UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

Improving QUIC

Bringing Forward Erasure Correction and unreliable transfer to a reliable protocol.

Dissertation presented by
François MICHEL

for obtaining the Master's degree in
Computer Science

Supervisor(s)
Olivier BONAVENTURE

Reader(s)
Quentin DE CONINCK, Ramin SADRE

Academic year 2017-2018

Contents

Introduction

During the last couple of years, the QUIC protocol [1] has rapidly increased in popularity. This new transport protocol proposes a large set of built-in features such as reliable data transfer, encryption, authentication and a stream abstraction for data multiplexing. All these features make this protocol attractive for web browsing and video streaming, which are common activities in the Internet nowadays. While the main use-case for which QUIC has been designed is HTTP/2 [2], other applications have already been considered [3]. Although real-time communications would also benefit from these features, QUIC only provides a fully reliable data transfer based on retransmissions of lost packets. As real-time communications typically exchange data with an expiration deadline [4], providing a fully reliable data transfer might not be the most relevant strategy, especially if the data expiration deadline is shorter than the retransmission timer of the QUIC sender. An unreliable data transfer where the sender does not waste time retransmitting data whose deadline is expired would thus be relevant in this case. On the other hand, an unreliable data transfer would come with the drawback of potentially loosing an important part of the transmitted data. This is why Forward Erasure Correction is considered in order to reduce the data loss for an unreliable data transfer.

The topic of this thesis is to enlarge the set of use-cases in which QUIC could be considered as a solution, by providing the protocol with unreliable data transfer along with Forward Erasure Correction. This thesis describes the design of the partial reliability and Forward Erasure Correction extensions of QUIC and analyses the performances of an implementation of this design. We finally evaluate the benefits of a multipath transmission on data transfer using Forward Erasure Correction.

Chapter 1

State of the art and theoretical prerequisites

In this section, we describe the context on which this thesis has been developed. The transfer of data over a network can be classified in different categories depending on the requirements of the connection. We focus on reliable and unreliable data transfer and existing transport protocols providing these kinds of transfer. We then describe the QUIC protocol, principal topic of this thesis. Finally, we discuss Forward Error Correction and explain some theoretical background, necessary to correctly understand the work of this thesis.

1.1 Reliable data transfer

A fully reliable data transfer ensures that all the data sent have been correctly received and delivered in order to the application on the receiver-side. As the quality of the connection between two hosts can vary and losses can occur during the transmission, a fully reliable data transfer will retransmit the data that have been lost. This kind of data transfer is provided by different transport protocols. The Transmission Control Protocol (TCP) [5] is the most popular and is used to transmit a large fraction of the internet traffic nowadays [6]. TCP offers a network abstraction such that every application using it can see the received data as a reliable data stream. The reliable transmission of data relies on acknowledgements: the TCP receiver explicitly acknowledges the received data. This implies that if a segment of data is lost, the receiver will have to wait the retransmission timeout (RTO) of the sender to be over before receiving the retransmission of these data. This behaviour can make TCP unsuitable for real-time data transfer over a lossy channel, as the RTO of a TCP sender must be longer than the round-trip-time of the connection. TCP is thus mostly used for use-cases like web browsing and non real-time video streaming. In these use-cases, the wait for an additional RTO to receive a retransmitted TCP segment does not have a high impact on the quality of transmission perceived by the user.

Despite its age, TCP is still analysed and extended nowadays to improve its efficiency. For example, the TCP Fast Open (TFO) [7] extension has recently been proposed. It securely allows a server to send application data before the end of the protocol handshake. Figure 1.1 shows a classical TCP handshake and an HTTP request while Figure 1.2 shows a TCP handshake and a request with the TCP Fast Open extension. The client of Figure 1.1 asks for the content of a web page at the end of the TCP handshake. It thus needs to wait for a full round-trip-time before performing its request. The client of Figure 1.2 can perform its request along with the first message of the TCP handshake, saving the time of the handshake.

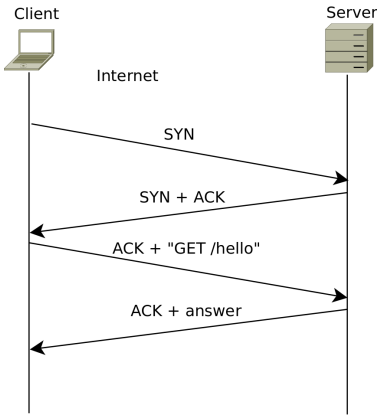


Figure 1.1: HTTP Over TCP

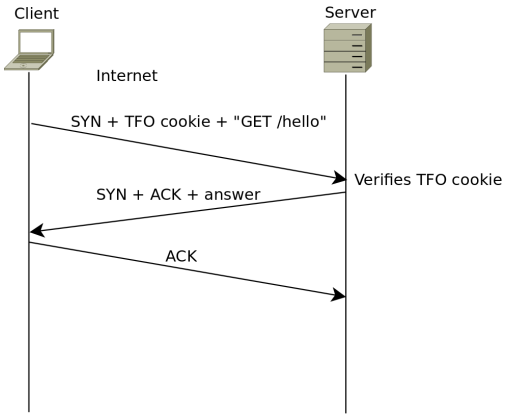


Figure 1.2: HTTP over TCP with TFO

Besides TCP, other transport protocols provide fully reliable transfer, like the Stream Control Transmission Protocol (SCTP) [8] or the new Quick UDP Internet Connections (QUIC) [1], which is the protocol that we study in this thesis.

1.2 Unreliable data transfer

An unreliable data transfer does not provide the guarantee that all the sent data will be delivered. This is useful when the data loose their value after some time. In this case, it could be better to transmit new data than wasting time and bandwidth retransmitting the expired lost data. The most popular transport protocol that does not provide a reliable data transfer is the User Datagram Protocol (UDP) [9]. UDP only sends messages from the application as datagrams, without caring about the ordering of the data delivery or the loss of transmitted data. Due to its simplicity and its small header overhead, other transport protocols are built on top of UDP in order to provide more guarantees about the transmission. This is the case of the Real-time Transport Protocol (RTP) [10] which typically uses UDP to transmit its packets and adds some additional guarantees on the transmission like packets re-ordering. The QUIC protocol is also built on top of UDP and provides additional features such as full reliability. Figure 1.3 shows a UDP packet whose payload is a QUIC packet.

