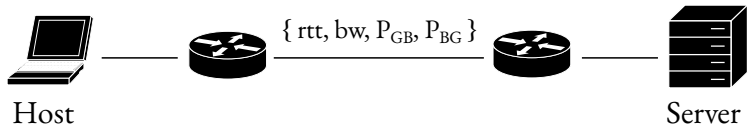


Figure 7.11: Experimental design setup using Mininet.

	rtt	bw	P_{GB}	P_{BG}
min	5 ms	5 Mbps	0	0.2
max	200 ms	100 Mbps	0.02	1

Table 7.3: Values ranges for the experimental design experiments.

the parameters space. This gives 200 distinct network configurations for our Mininet setup. Since the network is symmetric, we focus on downloads, as uploads lead to similar results.

To study more precisely the impact of the loss pattern during the Mininet experiments, we implement and load a modified version of the `netem` Linux kernel module that can be seeded in order to reproduce exactly the same loss pattern for several experiments. For 50kB transfers, QUIRL and QUIC have a very similar traffic pattern. We can therefore directly compare two 50kB experiments being run on the same network configuration and `netem` seed. For each network configuration and each solution, we performed 5 transfers, each with a different seed. For each seed, we computed the ratio between QUIRL and QUIC's TCT. A ratio below 1 (resp. above 1) means that the transfer completed faster with QUIRL (resp. QUIC) for this setup. Figure 7.12 shows the distribution of the TCT ratio for every network configuration and seed. As we can see, experiments with the same network configuration that experienced no packet loss obtain similar results with QUIC and QUIRL as they systematically have a TCT ratio of 1. Improvements can be seen when losses occur, especially upon last flight losses.

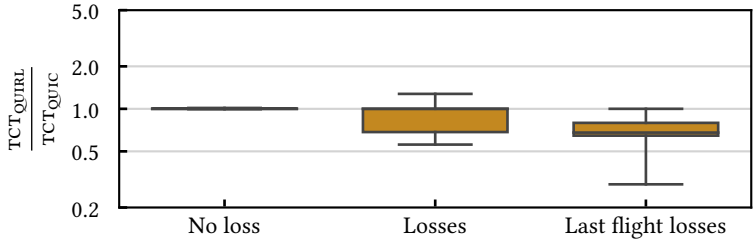


Figure 7.12: TCT ratio for 50kB transfers on our Mininet setup.

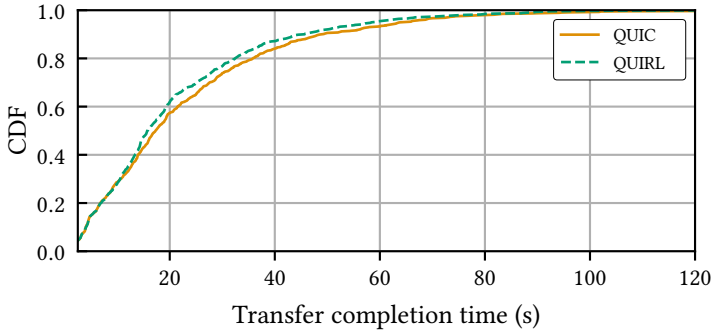


Figure 7.13: TCT distribution for 10MB downloads on our Mininet setup.

Comparing experiments one-by-one for 10MB transfers is more complicated since QUIRL and QUIC's traffic pattern become different on the long run as QUIC does not send REPAIR frames. We therefore analyze the global distribution of the transfer completion time. Similarly to our Starlink measurements in Section 7.5.1, we observe a benefit in using QUIRL for the same flow control reasons. The initial receive window of the curl client is 128 kilobytes, increasing gradually during the transfer. With such an initial receive window, the sender may be flow control-blocked several times during the download upon packet loss and sending repair symbols once no more stream data can be sent helps unblocking the window sooner, providing a significant advantage to QUIRL in many cases.

7.6 Conclusion

In this chapter, we leveraged the lessons learned from QUIC-FEC and FlEC and provided a FEC-enabled loss recovery mechanism for QUIC bringing latency benefits to real applications over real networks. QUIRL allows adapting the redundancy scheduling to both the network loss characteristics and the application's traffic and requirements. We have implemented QUIRL inside Cloudflare's quiche which is already largely deployed on smartphones and servers. We have evaluated QUIRL using both real measurements on Starlink and reproducible experiments on Mininet that cover a wide range of scenarios. Our evaluations using two popular applications demonstrated that QUIRL provides a lower latency than QUIC when losses occur without causing harm in loss-free scenarios. We also showed that QUIRL does not suffer from the performance issues we observed with FlEC. These performance issues were the price to pay for the flexibility of the protocol plugins.

QUIRL integrates the different concepts introduced during the previous chapters. It responds fairly to congestion events by announcing unambigu-

ously symbols recoveries. QUIRL is also adaptive in several ways. It first adapts to the channel conditions by tracking the loss characteristics of the network. Furthermore, its redundancy scheduling algorithm can be adapted to the application needs and traffic pattern. Despite being slightly less flexible than FlEC, QUIRL is the first FEC-enabled QUIC loss recovery mechanism providing actual benefits in real-world scenarios.

Conclusion

8

During this thesis, we incrementally built a FEC-enabled loss recovery mechanism for the QUIC protocol. Each chapter explored different aspects of the solution. Starting from a first prototype sending a fixed amount of redundancy, we took care step by step of new aspects and problematics when adding FEC to this modern multi-purpose transport protocol. We first handled the congestion fairness issues related to the loss recovery mechanism. We then proposed a solution to deploy easily such a heavy extension without having to wait for updates of QUIC clients and servers by their maintainers. We subsequently proposed a version of our FEC-enabled loss recovery mechanism that is adaptive to both the channel characteristics and the application's requirements. Finally, after exploring in details a real lossy network, we developed an efficient and mature implementation of our loss recovery mechanism providing benefits for existing network applications over real networks, implementing different redundancy scheduling behaviours depending on the application's requirements. While this thesis does a comprehensive study of adding of FEC to the QUIC protocol, several aspects remain to be explored in the future.