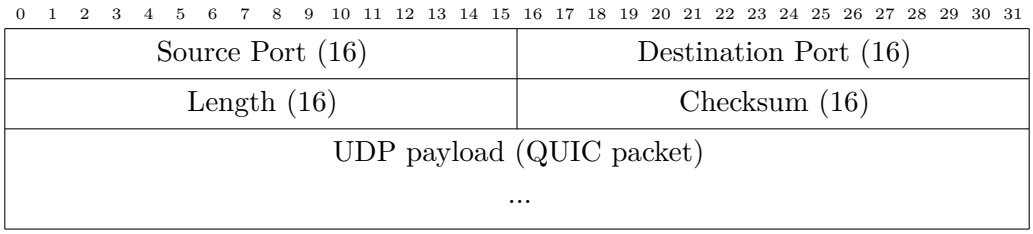


Figure 1.3: UDP packet containing a QUIC packet as its payload

1.3 Multipath transmissions

Multipath transmissions allow hosts to maintain a TCP connection while using multiple network interfaces and IP addresses at the same time, enhancing the throughput and resilience of a connection against network handovers compared to a single-path connection. Figure 1.4 shows a configuration where the client has access to two different network interfaces to contact the server.

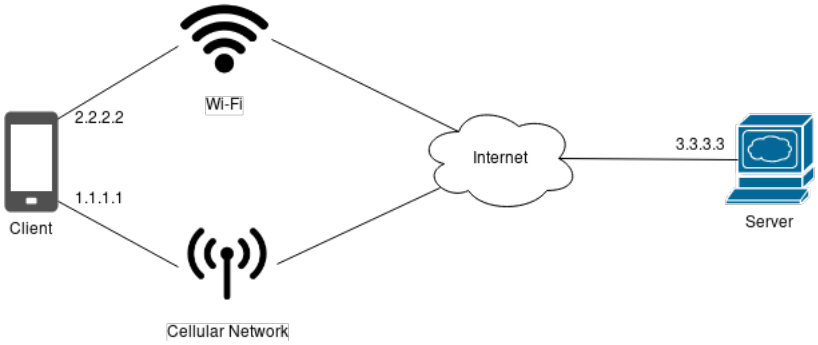


Figure 1.4: Configuration with multiple available interfaces

Using a multipath communication allows the server to answer to the client on any of its IP addresses. An extension of the TCP procotol, MPTCP [11] , has recently been proposed in order to support multipath transmissions.

There exists different strategies to select the path on which to send data. These strategies are called paths *schedulers*. There exists different multipath schedulers, designed for different purposes. For example, the *round-robin* scheduler alternates the paths on which the packets will be sent. It allows to spread the load equally on each path. The *LowRTT* scheduler sends a packet on the path with the lowest round-trip-time (RTT). It allows to avoid packets re-ordering at the receiver-side. Multipath transmissions are still an open field and will be considered in the work of this thesis.

1.4 The QUIC protocol

The *Quick UDP Internet Connections* (QUIC) protocol is a newly introduced transport protocol that aims at providing more functionalities than what TCP already provides. QUIC first provides a stream-based data abstraction that allows stream multiplexing. Multiple data streams can be used during the same connection and processed independently. Compared to TCP, this prevents the existence of head-of-line blocking between multiple streams compared to TCP. Figure 1.5 shows the sequential download of an image and a text in the same TCP connection. As TCP does not provide streams multiplexing, the data lost during the download of the image will prevent the text from being delivered to the application. As streams in QUIC are independently delivered to the application, using one stream for the text and one stream for the image avoids this problem, as shown in Figure 1.6.

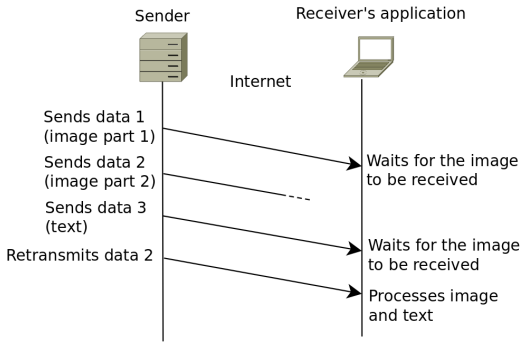


Figure 1.5: Head-of-line blocking

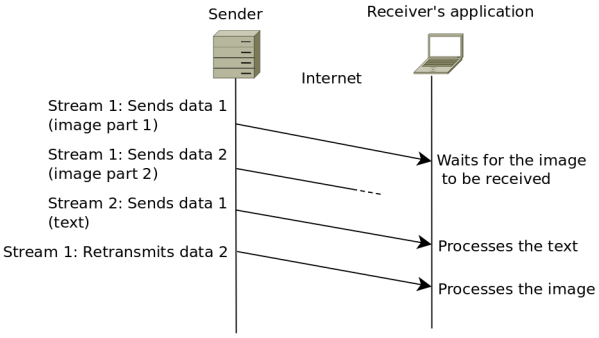


Figure 1.6: No head-of-line blocking thanks to streams multiplexing

All the application data that are sent with QUIC are encrypted and authenticated. In the recent IETF drafts [1], QUIC uses the TLS 1.3 security protocol and dedicates one stream (the

stream 0) to TLS[1]. Using a fully encrypted payload makes the protocol more robust towards middleboxes, well known to slow down the deployment of network protocols extensions [12][13].

A QUIC packet is currently divided into two parts: the header, which is authenticated, and the payload, which is authenticated and encrypted. The QUIC payload can be seen as a container for one or multiple QUIC frames, which are all independently handled by the receiver. There exists many types of frames. Among them, the frames containing the application data are the *stream frames*, while the frames acknowledging the received packets are the *ACK frames*. Using the frame abstraction allows QUIC to be more modular and extensible, by defining new protocol behaviours for handling new frames and letting the remaining of the protocol unchanged.

QUIC uses the concept of version and indicates its current version using a version number in its header. To make this work, the QUIC designers define a set of invariants that won't change between the different versions and that will make QUIC able to perform version negotiation between the client and the server [14].

QUIC implementations are currently defined as user-space libraries instead of being part of the OS kernel. Having a user-space implementation allows the protocol to evolve rapidly, regardless of the evolution of an underlying operating system.

There are currently two concurrent designs of QUIC. The first one is the original design of QUIC, currently called *Google-QUIC* (or *gQUIC*) [15] and the second one is *IETF-QUIC*, coming from the QUIC standardisation effort of the IETF [1]. While the implementations of Google-QUIC are supposed to transition towards IETF-QUIC, the QUIC implementation used for this master thesis has not completed its design shift yet. Furthermore, while IETF-QUIC is still regularly evolving, Google-QUIC is currently more mature and deployed at a large scale on the Google servers [16]. This is why we will focus on Google-QUIC in this thesis.

1.4.1 QUIC handshake

At the connection establishment, QUIC performs a cryptographic handshake on the stream 0, used to determine the keys to encrypt and authenticate the packets sent subsequently. No data exchange can be performed while the packets are not cryptographically protected.

The cryptographic handshake also allows the peers to exchange transport parameters that will be used to regulate the transmission. For example, the peers will advertise the maximum number of streams that they allow to use during the connection. These parameters can be modified afterwards using dedicated QUIC frames. The recent versions of QUIC use TLS 1.3 to perform the cryptographic handshake and thus perform a key exchange with reduced latency. The QUIC transport parameters exchange is done using a dedicated TLS extension. An example of QUIC handshake with TLS 1.3 is depicted in Figure 1.7.

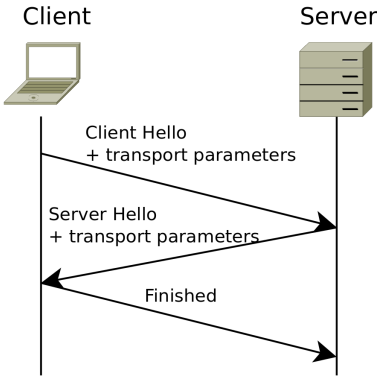


Figure 1.7: Classical QUIC cryptographic handshake using TLS 1.3

[1]As the time of writing, the QUIC working group plans to drop the Stream 0 and TLS in favor of DTLS, which does not need a reliable data transfer to establish a secure connection

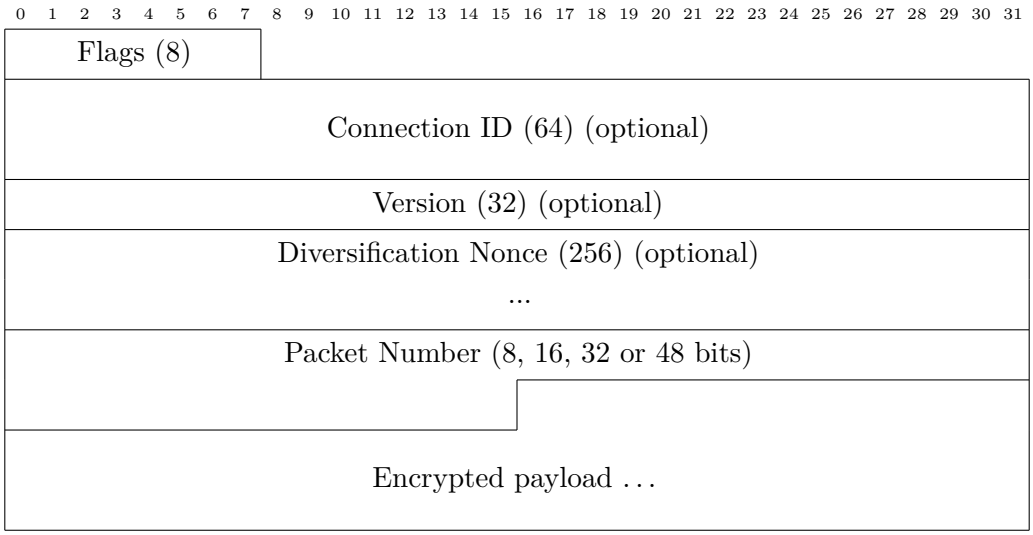


Figure 1.8: Regular QUIC Public Header

1.4.2 Google-QUIC header

Google-QUIC defines its packet header as a *public header*. It is in clear text, and protected by authentication. Figure 1.8 shows the format of the Google-QUIC Regular Packet Public Header [15]. Here are the different fields and their description :

- The *Flags* field holds different informations such as the QUIC packet type, the use of multipath communication, the presence of the *Connection ID* and *Version* fields and the length of the *Packet Number* field.
- The *Connection ID* field defines a unique ID for the connection that allows a QUIC connection to survive a change in the IP address or port number of one of the peers.
- The *Version* field indicates the version of the protocol.
- The *Diversification Nonce* is used for a cryptographic purpose.
- The *Packet Number* field indicates the sequence number of the packet.

The IETF-QUIC design defines two types of QUIC headers and has different header fields [1]. As the QUIC implementation used in this thesis is based on Google-QUIC, we do not describe the IETF-QUIC packet headers in this section.

1.4.3 Google-QUIC stream frame

The data sent through streams by the application transits into *stream frames*. Figure 1.9 shows the format of a Google-QUIC stream frame.

The first byte is the type byte of the stream frame. Every QUIC frame whose first bit is set will be considered as a Stream frame. The remaining bits of the type byte contain informations concerning the structure of the frame. Here are the informations contained in the type byte :

- The *FIN bit* (f) is set to 1 in the last stream frame of a stream to indicate that the sender will not send any additional data to this stream.
- The d bit indicates the presence of the *Data Length* field of the frame. This bit set to 1 indicates that the *Data Length* field is present in the frame header.

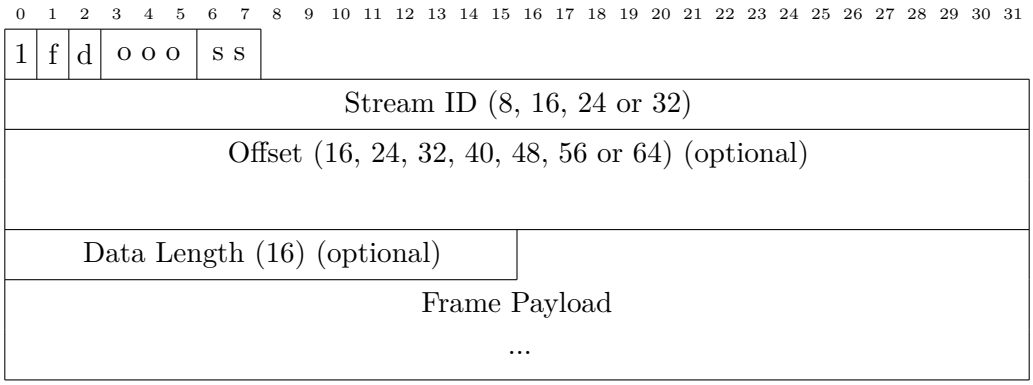


Figure 1.9: Google-QUIC stream frame format

- The *ooo* bits indicate the size of the *Offset* field of the frame.
- The *ss* bits indicate the size of the *Stream ID* field of the frame.