The most promising future work is the conjoint use of Forward Erasure Correction techniques together with Multipath QUIC [DB17; Liu+23] to stabilize the latency guarantees of the protocol over a set of heterogeneous networks. Multipath allows the FEC-mechanism to go further than simple loss recovery. For instance, some networks such as cellular or Wi-Fi can suffer from an important delay jitter [MB21b]. Even if the delayed packets are not lost, repair symbols can be sent on other network paths to ensure the lowest data delivery delay without the need to duplicate every sent packet on all available paths. Aside from that, the loss events occurring on such networks often cause a small silent period due for instance to Wi-Fi retransmissions on the MAC layer [MB21b]. These loss events are difficult to recover with FEC as the repair symbols will likely arrive after that silent period. The delay induced by Wi-Fi retransmissions therefore cannot be recovered by sending repair symbols on the same path. A multipath protocol enables recovering this kind of losses by sending the repair symbols on another network path.

Now that we implemented a functional extension of our FEC-enabled loss recovery mechanism, we want to explore how new media applications can leverage the concepts developed in this thesis to obtain good latency

guarantees over QUIC. The current best candidate is the MOQT protocol, intended to transport low latency media over QUIC [Law+23]. One approach would be to implement MOQT on top of our QUIRL implementation. Another approach would be to implement the concepts of QUIRL directly inside the MOQT protocol itself.

Additionally, we want to explore how FEC capabilities could be exposed to applications running in web browsers. Providing ways to protect the application data over the WebTransport API would allow simplifying significantly the protocol stack of web browsers. Applications would rely on the QUIC transport protocol only while current media applications currently use WebRTC, which itself needs both RTP and SCTP to implement its full set of features. Relying on QUIC would also provide a secure transport protocol by design to these applications.

The QUIC protocol has also been considered as a good candidate for ensuring modern multicast communications [HPF22; NMB22]. Compared to SR-ARQ, the use of FEC can bring significant benefits to multicast communications as repair symbols make no assumption on the position of the lost packets. Several multicast receivers can therefore encounter packet losses on different packets and still recover these packets using the same repair symbols sent by the source.

Bibliography

[21] *Quiche*. https://quiche.googlesource.com/quiche/+/refs/heads/main, file quic/tools/quic_client_base.cc. commit: 98966fd9b7183bcdb42ce78e58be40bcf6d68493. 2021.

[22a] *Browsertime*. 2022. URL: https://www.sitespeed.io/documentation/browsertime/ (visited on 05/13/2022).

[22b] *Disney+ - Internet speed recommendations*. 2022. URL: https://help.disneyplus.com/csp?id=csp_article_content&sys_kb_id=bb07d3cd1b8d0010b8651f861a4bcbfd (visited on 05/13/2022).

[22c] *Netflix - Internet connection speed recommendations*. 2022. URL: https://help.netflix.com/en/node/306 (visited on 05/13/2022).

[22d] *NVIDIA GeForce Now System Requirements*. 2022. URL: https://www.nvidia.com/it-it/geforce-now/system-reqs (visited on 05/13/2022).

[22e] *RIPE Atlas*. 2022. URL: https://atlas.ripe.net/ (visited on 05/13/2022).

[22f] *Space.com - SpaceX lofts 49 Starlink internet satellites to orbit in 1st launch of 2022*. 2022. URL: https://www.space.com/spacex-starlink-launch-success-january-2022 (visited on 05/13/2022).

[22g] *Speed Index*. 2022. URL: https://web.dev/speed-index/ (visited on 05/13/2022).

[22h] *Speedtest CLI*. 2022. URL: https://www.speedtest.net/it/apps/cli (visited on 05/13/2022).

[22i] *Starlink*. 2022. URL: https://www.starlink.com/ (visited on 05/13/2022).

[22j] *Tixeo, secure video conferencing*. https://www.tixeo.com/. 2022.

[22k] *Website Traffic Analysis & Competitive Intelligence, SimilarWeb*. 2022. URL: https://www.similarweb.com/ (visited on 05/13/2022).

[23] "FFmpeg". In: (2023). http://www.ffmpeg.org.

[3GP19] 3GPP. *Release description; Release 15*. Technical Report (TR) 21.915. Version 15.0.0. 3rd Generation Partnership Project (3GPP), Oct. 2019.

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

[al22] M. S. et al. "A QUIC implementation in pure go." https://github.com/quic-go/quic-go. 2022.

[App18] Apple. "Improving Network Reliability Using Multipath TCP". https://developer.apple.com/documentation/foundation/urlsessionconfiguration/improving_network_reliability_using_multipath_tcp. 2018.

[Aqe+20] W. Aqeel, B. Chandrasekaran, A. Feldmann, and B. M. Maggs. "On landing and internal web pages: The strange case of jekyll and hyde in web performance measurement". In: *Proceedings of the ACM Internet Measurement Conference*. 2020, pp. 680–695.

[ARS16] M. Agiwal, A. Roy, and N. Saxena. "Next generation 5G wireless networks: A comprehensive survey". In: *IEEE Communications Surveys & Tutorials* 18.3 (2016), pp. 1617–1655.

[Ast+22] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers. "The Prusti Project: Formal Verification for Rust (invited)". In: *NASA Formal Methods (14th International Symposium)*. Springer, 2022, pp. 88–108.

[ATT] "AT&T". *Fixed Wireless Internet*. https://www.att.com/internet/fixed-wireless/.

[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.

[Bal+22] A. Baltaci, H. Cech, N. Mohan, F. Geyer, V. Bajpai, J. Ott, and D. Schupke. "Analyzing real-time video delivery over cellular networks for remote piloting aerial vehicles". In: *Proceedings of the 22nd ACM Internet Measurement Conference*. 2022, pp. 98–112.

[BBK17] K. Bhargavan, B. Blanchet, and N. Kobeissi. "Verified models and reference implementations for the TLS 1.3 standard candidate". In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 483–502.

[Bel+] M. Belshe et al. *SPDY protocol*. `https://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3-1`. Accessed: 2022-02-09.

[Bie93] E. W. Biersack. "Performance evaluation of forward error correction in an ATM environment". In: *IEEE JSAC* 11.4 (1993), pp. 631–640.

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

[BK23] M. Brockschmidt and H. Khlaaf. *T2 Temporal Prover*. `http://mmjb.github.io/T2/`. Jan. 2023.

[BLK04] L. Baldantoni, H. Lundqvist, and G. Karlsson. "Adaptive end-to-end FEC for improving TCP performance over wireless links". In: *2004 IEEE International Conference on Communications (IEEE Cat. No. 04CH37577)*. Vol. 7. IEEE. 2004, pp. 4023–4027.