Once the type byte of the stream frame has been parsed, the QUIC receiver can parse the header of the frame. Here are the fields composing the header of a stream frame :

- The *Stream ID* field represents the unique ID of the stream. This allows to send data on multiple streams concurrently.
- The *Offset* field represents the position in the stream of the first byte of data contained in the stream frame. A stream frame containing the first byte of a stream has the offset 0.
- The *Data length* field indicates the length in bytes of the frame payload. This field may be omitted if the stream frame is at the end of a QUIC packet.

1.4.4 QUIC reliable data transfer

QUIC provides a fully reliable data transfer based on retransmissions: the retransmittable content of lost QUIC packets will be retransmitted to the receiver. A packet itself is never retransmitted. Its content will be placed in another packet with a new packet number. A QUIC receiver acknowledges the received packets by sending an *ACK frame* to the sender. The ACK frame contains the ranges of packet numbers of packets that have been successfully received by the receiver. There will be gaps in these ranges for packets that have not been received yet. This mechanism is known as *selective acknowledgements*, allowing the sender to retransmit only the payload of packets that have not been acknowledged.

1.4.5 Google-QUIC encryption and authentication

Google-QUIC sends encrypted and authenticated packets using Authenticated-Encryption with Associated-Data (AEAD) [17] [18] that offers a way to encrypt data and to authenticate it along with an unencrypted part. The encrypted data correspond to the packet payload and the unencrypted data correspond to the QUIC Public Header. A tag of 12 bytes is added at the end of the packet, containing data to ensure authentication and integrity of the payload and Public Header. Figure 1.10 shows the different parts of a QUIC protected packet. The Public Header, frames header and AEAD tag add a certain overhead compared to sending application data directly through UDP.

Public Header ...
Encrypted payload ...
Tag (12 bytes)

Figure 1.10: Google-QUIC full packet format

1.5 Forward Erasure Correction

When dealing with lossy communication channels and short expiration deadlines for the transmitted data, basing the recovery of lost data on retransmissions may not be the most appropriate solution. Waiting for a full retransmission timeout before retransmitting a packet may be problematic with real-time applications, especially in the case where the data expiration deadline is smaller than the RTT of the connection. Furthermore, the RTT is a parameter that can strongly vary depending on the connection, meaning that if waiting for a retransmission is not harmful for the data on-time arrival in some particular network configuration, it may not necessarily be the case in another one. A retransmission-based data transfer is thus not general enough to provide guarantees about lost data recovery.

The idea behind Forward Erasure Correction (FEC) is to send redundancy (Repair Symbols) along with the data that will help to recover the data packets (Source Symbols) that have been lost. Figure 1.11 shows how sending redundancy can help to recover a lost data packet rapidly. In this example, the data 4 will be immediately recovered upon the reception of the redundancy. No retransmission is needed.

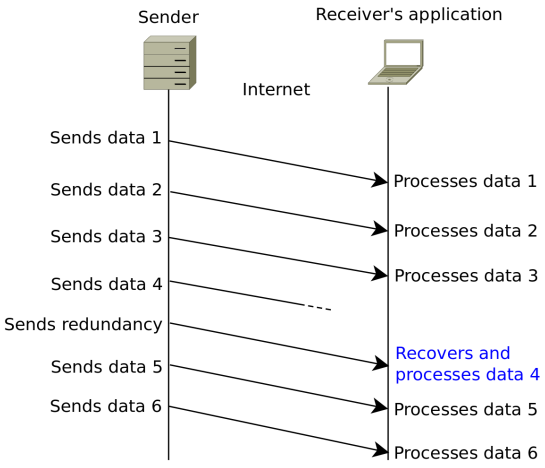


Figure 1.11: The redundancy will help to recover data 4 without waiting for its retransmission.

There are several ways to compute the redundancy to be sent along with the data packet. The simplest idea would be to send a duplicate of each data packet, hoping that at least one copy will arrive correctly and on-time to the receiver. Although straightforward and easy to implement, this solution is far from optimal and there exists a multitude of different techniques to define the redundancy. We use the word *FEC scheme* to refer to one of these techniques.

1.5.1 Error Correction versus Erasure Correction

During a transmission over a lossy communication channel, the exchanged packets can be altered in two ways.

1. The packets can be modified due to an error caused by the physical imperfection of the channel. The receiver should be able to detect these errors and correct it. In this case, we talk about *Forward Error Correction*. Figure 1.12 shows an example of communication using Forward Error Correction.

2. The packets can simply never arrive to the receiver. This can happen due to a failure on the network path between the two hosts or an important congestion on the network path that would cause a router to drop them. The receiver will have to detect which packets have been lost and recover them. When we need to recover lost packets and we know the position of the lost packet in the packets sequence, we talk about *Forward Erasure Correction*. Figure 1.13 shows an example of communication using Forward Erasure Correction.

Some transport protocols also perform an integrity check on the received packets (like UDP that integrates a checksum field in its header) and can decide to not deliver the packet to the application if it appears that the received packet has been modified. This means that errors can be perceived as erasures, as the error detection has already been performed upstream.

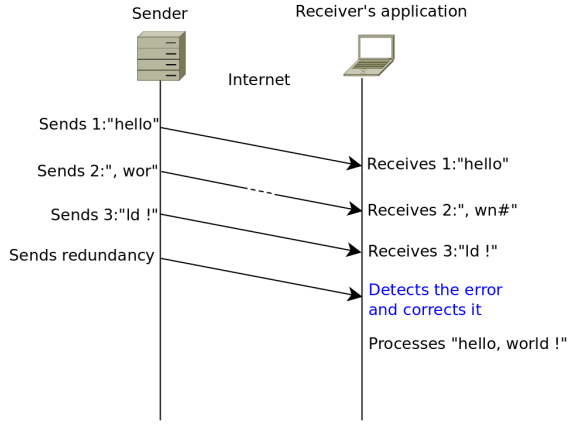


Figure 1.12: Forward Error Correction

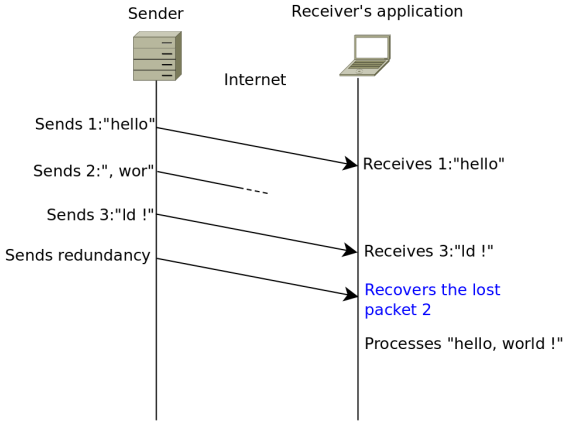


Figure 1.13: Forward Erasure Correction

1.5.2 Erasure correction strategies

In this master thesis, we look more closely to Forward Erasure Correction. Indeed, QUIC relies on UDP which provides an integrity check on the received data and QUIC itself checks the integrity and authenticates the received data. There is thus a small probability that data arrive to the QUIC receiver with errors in the payload. This is why when we use the terms FEC in this document, we refer to *Forward Erasure Correction* and not *Forward Error Correction*.

There are different strategies for erasure and error correction. Some of them have an *optimal* data recovery capability, some others are easier to compute.

1.5.3 FECFRAME

Forward Error/Erasure correction can be useful for many different real-time use-cases. For example, FEC can be considered for video-conferences over the web with WebRTC [19] [20] or communications over the cellular network. These different use-cases come with different software solutions or different network protocols, which may lead to duplicate the implementation of the different FEC schemes for using them with a specific software solution. FECFRAME, defined in

RFC6363 [21], proposes a standard way to implement FEC frameworks and their interactions with software applications. Different FEC schemes are also defined in a standard way that is compliant with FECFRAME, which is for example the case for the Reed-Solomon FEC scheme defined in RFC6865 [22]. FECFRAME and its standard FEC scheme definitions have been an important source of inspiration for the implementation work of this thesis.

1.5.4 Block codes

A (n, k) block code is an error correcting code that takes an M-ary codeword A of k symbols and maps it to an M-ary codeword B of n symbols such that $k < n$. The alphabet $\mathcal{A}_B$ for the symbol B has the same size as the alphabet for the symbol A, i.e. $|\mathcal{A}_B| = M^k$. We define the *code rate* of a code as the proportion of the data sent that are not redundant. The code rate of a (n, k) block code is thus equal to $\frac{k}{n}$.

For example, if we consider ternary codewords, a valid $(3, 2)$ block code could be the following mapping :

input	output
00	222
01	221
02	220
10	212
11	211
12	210
20	202
21	201
22	200

Table 1.1: An arbitrary block code

Both input and output alphabet have the same size ($|\mathcal{A}_A| = |\mathcal{A}_B| = 3^2 = 9$). There is however an advantage in transmitting the coded words: we will be able to detect transmission errors if they appear on their first symbol, because the words 0XX and 1XX do not belong to $\mathcal{A}_B$. When each codeword is equally likely to appear, the maximum likelihood decoder will decode a received symbol $\tilde{s}$ by the symbol $s^* = \underset{s \in \mathcal{A}_B}{\operatorname{argmin}} \quad hamming\_distance(\tilde{s}, s)$.

We say that a (n, k) code that maps a word in $\mathcal{A}_A$ to a word in $\mathcal{A}_B$ is a *Maximum Distance Separable* (MDS) code when the minimum hamming distance between all the words in $\mathcal{A}_B$ is equal to $n - k + 1$, which is the maximum possible value. This guarantees an optimal recovery capability for the code. While all error-correcting codes are not necessarily MDS, there exists MDS codes in practice (the Reed-Solomon [23] code is one of them).

1.5.4.1 Systematic codes

As we saw it in our example in Table 1.1, the coded words do not necessarily have to contain the input words: they can be recovered thanks to the table. However, it can be interesting to have the input words contained in the coded words, as in a real-time communication, it would allow to separate the codeword in two parts, the *Source Symbols* and the *Repair Symbols*, send the Source Symbols as soon as possible regardless of the coding and generate and the Repair Symbols afterwards.

The codes in which the source data reside in the coded data are called *systematic codes*. While it is sometimes harder to generate systematic codes, it can have a high impact on the

delay needed to decode the received coded words. Our design for the FEC extension of QUIC will be focused on systematic codes.

1.5.4.2 Packet-level coding

In a packet-based transport protocol like UDP, we never receive only a part of a packet. Either the packet is received, potentially with errors if we do not only consider erasures, either it is not received at all. We can thus go one level of abstraction higher and consider entire packets as symbols, instead of protecting an arbitrary number of bits in the packets. This is called *packet-level coding*. When using packet-level coding, a systematic (n, k) block code considers a block of n packets in which k packets are the k *Source Symbols* and the $n - k$ remaining packets are the $n - k$ *Repair Symbols*.

1.5.4.3 The 1D XOR code

One of the simplest code we can imagine is an extension of the parity-bit to the packet-level. A XOR code is a systematic $(n, n-1)$ block code: only one Repair Symbol is generated for the block. With an input block of $n-1$ symbols $s_1\, s_2 \ldots s_{n-1}$ and the symbol $\otimes$ representing the bitwise XOR operation, here is how to generate the Repair Symbol r :

$$r := s_1 \otimes s_2 \otimes \ldots \otimes s_{n-1}$$

The coded word $s_1\, s_2 \ldots s_{n-1}\, r$ contains one more symbol than the input word. The code rate is thus $\frac{n-1}{n}$, which means that some code rates cannot be achieved with the 1D XOR code, such as the $\frac{3}{5}$ code rate.

The 1D XOR code can handle the erasure of maximum 1 symbol to be able to recover it. If the erased symbol is the Repair Symbol r, nothing needs to be done as all the Source Symbols have been received. If the erased symbol is the i^{th} Source Symbol, here is how to recover it :

$$s_i = s_1 \otimes \ldots \otimes s_{i-1} \otimes s_{i+1} \otimes \ldots \otimes s_{n-1} \otimes r$$

The 1D XOR code is thus easy to implement but isn't either good to handle loss bursts or to easily achieve an arbitrary code rate.

1.5.4.4 The Reed-Solomon code

The Reed-Solomon code [23] is one of the most popular error correcting codes. It relies on a well known property of the polynomials, stating that k points uniquely identify a polynomial of degree $k-1$. For a Source Symbols sequence $s_1\, s_2 \ldots s_k$, the basic version of the code computes the polynomial

$$P(x) = s_1 + s_2 x + s_3 x^2 + \ldots + s_k x^{k-1}$$

The coded word composed of n symbols is $P(\alpha_1)\, P(\alpha_2) \ldots P(\alpha_n)$, with α_1, α_2, ... , α_k being chosen points on which the polynomial is evaluated. However, this code is not systematic. This implies that the Source Symbols are not present in the coded words. Another way of encoding the Reed-Solomon code provides a systematic code. It defines $P(x)$ such that $P(\alpha_i) = s_i\ \forall i : 1 \leq i \leq k$.