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

[BOP94] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. "TCP Vegas: New techniques for congestion detection and avoidance". In: *Proceedings of the conference on Communications architectures, protocols and applications*. 1994, pp. 24–35.

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

[Bye+98] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. "A digital fountain approach to reliable distribution of bulk data". In: *ACM SIGCOMM Computer Communication Review* 28.4 (1998), pp. 56–67.

[Cam+14] D. Camara, H. Tazaki, E. Mancini, T. Turletti, W. Dabbous, and M. Lacage. "DCE: Test the real code of your protocols and applications over simulated networks". In: *IEEE Communications Magazine* 52.3 (2014), pp. 104–110.

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

[Car+16b] G. Carlucci, L. De Cicco, S. Holmer, and S. Mascolo. "Analysis and design of the google congestion control for web real-time communication (WebRTC)". In: *Proceedings of the 7th International Conference on Multimedia Systems*. 2016, pp. 1–12.

[Car+17] N. Cardwell, Y. Cheng, S. H. Yeganeh, and V. Jacobson. *BBR Congestion Control*. Internet-Draft draft-cardwell-iccrg-bbr-congestion-control-00. Work in Progress. Internet Engineering Task Force, 2017. 34 pp.

[Car+22] N. Cardwell, Y. Cheng, S. H. Yeganeh, I. Swett, and V. Jacobson. *BBR Congestion Control*. Internet-Draft draft-cardwell-iccrg-bbr-congestion-control-02. Work in Progress. Internet Engineering Task Force, Mar. 2022. 66 pp.

[CGK10] P. Chini, G. Giambene, and S. Kota. "A survey on mobile satellite systems". In: *International Journal of Satellite Communications and Networking* 28.1 (2010), pp. 29–57.

[Chu+18] A. Chudnov, N. Collins, B. Cook, J. Dodds, B. Huffman, C. MacCárthaigh, S. Magill, E. Mertens, E. Mullen, T. Serdar, A. Tomb, and E. Westbrook. "Continuous formal ffverification of Amazon s2n". In: *International Conference on Computer Aided Verification*. Springer. 2018, pp. 430–446.

[CLM15] J. Cloud, D. Leith, and M. Médard. "A coded generalization of selective repeat ARQ". In: *INFOCOM 2015*. IEEE. 2015, pp. 2155–2163.

[Clo+13] J. Cloud et al. "Multi-path TCP with network coding for mobile devices in heterogeneous networks". In: *VTC, 2013 IEEE*. IEEE. 2013, pp. 1–5.

[Clo22] Cloudflare. "quiche". `https://github.com/cloudflare/quiche`. 2022.

[CM19] R. Chandra and T. Moscibroda. "Perspective: White space networking with Wi-Fi like connectivity". In: *ACM SIGCOMM Computer Communication Review* 49.5 (2019), pp. 107–109.

[Coh+20] A. Cohen, D. Malak, V. B. Bracha, and M. Médard. "Adaptive Causal Network Coding With Feedback". In: *IEEE Transactions on Communications* 68.7 (2020), pp. 4325–4341. DOI: 10.1109/TCOMM.2020.2989827.

[CP07] P. Chaporkar and A. Proutiere. "Adaptive network coding and scheduling for maximizing throughput in wireless networks". In: *MobiCom 2007*. ACM. 2007, pp. 135–146.

[CPR06] B. Cook, A. Podelski, and A. Rybalchenko. "TERMINATOR: beyond safety". In: *International Conference on Computer Aided Verification*. Springer. 2006, pp. 415–418.

[CSA03] B. Cavusoglu, D. Schonfeld, and R. Ansari. "Real-time adaptive forward error correction for MPEG-2 video communications over RTP networks". In: *2003 International Conference on Multimedia and Expo. ICME'03. Proceedings (Cat. No. 03TH8698)*. Vol. 3. IEEE. 2003, pp. III–261.

[Cui+14] Y. Cui, L. Wang, X. Wang, H. Wang, and Y. Wang. "FMTCP: A fountain code-based multipath transmission control protocol". In: *IEEE/ACM Transactions on Networking* 23.2 (2014), pp. 465–478.

[Cur+23] L. Curley, K. Pugin, S. Nandakumar, and V. Vasiliev. *Media over QUIC Transport*. Internet-Draft draft-lcurley-moq-transport-00. Work in Progress. Internet Engineering Task Force, May 2023. 26 pp.

[Curl] *Curl, command line tool and library for transferring data with URLs*. `https://curl.se/`. Accessed: 2023-04-12.

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

[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*. 2013, pp. 1–8.

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

[ED19] K. Edeline and B. Donnet. "A bottom-up investigation of the transport-layer ossification". In: *2019 Network Traffic Measurement and Analysis Conference (TMA)*. IEEE. 2019, pp. 169–176.

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

[Ege+11] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. “PiOS: Detecting Privacy Leaks in iOS Applications”. In: *Network and Distributed System Security Symposium (NDSS’11)*. 2011, pp. 177–183.

[Egg20] L. Eggert. *Towards Securing the Internet of Things with QUIC*. Tech. rep. EasyChair, 2020.

[Ell63] E. O. Elliott. “Estimates of error rates for codes on burst-noise channels”. In: *The Bell System Technical Journal* 42.5 (1963), pp. 1977–1997.

[Fer+18] S. Ferlin, S. Kucera, H. Claussen, and Ö. Alay. “MPTCP meets FEC: Supporting latency-sensitive applications over heterogeneous networks”. In: *IEEE/ACM Transactions on Networking* 26.5 (2018), pp. 2005–2018.

[FHK18] A. Frömmgen, J. Heuschkel, and B. Koldehofe. “Multipath TCP scheduling for thin streams: Active probing and one-way delay-awareness”. In: *2018 IEEE International Conference on Communications (ICC)*. IEEE. 2018, pp. 1–7.

[Fis49] R. A. Fisher. *The design of experiments*. Oliver and Boyd, 1949.

[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.

[FKV23] A. Frindell, E. Kinnear, and V. Vasiliev. *WebTransport over HTTP/3*. Internet-Draft draft-ietf-webtrans-http3-04. Work in Progress. Internet Engineering Task Force, Jan. 2023. 16 pp.

[FL20] N. Feamster and J. Livingood. “Measuring internet speed: current challenges and future recommendations”. In: *Communications of the ACM* 63.12 (2020), pp. 72–80.

[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: *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*. 2013, pp. 159–170.