The Reed-Solomon code is an MDS code: the minimum distance between all the possible coded words is exactly $n - k + 1$. Indeed, two distinct polynomials of degree $k-1$ cannot have more than $k-1$ points in common. In an erasure channel, the receiver will then be able to recover the Source Symbols as soon as k symbols are received. This makes the Reed Solomon code a good candidate to correct erasure bursts (with a length until $n - k$).

1.5.4.5 The Random Linear Codes (RLC)

The RLC code is a systematic code that generates Repair Symbols as a linear combination of the Source Symbols with randomly generated coefficients. The coded word is thus composed by all the Source Symbols appended with the generated Repair Symbols. The random coefficients generation must be reproducible and thus be done with a Pseudo Random Number Generator whose seed will be sent to the FEC decoder along with the Repair Symbols. Each received Repair Symbol will allow the FEC decoder to build a linear equation in which the unknown variables are the lost Source Symbols and the constant term is composed by the Repair Symbol and the received Source Symbols

1.5.4.5.1 Example

If we want to encode the $D_1\, D_2\, D_3\, D_4$ symbols sequence with a $(6, 4)$ RLC code, we need to generate two Repair Symbols R_1 and R_2. Let $\{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8\}$ be a sequence of randomly generated coefficients with the seed s, we generate R_1 and R_2 as :

$$c_1 D_1 + c_2 D_2 + c_3 D_3 + c_4 D_4 = R_1 \tag{1.1}$$

$$c_5 D_1 + c_6 D_2 + c_7 D_3 + c_8 D_4 = R_2 \tag{1.2}$$

The coded word $D_1\, D_2\, D_3\, D_4\, R_1\, R_2$ is sent to the FEC decoder as well as the seed s. If two Source Symbols are lost, say D_1 and D_4, the FEC decoder is able to build the following system of equations :

$$c_1 D_1 + c_4 D_4 = R_1 - c_2 D_2 - c_3 D_3 \tag{1.3}$$

$$c_5 D_1 + c_8 D_4 = R_2 - c_6 D_2 - c_7 D_3 \tag{1.4}$$

where the right-hand side is a constant term and D_1 and D_4 are the unknown variables of this system. If the equations 1.3 and 1.4 are linearly independent (which will occur with a very high probability if the coefficients of the equations have correctly been randomly generated), this system of two equations is a full-rank system and will be easily solved using for example a Gaussian elimination algorithm. After solving the system, we will be able to recover D_1 and D_4.

1.5.5 Convolutional codes

A *convolutional* code (a.k.a. *sliding window* code) is an error correcting code based on the incremental generation of the Repair Symbols by applying a sliding window computation on the sequence of Source Symbols to be sent. Instead of considering blocks of Source Symbols, we see the Source Symbols as a stream and do the coding incrementally. This means that one Source Symbol will be used multiple times to compute different codewords. The window has a length L, performs shifts of k Source Symbols at each iteration and outputs a codeword of c symbols for each iteration, implying that the code rate is $\frac{k}{c}$. We describe such a code as a (c, k, L) convolutional code.

Figure 1.14 shows an example of a $(3, 2, 4)$ convolutional code: the window of size 4 slides with steps of 2 on the Source Symbols stream and outputs 3 symbols at each step. For example, the coded word for the source word $D_4\, D_5$ is $C_1\, C_2\, C_3$. In this case, the coded symbol has a size of 3, which means that the code rate is $\frac{2}{3}$, as we send three symbols instead of two. Note that the Source Symbols D_4 and D_5 are used to compute both $C_1\, C_2\, C_3$ and $C_4\, C_5\, C_6$.

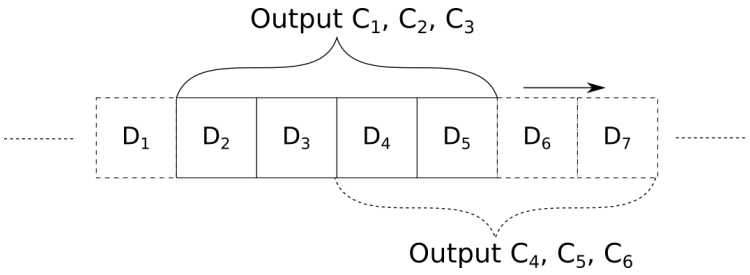


Figure 1.14: Example of convolutional code

As for the block codes, the implementation of convolutional codes can be *systematic*, meaning that the coded words can be decomposed in two parts, the first containing the input Source Symbols, and the second containing redundancy, which can be seen as Repair Symbols.

An important difference between block and convolutional codes is that a block code has to gather a number of new Source Symbols equal to the size of its block to generate an n-size codeword, while a convolutional code needs a number of new Source Symbols equal to its step size k to generate a new codeword.

A (n, k) block code applied on sequential chunks of a sequence of symbols can be seen as a (n, k, k) convolutional code.

1.5.5.1 The Random Linear Codes convolutional code

The incremental computations of a convolutional code allows receiving the Repair Symbols dispersed among the Source Symbols instead of receiving them all at once after all the Source Symbols. The recent work of Roca *et al.* [24] reveals that convolutional codes can reduce drastically the latency needed to recover lost data packets and points the use of Random Linear Codes (RLC) with sliding window as a simple but effective convolutional code. Roca also defines an extension of FECFRAME [25] to use convolutional codes and a draft for implementing convolutional Random Linear Codes [26] with FECFRAME.

Although any block code can potentially be used with a sliding window, the RLC has the nice property to generate systems of *linear* equations. The systems generated by the sliding window at different positions can be put together if they concern the same unknown variables and can thus potentially help to recover loss bursts longer than the number of Repair Symbols generated at each step. Figure 1.15 shows an example of recovering loss bursts with convolutional RLC. The symbols R_1 and R_2 allow the FEC decoder to generate two equations with two unknowns, D_4 and D_5. Except if the two equations are co-linear, this leads to the recovery of these two unknowns.

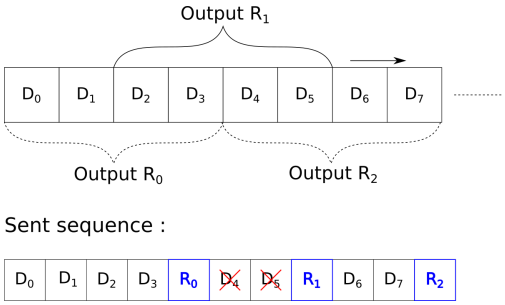


Figure 1.15: The packets D_4 and D_5 will be recovered.