[Fle17a] M. Fleming. “A thorough introduction to eBPF”. In: (2017). `https://lwn.net/Articles/740157/`.

[Fle17b] M. Fleming. “A thorough introduction to eBPF”. In: *Linux Weekly News* (Dec. 2017). https://old.lwn.net/Articles/740157/.

[FLW06] C. Fragouli, J.-Y. Le Boudec, and J. Widmer. “Network coding: an instant primer”. In: *ACM SIGCOMM Computer Communication Review* 36.1 (2006), pp. 63–68.

[For16] ". Forum". *"TR-348 Hybrid Access Broadband Network Architecture"*. Aug. 2016.

[Fuk11] K. Fukuda. “An analysis of longitudinal TCP passive measurements (short paper)”. In: *Traffic Monitoring and Analysis: Third International Workshop, TMA 2011, Vienna, Austria, April 27, 2011. Proceedings 3*. Springer. 2011, pp. 29–36.

[Gar+19] P. Garrido, I. Sanchez, S. Ferlin, R. Aguero, and O. Alay. “rQUIC: Integrating FEC with QUIC for robust wireless communications”. In: *2019 IEEE Global Communications Conference (GLOBECOM)*. IEEE. 2019, pp. 1–7.

[Geo] A. B. Geoff Houston. *A look at QUIC use*. `https://stats.labs.apnic.net/quic`. Accessed: 2022-02-09.

[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.

[Gie+18] H. Giesen et al. “In-network computing to the rescue of faulty links”. In: *NetCompute 2018*. ACM. 2018, pp. 1–6.

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

[GStrDoc] G. Documentation. *GStreamer rtpjitterbuffer*. `https://gstreamer.freedesktop.org/documentation/rtpmanager/rtpjitterbuffer.html`. Accessed: 2023-04-27.

[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 et al. “Reproducible network experiments using container-based emulation”. In: *CoNEXT*. ACM. 2012, pp. 253–264.

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

[HH08] G. Haßlinger and O. Hohlfeld. "The Gilbert-Elliott model for packet loss in real time services on the Internet". In: *14th GI/ITG Conference-Measurement, Modelling and Evalutation of Computer and Communication Systems*. VDE. 2008, pp. 1–15.

[Ho+03] T. Ho, R. Koetter, M. Médard, D. R. Karger, and M. Effros. "The benefits of coding over routing in a randomized setting". In: (2003).

[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*. 2011, pp. 181–194.

[Hon+14] M. Honda, F. Huici, C. Raiciu, J. Araujo, and L. Rizzo. "Rekindling network protocol innovation with user-level stacks". In: *ACM SIGCOMM Computer Communication Review* 44.2 (2014), pp. 52–58.

[Hor+18] D. N. da Hora, A. S. Asrese, V. Christophides, R. Teixeira, and D. Rossi. "Narrowing the gap between QoS metrics and Web QoE using Above-the-fold metrics". In: *International Conference on Passive and Active Network Measurement*. Springer. 2018, pp. 31–43.

[Hou+08] I.-H. Hou, Y.-E. Tsai, T. F. Abdelzaher, and I. Gupta. "Adapcode: Adaptive network coding for code updates in wireless sensor networks". In: *IEEE INFOCOM 2008*. IEEE. 2008, pp. 1517–1525.

[HPF22] J. Holland, L. Pardue, and M. Franke. *Multicast Extension for QUIC*. Internet-Draft draft-jholland-quic-multicast-02. Work in Progress. Internet Engineering Task Force, July 2022. 38 pp.

[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 (2008), pp. 64–74.

[Hua+13] J. Huang, F. Qian, Y. Guo, Y. Zhou, Q. Xu, Z. M. Mao, S. Sen, and O. Spatscheck. "An in-depth study of LTE: Effect of network protocol and application behavior on performance". In: *ACM SIGCOMM Computer Communication Review* 43.4 (2013), pp. 363–374.

[Hui+21] C. Huitema et al. *Minimal implementation of the QUIC protocol*. `https://github.com/private-octopus/picoquic/blob/master/picoquic/quicctx.c`. commit: 7f49f62ff7f3938eb1a0f49dfc551d7ed189454c. 2021.

[Hui22a] C. Huitema. “picoquic”. `https://github.com/private-octopus/picoquic`. 2022.

[Hui22b] C. Huitema. *Quic Timestamps For Measuring One-Way Delays*. Internet-Draft draft-huitema-quic-ts-08. Work in Progress. Internet Engineering Task Force, Aug. 2022. 11 pp.

[Hui97] C. Huitema. “The case for packet level FEC”. In: *Protocols for High-Speed Networks V: TC6 WG6. 1/6.4 Fifth International Workshop on Protocols for High-Speed Networks (PfHSN’96) 28–30 October 1996, Sophia Antipolis, France*. Springer. 1997, pp. 109–120.

[IO 18] IO Visor Project. “Userspace eBPF VM”. `https://github.com/iovisor/ubpf`. 2018.

[IS18] J. Iyengar and I. Swett. “QUIC: Developing and Deploying a TCP Replacement for the Web”. In: *Netdev 0x12*. 2018.

[ISO22] ISO Central Secretary. *Information technology – Dynamic adaptive streaming over HTTP (DASH) – Part 1: Media presentation description and segment formats*. Standard ISO/IEC TR 23009-1:2022. International Organization for Standardization, 2022.

[IT16] J. Iyengar and M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Internet-Draft draft-ietf-quic-transport-00. Work in Progress. Internet Engineering Task Force, Nov. 2016. 45 pp.

[ITU03] ITU-T. *Recommendation G.114: One-way transmission time*. Tech. rep. 2003.

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

[Jae+23] B. Jaeger, J. Zirngibl, M. Kempf, K. Ploch, and G. Carle. “QUIC on the Highway: Evaluating Performance on High-rate Links”. In: *International Federation for Information Processing (IFIP) Networking 2023 Conference (IFIP Networking 2023)*. 2023.

[Jai86] R. Jain. “A timeout-based congestion control scheme for window flow-controlled networks”. In: *IEEE Journal on Selected Areas in Communications* 4.7 (1986), pp. 1162–1167.

[JR88] R. Jain and K. Ramakrishnan. “Congestion avoidance in computer networks with a connectionless network layer: concepts, goals and methodology”. In: *[1988] Proceedings. Computer Networking Symposium*. IEEE. 1988, pp. 134–143.

[Kas+20] S. Kassing, D. Bhattacherjee, A. B. Águas, J. E. Saethre, and A. Singla. “Exploring the" Internet from space" with Hypatia”. In: *Proceedings of the ACM Internet Measurement Conference*. 2020, pp. 214–229.

[Kas+22] M. M. Kassem, A. Raman, D. Perino, and N. Sastry. “A browser-side view of starlink connectivity”. In: *Proceedings of the 22nd ACM Internet Measurement Conference*. 2022, pp. 151–158.

[Kat+08] S. Katti et al. “XORs in the air: practical wireless network coding”. In: *IEEE/ACM ToN* 16.3 (2008), pp. 497–510.

[KD19] P. Kumar and B. Dezfouli. “Implementation and analysis of QUIC for MQTT”. In: *Computer Networks* 150 (2019), pp. 28–45.

[Ken+16] J. Keniston et al. *Kernel probes (kprobes)*. Documentation provided with the Linux kernel sources (v2. 6.29). 2016.

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

[Kha+17] A. S. Khatouni, M. Mellia, M. A. Marsan, S. Alfredsson, J. Karlsson, A. Brunstrom, O. Alay, A. Lutu, C. Midoglu, and V. Mancuso. “Speedtest-like measurements in 3g/4g networks: The monroe experience”. In: *2017 29th International Teletraffic Congress (ITC 29)*. Vol. 1. IEEE. 2017, pp. 169–177.

[KHB20] N. Keukeleire, B. Hesmans, and O. Bonaventure. “Increasing broadband reach with hybrid access networks”. In: *IEEE Communications Standards Magazine* 4.1 (2020), pp. 43–49.

[Khl+15] H. Khlaaf, M. Brockschmidt, S. Falke, D. Kapur, and C. Sinz. *llvm2KITTeL tailored for T2*. Source code. `https://github.com/hkhlaaf/llvm2kittel`. 2015.

[Kim+12] M. Kim, J. Cloud, A. ParandehGheibi, L. Urbina, K. Fouli, D. Leith, and M. Médard. “Network coded TCP (CTCP)”. In: *arXiv preprint arXiv:1212.2291* (2012).

[Kim+14] M. Kim et al. “Congestion control for coded transport layers”. In: *Communications (ICC), 2014 IEEE International Conference on*. IEEE. 2014, pp. 1228–1234.

[Kod+20] O. Kodheli, E. Lagunas, N. Maturo, S. K. Sharma, B. Shankar, J. F. M. Montoya, J. C. M. Duncan, D. Spano, S. Chatzinotas, S. Kisseleff, et al. "Satellite communications in the new space era: A survey and future challenges". In: *IEEE Communications Surveys & Tutorials* 23.1 (2020), pp. 70–109.

[Kos+22] M. Kosek, T. V. Doan, M. Granderath, and V. Bajpai. "One to Rule Them All? A First Look at DNS over QUIC". In: *Passive and Active Measurement: 23rd International Conference, PAM 2022, Virtual Event, March 28–30, 2022, Proceedings*. Springer. 2022, pp. 537–551.

[Kuh+18] N. Kuhn et al. *Network coding and satellites*. Working Draft. Internet-Draft. draft-irtf-nwcrg-network-coding-satellites-02. Nov. 2018.

[Kuh+20] N. Kuhn, G. Fairhurst, J. Border, and S. Emile. *QUIC for SATCOM*. Internet-Draft draft-kuhn-quic-4-sat-03. `http://www.ietf.org/internet-drafts/draft-kuhn-quic-4-sat-03.txt`. IETF Secretariat, Jan. 2020.

[Lab] A. Labs. *HTTP/3 Use by country*. `https://stats.labs.apnic.net/quic`. Accessed: 2023-04-07.

[Lan+17] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, et al. "The QUIC transport protocol: Design and internet-scale deployment". In: *Proceedings of the conference of the ACM special interest group on data communication*. 2017, pp. 183–196.

[Law+23] W. Law, L. Curley, V. Vasiliev, S. Nandakumar, and K. Pugin. *WARP Streaming Format*. Internet-Draft draft-law-moq-warpstreamingformat-00. Work in Progress. Internet Engineering Task Force, June 2023. 8 pp.

[LCB10] D. Lee, B. E. Carpenter, and N. Brownlee. "Observations of UDP to TCP ratio and port numbers". In: *2010 Fifth International Conference on Internet Monitoring and Protection*. IEEE. 2010, pp. 99–104.

[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+19] F. Li, A. A. Niaki, D. Choffnes, P. Gill, and A. Mislove. "A large-scale analysis of deployed traffic differentiation practices". In: *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 130–144.

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

[LinBBR] *Linux source code git repository*. `https://git.kernel.org`. commit: 0f8782ea14974ce992618b55f0c041ef43ed0b78.

[LinCub] *Linux source code git repository*. `https://git.kernel.org`. commit: 597811ec167fa01926a0957a91d9e39baa30e64.

[Liu+23] Y. Liu, Y. Ma, Q. D. Coninck, O. Bonaventure, C. Huitema, and M. Kühlewind. *Multipath Extension for QUIC*. Internet-Draft draft-ietf-quic-multipath-04. Work in Progress. Internet Engineering Task Force, Mar. 2023. 30 pp.

[LK04] H. Lundqvist and G. Karlsson. "TCP with end-to-end FEC". In: *International Zurich Seminar on Communications, 2004*. IEEE. 2004, pp. 152–155.

[LLV23] LLVM Team. "Clang: a C language family frontend for LLVM". https://clang.llvm.org/. 2023.

[Lub02] M. Luby. "LT codes". In: *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.* IEEE Computer Society. 2002, pp. 271–271.

[M K+22] M. M. Kassem, A. Raman, D. Perino, and N. Sastry. "A Browser-side View of Starlink Connectivity". In: *Proceedings of the 2022 Internet Measurement Conference*. 2022. DOI: 10.1145/3517745.3561457.

[Ma+22] S. Ma, Y. C. Chou, H. Zhao, L. Chen, X. Ma, and J. Liu. "Network Characteristics of LEO Satellite Constellations: A Starlink-Based Measurement from End Users". In: *arXiv preprint arXiv : 2212.13697* (2022).

[Mac+] K. MacMillan, T. Mangla, J. Saxon, N. P. Marwell, and N. Feamster. "A Comparative Analysis of Ookla Speedtest and Measurement Labs Network Diagnostic Test (NDT7)". In: ().