Besides the burst recovery ability, which is also provided by block codes, the use of convolutional codes instead of block codes can also reduce the latency needed to recover loss bursts that are shorter than the maximum burst length supported by the code. Figures 1.16 and 1.17 show

an example of an isolated loss with RLC and Reed-Solomon, both configured with a code rate of 66% and a Source Symbols loss bursts handling capability of 2 symbols. In the case of RLC in Figure 1.16, the symbol D_4 will have to wait for the reception of D_5 and R_1 before being recovered. In the case of Reed-Solomon in Figure 1.17, the symbol D_4 will have to wait for the reception of D_5, D_6, D_7 and R_2. Waiting for that much Source Symbols to be received can be problematic in a real-time use-case as packets are often spaced by several milliseconds.

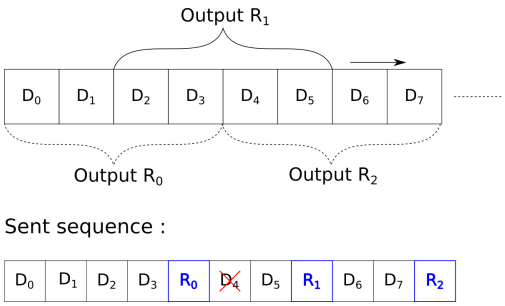


Figure 1.16: One isolated lost packet with convolutional RLC.

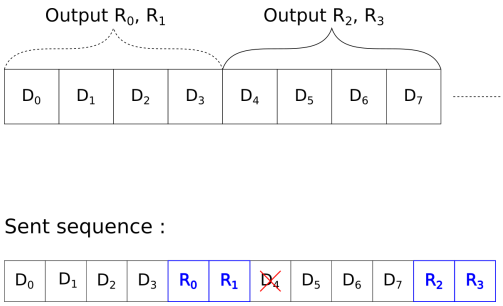


Figure 1.17: One isolated lost packet with Reed Solomon $(6, 4)$.

In order to avoid such a problem, the Reed-Solomon block code of the example should be used with the $(3, 2)$ parameters, but would loose its bursts recovery capabilities. As stated by Roca *et al.* [27], when the burst length is unknown, the application will often fix the burst handling capabilities of their block codes to the maximal burst length that can be recovered with respect to a maximal allowed latency. Using convolutional RLC offers thus a nice way to reduce the latency when the loss bursts are of variable length.

1.5.6 The limits of FEC

Although Forward Erasure Correction allows to avoid waiting for a retransmission of the lost data, it induces some delay (reduced in the case of convolutional codes) before correcting the erasures.

Some real-time delay-sensitive applications cannot afford such a delay and would prefer to skip the data directly instead of waiting for the data erasure correction. This can be the case of online video games, in which each sent message must be received and processed as soon as possible to avoid having a disadvantage compared to the other users. Some video games will thus use simple message-based network protocols without FEC. This is what Valve does with their recently open-sourced *GameNetworkingSockets*[2]. Low bitrate applications also see the benefit of FEC mitigated. If one packet is sent every minute, using a (n, k) block code would make the receiver to wait at least for k additional minutes to recover the first packet of the block if it has been lost.

[2] https://github.com/ValveSoftware/GameNetworkingSockets

Chapter 2

Forward Erasure Correction with QUIC

Forward Erasure Correction was originally part of the QUIC protocol [28], with the hope to mitigate the re-buffering time during video streaming. It has been rapidly dropped from the protocol after the results of experiments showed that the impact of Forward Erasure Correction in QUIC was mostly negative [16]. However, the considered FEC Scheme only allowed to recover single packet losses, leading to poor recovery capabilities with losses that were correlated most of the time [29].

In this work, the interest of Forward Erasure Correction has been re-considered as it could make sense in the case of partially reliable transmission with real-time communications. We also consider the recovery of correlated losses. This chapter presents the design of the Forward Erasure Correction extension of QUIC.

2.1 Representation of a Source Symbol

In order to use Forward Erasure Correction, one needs to define the Source Symbols on which the Forward Erasure Correction will be applied. QUIC offers frames and packets as data containers. QUIC packets and stream frames are thus two candidates to be used as Source Symbols.

2.1.1 Frames as Source Symbols

The first idea would consist in directly considering a raw, unencrypted stream frame as a FEC Source Symbol. This would allow to easily define a level of redundancy on a per-stream basis: each stream could see its frames protected with a different level of redundancy. FEC Schemes generally require all the Source Symbols to have the same size. This implies that padding may need to be added to support shorter stream frames. The *Data Length* field of the stream frames used as Source Symbols should thus always be present in order to recover the original frame from the decoded symbol. This approach has however some drawbacks. A QUIC packet can contain one or more stream frames. In the case of erasure correction, when a packet is dropped, all its contained frames are dropped. If multiple protected stream frames are packed together in a QUIC packet, the loss of a single QUIC packet will cause the loss of several Source Symbols.

2.1.2 Raw QUIC packets as Source Symbols

Another idea consists in using unencrypted QUIC packets as Source Symbols. While it is harder to perform FEC on a per-stream basis, the loss of a QUIC packet is now equivalent to

the loss of a Source Symbol, making the level of redundancy easier to interpret. Conversely to the stream frames solution, there is no need to encode the packet length to recover the packet correctly. Indeed, a padding with zeroes is naturally understood by the QUIC protocol thanks to the padding frame which has no content and has the frame type `0x0`. The only difference between a packet and its recovered version is that the latter will potentially have several padding frames at its end. This however implies to add the *Data Length* field on the stream frames at the end of the packet. Otherwise, the padding will be included into the stream frame.

Using raw QUIC packets or Stream Frames as Source Symbols however implies that the Repair Symbols must be sent encrypted and authenticated. Indeed, if the Repair Symbols are sent without being authenticated, an attacker will be able to modify the decoded symbols by modifying the Repair Symbols. Even if choosing the content of the altered decoded symbols would be hard for an attacker, performing a Denial of Service by modifying Repair Symbols at random would be easy to do, as QUIC connections are closed when a peer attempts to write on unauthorized streams or sends invalid frames. Sending the Repair Symbols unencrypted could allow an attacker to determine the content of the Source Symbols. For example, an attacker could attack a connection using a (n, k) block code if they correctly guess $k - (n - k)$ of the Source Symbols. They will then have to eavesdrop the $n - k$ Repair Symbols and then will be able to recover all the remaining Source Symbols. While this attack can work with any block FEC Scheme, there could also exist simpler attacks that are specific to a FEC Scheme. For example, using the FECFRAME RLC FEC Scheme [26] with some specific parameter values can imply a non-zero probability to have a Repair Symbol equal to a Source Symbol. An attacker could thus have access to some Source Symbols only by eavesdropping some Repair Symbols.