[Mas+01] S. Mascolo et al. "TCP westwood: Bandwidth estimation for enhanced transport over wireless links". In: *MobiCom*. ACM. 2001, pp. 287–297.

[MB21a] F. Michel and O. Bonaventure. "Packet delivery time as a tie-breaker for assessing Wi-Fi access points". In: *IAB Workshop on Measuring Network Quality for End-Users* (2021).

[MB22] F. Michel and O. Bonaventure. *Forward Erasure Correction for QUIC loss recovery*. Internet-Draft draft-michel-quic-fec-00. Work in Progress. Internet Engineering Task Force, Oct. 2022. 14 pp.

[MDB18] F. Michel, Q. De Coninck, and O. Bonaventure. "Adding Forward Erasure Correction to QUIC". In: *arXiv preprint arXiv : 1809.04822* (2018).

[Mic+23b] F. Michel, M. Trevisan, D. Giordano, and O. Bonaventure. "A first look at Starlink performance: open data". `https://smartdata.polito.it/a-first-look-at-starlink-performance-open-data/`. 2023.

[Mic23a] F. Michel. "ebpf_dropper". `https://github.com/francoismichel/ebpf_dropper`. 2023.

[Mic23b] F. Michel. "FlEC". `https://github.com/francoismichel/flec`. 2023.

[Mic23c] F. Michel. "FlEC simulations experiments". `https://github.com/francoismichel/flec-simulation-experiments`. 2023.

[Mic23d] F. Michel. "QUIC-FEC on Bitbucket". `https://bitbucket.org/michelfra/quic-fec`. 2023.

[Mic23e] F. Michel. "QUIRL". `https://github.com/francoismichel/quirl`. 2023.

[Mil] L. Milne-Thompson. *The calculus of finite differences 1951, 171*. Macmillan, London, p. 9.

[Mor12] P. Morandi. *Field and Galois theory*. Vol. 167. Springer Science & Business Media, 2012.

[Nec02] G. C. Necula. "Proof-carrying code. Design and implementation". In: *Proof and system-reliability*. Springer, 2002, pp. 261–288.

[NMB22] L. Navarre, F. Michel, and O. Bonaventure. "It Is Time to Reconsider Multicast". In: *IAB workshop on Environmental Impact of Internet Applications and Systems, 2022*. 2022.

[NTM08] A. Nafaa, T. Taleb, and L. Murphy. "Forward error correction strategies for media streaming over wireless networks". In: *IEEE Communications Magazine* 46.1 (2008), pp. 72–79.

[OE23] J. Ott and M. Engelbart. *RTP over QUIC*. Internet-Draft draft-ietf-avtcore-rtp-over-quic-02. Work in Progress. Internet Engineering Task Force, Feb. 2023. 27 pp.

[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.

[Pau+23] T. Pauly, B. Trammell, A. Brunstrom, G. Fairhurst, and C. Perkins. *An Architecture for Transport Services*. Internet-Draft draft-ietf-taps-arch-18. Work in Progress. Internet Engineering Task Force, May 2023. 33 pp.

[PDB18] 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*. 2018, pp. 8–14.

[Per+22] D. Perdices, G. Perna, M. Trevisan, D. Giordano, and M. Mellia. "When Satellite is All You Have When Satellite is All You Have: Watching the Internet from 550 ms". In: *Proceedings of the 2022 Internet Measurement Conference*. 2022. DOI: 10.1145/3517745.3561432.

[PO18] C. Perkins and J. Ott. "Real-time audio-visual media transport over QUIC". In: *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. 2018, pp. 36–42.

[PquicRepo] *Pluginized QUIC*. `https://github.com/p-quic/pquic`. commit: 68e61c5496d8d3ef9b39e7bd5d60a14b9789e977.

[PR04] A. Podelski and A. Rybalchenko. "Transition invariants". In: *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.* IEEE. 2004, pp. 32–41.

[PR05] A. Podelski and A. Rybalchenko. "Transition predicate abstraction and fair termination". In: *ACM SIGPLAN Notices* 40.1 (2005), pp. 132–144.

[Pro+19] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan. "Formally verified cryptographic web applications in webassembly". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 1256–1274.

[Pur+01] R. Puri, K. Ramchandran, K.-W. Lee, and V. Bharghavan. "Forward error correction (FEC) codes based multiple description coding for Internet video streaming and multicast". In: *Signal Processing: Image Communication* 16.8 (2001), pp. 745–762.

[QUICBlog] *Experimenting with QUIC*. https://blog.chromium.org/2013/06/experimenting-with-quic.html. accessed: 2022-02-09.

[QUICDNS] *DNS-over-HTTP/3 in Android*. https://security.googleblog.com/2022/07/dns-over-http3-in-android.html. accessed: 2022-02-09.

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

[Raj+19] M. Rajiullah, A. Lutu, A. S. Khatouni, M.-R. Fida, M. Mellia, A. Brunstrom, O. Alay, S. Alfredsson, and V. Mancuso. "Web experience in mobile networks: Lessons from two million page visits". In: *The world wide web conference*. 2019, pp. 1532–1543.

[Rao+23] X. Rao, A. L. Georges, M. Legoupil, C. Watt, J. Pichon-Pharabod, P. Gardner, and L. Birkedal. "Iris-Wasm: Robust and Modular Verification of WebAssembly Programs". In: *Proceedings of the 44th ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2023). Association for Computing Machinery*. 2023.

[Rap+13] T. S. Rappaport, S. Sun, R. Mayzus, H. Zhao, Y. Azar, K. Wang, G. N. Wong, J. K. Schulz, M. Samimi, and F. Gutierrez. "Millimeter wave mobile communications for 5G cellular: It will work!" In: *IEEE access* 1 (2013), pp. 335–349.

[RBC16] J. P. Rula, F. E. Bustamante, and D. R. Choffnes. "When IPs Fly: A Case for Redefining Airline Communication". In: *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications*. ACM. 2016, pp. 9–14.

[RFB01] K. Ramakrishnan, S. Floyd, and D. Black. *RFC3168: The addition of explicit congestion notification (ECN) to IP*. Tech. rep. 2001.

[RFC1323] D. A. Borman, R. T. Braden, and V. Jacobson. *TCP Extensions for High Performance*. RFC 1323. May 1992. DOI: 10.17487/RFC1323.

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

[RFC2883] M. Podolsky, S. Floyd, J. Mahdavi, and M. Mathis. *An Extension to the Selective Acknowledgement (SACK) Option for TCP*. RFC 2883. July 2000. DOI: 10.17487/RFC2883.

[RFC3135] J. Griner, J. Border, M. Kojo, Z. D. Shelby, and G. Montenegro. *Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations*. RFC 3135. June 2001. DOI: 10.17487/RFC3135.

[RFC3168] S. Floyd, D. K. K. Ramakrishnan, and D. L. Black. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168. Sept. 2001. DOI: 10.17487/RFC3168.

[RFC3708] E. Blanton and M. Allman. *Using TCP Duplicate Selective Acknowledgement (DSACKs) and Stream Control Transmission Protocol (SCTP) Duplicate Transmission Sequence Numbers (TSNs) to Detect Spurious Retransmissions*. RFC3708. Feb. 2004. DOI: 10.17487/RFC3708.

[RFC4960] R. R. Stewart. *Stream Control Transmission Protocol*. RFC 4960. Sept. 2007. DOI: 10.17487/RFC4960.

[RFC5681] E. Blanton, D. V. Paxson, and M. Allman. *TCP Congestion Control*. RFC 5681. Sept. 2009. DOI: 10.17487/RFC5681.

[RFC6184] R. Jesup, T. Kristensen, Y. Wang, and R. Even. *RTP Payload Format for H.264 Video*. RFC 6184. May 2011. DOI: 10.17487/RFC6184.

[RFC6363] V. Roca, M. Watson, and A. C. Begen. *Forward Error Correction (FEC) Framework*. RFC 6363. Oct. 2011.

[RFC6582] A. Gurtov, T. Henderson, S. Floyd, and Y. Nishida. *The NewReno Modification to TCP's Fast Recovery Algorithm*. RFC 6582. Apr. 2012. DOI: 10.17487/RFC6582.

[RFC6675] E. Blanton, M. Allman, L. Wang, I. Järvinen, M. Kojo, and Y. Nishida. *A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP*. RFC 6675. Aug. 2012. DOI: 10.17487/RFC6675.

[RFC6816] V. Roca, M. Cunche, and J. Lacan. *Simple Low-Density Parity Check (LDPC) Staircase Forward Error Correction (FEC) Scheme for FECFRAME*. RFC 6816. Dec. 2012. DOI: 10.17487/RFC6816.

[RFC6865] V. Roca, M. Cunche, J. Lacan, A. Bouabdallah, and K. Matsuzono. *Simple Reed-Solomon Forward Error Correction (FEC) Scheme for FECFRAME*. RFC 6865. Feb. 2013. DOI: 10.17487/RFC6865.

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

[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. T. Braden, W. Eddy, E. Blanton, and A. Zimmermann. *A Roadmap for Transmission Control Protocol (TCP) Specification Documents*. RFC 7414. Feb. 2015. DOI: 10.17487/RFC7414.

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

[RFC791] *Internet Protocol*. RFC 791. Sept. 1981. DOI: 10.17487/RFC0791.

[RFC8200] D. S. E. Deering and B. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*. RFC 8200. July 2017. DOI: 10.17487/RFC8200.

[RFC8290] T. Høiland-Jørgensen, P. McKenney, D. Taht, J. Gettys, and E. Dumazet. *The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm*. RFC 8290. Jan. 2018. DOI: 10.17487/RFC8290.

[RFC8298] I. Johansson and Z. Sarker. *Self-Clocked Rate Adaptation for Multimedia*. RFC 8298. Dec. 2017. DOI: 10.17487/RFC8298.

[RFC8312bis] L. Xu, S. Ha, I. Rhee, V. Goel, and L. Eggert. *CUBIC for Fast and Long-Distance Networks*. Internet-Draft draft-ietf-tcpm-rfc8312bis-15. Work in Progress. Internet Engineering Task Force, Jan. 2023. 41 pp.

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

[RFC8681] V. Roca and B. Teibi. *Sliding Window Random Linear Code (RLC) Forward Erasure Correction (FEC) Schemes for FECFRAME*. RFC 8681. Jan. 2020. DOI: 10.17487/RFC8681.

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

[RFC8825] H. T. Alvestrand. *Overview: Real-Time Protocols for Browser-Based Applications*. RFC 8825. Jan. 2021. DOI: 10.17487/RFC8825.

[RFC8985] Y. Cheng, N. Cardwell, N. Dukkipati, and P. Jha. *The RACK-TLP Loss Detection Algorithm for TCP*. RFC 8985. Feb. 2021. DOI: 10.17487/RFC8985.

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

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

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

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

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

[RFC9220] R. Hamilton. *Bootstrapping WebSockets with HTTP/3*. RFC 9220. June 2022. DOI: 10.17487/RFC9220.

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

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

[RH10] G. F. Riley and T. R. Henderson. "The ns-3 network simulator". In: *Modeling and tools for network simulation*. Springer, 2010, pp. 15–34.

[Roc+17] V. Roca et al. "Less latency and better protection with AL-FEC sliding window codes: A robust multimedia CBR broadcast case study". In: *WiMob*. IEEE. 2017, pp. 1–8.

[Ros12] J. Rosking. *QUIC: Design Document and Specification Rationale*. https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34. accessed: 2022-02-09. 2012.

[RS60] I. S. Reed and G. Solomon. "Polynomial codes over certain finite fields". In: *Journal of the society for industrial and applied mathematics* 8.2 (1960), pp. 300–304.

[Rul+18] Rula et al. "Mile High WiFi: A First Look at In-Flight Internet Connectivity". In: *2018 World Wide Web Conference on World Wide Web*. 2018.

[Rüt+18] J. Rüth, I. Poese, C. Dietzel, and O. Hohlfeld. "A First Look at QUIC in the Wild". In: *Passive and Active Measurement: 19th International Conference, PAM 2018, Berlin, Germany, March 26–27, 2018, Proceedings 19*. Springer. 2018, pp. 255–268.

[RX05] I. Rhee and L. Xu. "CUBIC: a new TCP-friendly high-speed TCP variant". In: *Proceedings of the third PFLDNet Workshop* (2005).

[Ryb+21] N. Rybowski, Q. De Coninck, T. Rousseaux, A. Legay, and O. Bonaventure. "Implementing the plugin distribution system". In: *Proceedings of the SIGCOMM'21 Poster and Demo Sessions*. 2021, pp. 39–41.

[Sch+13] P. S. Schmidt, T. Enghardt, R. Khalili, and A. Feldmann. "Socket intents: Leveraging application awareness for multi-access connectivity". In: *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. 2013, pp. 295–300.

[Sch23] D. Schinazi. *The MASQUE Proxy*. Internet-Draft draft-schinazi-masque-proxy-00. Work in Progress. Internet Engineering Task Force, Mar. 2023. 7 pp.

[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.

[Sho06] A. Shokrollahi. "Raptor codes". In: *IEEE transactions on information theory* 52.6 (2006), pp. 2551–2567.

[SMR19] I. Swett, M.-J. Montpetit, and V. Roca. *Coding for QUIC*. Internet-Draft draft-swett-nwcrg-coding-for-quic-02. Work in Progress. Internet Engineering Task Force, Feb. 2019. 15 pp.

[SOCKET] *socket(2) Linux User's Manual*. `https://man7.org/linux/man-pages/man2/socket.2.html`.

[Sun+11] J. K. Sundararajan, D. Shah, M. Médard, S. Jakubczak, M. Mitzenmacher, and J. Barros. "Network coding meets TCP: Theory and implementation". In: *Proceedings of the IEEE* 99.3 (2011), pp. 490–512.

[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.

[Swa+16] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, et al. "Dependent types and multi-monadic effects in F". In: *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2016, pp. 256–270.

[Swe+20a] I. Swett, M.-J. Montpetit, V. Roca, and F. Michel. *Coding for QUIC*. Internet-Draft draft-swett-nwcrg-coding-for-quic-03. Work in Progress. Internet Engineering Task Force, Jan. 2020. 16 pp.

[Swe+20b] I. Swett, M.-J. Montpetit, V. Roca, and F. Michel. *Coding for QUIC*. Internet-Draft draft-swett-nwcrg-coding-for-quic-04. Work in Progress. Internet Engineering Task Force, Mar. 2020. 17 pp.

[Swe17] I. Swett. *QUIC-FEC*. `https://datatracker.ietf.org/meeting/99/materials/slides-99-nwcrg-08-swett-quic-fec-00`. Accessed: 2018-06-02. 2017.

[TE22] D. J. D. Touch and W. Eddy. *TCP Extended Data Offset Option*. Internet-Draft draft-ietf-tcpm-tcp-edo-13. Work in Progress. Internet Engineering Task Force, Oct. 2022. 23 pp.

[Tho+19] L. Thomas, E. Dubois, N. Kuhn, and E. Lochin. "Google QUIC performance over a public SATCOM access". In: *International Journal of Satellite Communications and Networking* 37.6 (2019), pp. 601–611.

[Tic+05] O. Tickoo, V. Subramanian, S. Kalyanaraman, and K. Ramakrishnan. "LT-TCP: End-to-end framework to improve TCP performance over networks with lossy channels". In: *IWQoS*. Springer. 2005, pp. 81–93.

[TKG20] M. Trevisan, A. S. Khatouni, and D. Giordano. "ERRANT: Realistic emulation of radio access networks". In: *Computer Networks* 176 (2020), p. 107289.

[TMW97] K. Thompson, G. J. Miller, and R. Wilder. "Wide-area Internet traffic patterns and characteristics". In: *IEEE network* 11.6 (1997), pp. 10–23.

[Tou+11] P. U. Tournoux, E. Lochin, J. Lacan, A. Bouabdallah, and V. Roca. "On-the-fly erasure coding for real-time video applications". In: *IEEE Transactions on Multimedia* 13.4 (2011), pp. 797–812.

[Tou22] D. J. D. Touch. *Transport Options for UDP*. Internet-Draft draft-ietf-tsvwg-udp-options-19. Work in Progress. Internet Engineering Task Force, Dec. 2022. 43 pp.

[Tre+18] M. Trevisan, D. Giordano, I. Drago, M. Mellia, and M. Munafo. "Five years at the edge: Watching internet from the ISP network". In: *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. 2018, pp. 1–12.

[Twi22a] *@ElonMusk, Twitter, on the launch of ISL-enabled satellites*. 2022. URL: https://twitter.com/elonmusk/status/1436541063406264320 (visited on 05/13/2022).

[Twi22b] *@ElonMusk, Twitter, on ISL activation*. 2022. URL: https://twitter.com/elonmusk/status/1535394359373443073 (visited on 05/13/2022).

[VJA23] V. Vasiliev, N. Jaju, and B. Aboba. *WebTransport*. W3C Working Draft. https://www.w3.org/TR/2023/WD-webtransport-20230405/. W3C, Apr. 2023.

[W3C23a] W3C. "WebCodec API'". In: (2023). https://www.w3.org/TR/webcodecs/.

[W3C23b] W3C. "WebTransport API". In: (2023). https://www.w3.org/TR/webtransport/.

[Wah+94] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. "Efficient software-based fault isolation". In: *ACM SIGOPS Operating Systems Review*. Vol. 27. 5. ACM. 1994, pp. 203–216.

[Wan+04] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. "Image quality assessment: from error visibility to structural similarity". In: *IEEE transactions on image processing* 13.4 (2004), pp. 600–612.

[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.

[Wie+03] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. "Overview of the H. 264/AVC video coding standard". In: *IEEE Transactions on circuits and systems for video technology* 13.7 (2003), pp. 560–576.

[Wir] T. Wirtgen. "xBGP: Faster Innovation in Routing Protocols". In.

[Wir+19] T. Wirtgen, C. Dénos, Q. De Coninck, M. Jadin, and O. Bonaventure. “The case for pluginized routing protocols”. In: *2019 IEEE 27th International Conference on Network Protocols (ICNP)*. IEEE. 2019, pp. 1–12.

[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*. IEEE. 2009, pp. 79–93.

[Zha+05] Q. Zhang et al. “End-to-end QoS for video delivery over wireless Internet”. In: *Proceedings of the IEEE* 93.1 (2005), pp. 123–134.

[Zve+21] M. Zverev, P. Garrido, F. Fernández, J. Bilbao, Ö. Alay, S. Ferlin, A. Brunstrom, and R. Agüero. “Robust QUIC: Integrating practical coding in a low latency transport protocol”. In: *IEEE Access* 9 (2021), pp. 138225–138244.

[ZZZ04] Q. Zhang, W. Zhu, and Y.-Q. Zhang. “Channel-adaptive resource allocation for scalable video transmission over 3G wireless network”. In: *IEEE TCSVT* 14.8 (2004), pp. 1049–1063.