2.1.3 Encrypted packets as Source Symbols

Our last idea consists in using encrypted and authenticated QUIC packets as Source Symbols. In that case, the Repair Symbols could be sent unencrypted. Using encrypted QUIC packets has however some drawbacks. First, the padding with zeroes is not naturally handled anymore by the protocol. Adding padding to an encrypted and authenticated QUIC packet would cause any recovered packet with padding to be rejected by the authentication verification and would potentially corrupt the decrypted payload, depending on the encryption function used. To avoid this problem, the length of the packet should be added to it.

2.1.4 Chosen approach

The approach chosen by this design is to consider raw QUIC packets as FEC Source Symbols to make the implementation and experimentation simpler as a lost source packet exactly corresponds to a lost Source Symbol. It also avoids the problems implied by the use of encrypted QUIC packets as Source Symbols.

2.2 Transmission of the FEC Repair Symbols

The FEC Payload must be distinguished from a classical application data payload in the sense that it should not be transferred to the application upon reception. The FEC Payload is indeed generated by and for the FEC Scheme used by the transport protocol. There are three obvious ways to send the FEC Payload to the receiver.

2.2.1 Sending the FEC Payload as a QUIC packet

The idea would be to define a new type of QUIC packet whose purpose would only be to carry the FEC payload. While this solution would provide a low overhead (only the header of the packet and some metadata to divide the payload into FEC Repair Symbols), it would need to define a new packet type as well as its inner structure to divide the payload into the different FEC Groups and Repair Symbols. Adding this solution to existing QUIC implementations could be a painful process: while implementations are built to handle several different frame types, this is not necessarily the case for packet types and it could imply an important amount of work to handle a new packet type.

2.2.2 Sending the FEC Payload in a dedicated stream

In this case, we would use the stream abstraction already provided by QUIC to transmit the data. While this solution provides some advantages as we would not need to define a new type of packet or frame, this approach also has important drawbacks. First, the smallest meaningful unit of the FEC Payload is the FEC Repair symbol, which often has the size of a QUIC packet. This means that sending a FEC Payload as a stream of bytes does not bring any advantage. It would even bring head-of-line blocking to the transmission of the FEC Payload and block the processing of already received Repair Symbols while they could be processed in any order.

Furthermore, sending the FEC payload via a QUIC stream would either force us to create a new stream type or to reserve a stream ID for the transmission of the FEC payload, which could break the compatibility of some applications already using the QUIC protocol and relying on some precise stream ID's. Furthermore, adding a new corner-case on the stream ID's would bring more complexity to a QUIC implementation.

2.2.3 Sending the FEC Payload in a dedicated frame type

This approach consists in using the frame abstraction provided by QUIC and adding a new type of frame. While this solution needs the definition of a new frame and its header format in addition to the reservation of a new frame type, it provides a good trade-off between the two previous solutions. It allows to send the FEC Payload among other QUIC frames and can be designed in a way to fit the shape of the FEC payload. This is the approach that has been chosen and that will be described in details in the section 2.4.2.

2.3 Modifications to the QUIC protocol

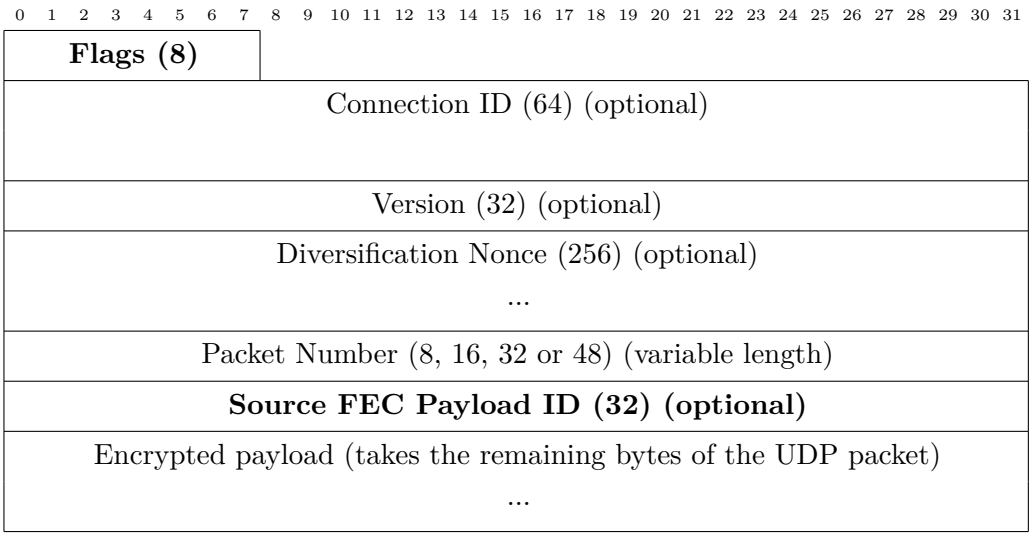
This section defines our modifications to the QUIC packets and frames to support the Forward Erasure Correction extension. Although Forward Erasure Correction allows to recover lost packets without waiting for retransmissions, it consumes more bandwidth than a normal, non-FEC-protected transmission. In order to avoid spending additional bandwidth when it is not needed, our design must allow to define which QUIC packets should be considered as FEC Source Symbols. The design described in this thesis considers two kinds of QUIC packets :

- *Regular QUIC packets*: these packets represent the packets in the already existing implementations of Google-QUIC.
- *FEC-Protected packets*: these packets are similar to the original QUIC packets excepted that they are protected by Forward Erasure Correction and are considered as Source Symbols. If FEC-Protected packets are lost, a QUIC receiver will be able to recover them thanks to the FEC Repair Symbols.

While the payload of FEC-protected packets can potentially be identical to the payload of the regular QUIC packets, the public header of Google-QUIC has been slightly modified to welcome this distinction.

2.4 Google-QUIC Public Header modifications

In order to bring Forward Erasure Correction to QUIC, modifications have been applied to the Google QUIC Public Header format to represent FEC-protected packets. Hereunder is the new format of a QUIC Packet.



This design uses a previously unused flag in the *Flags* field, flag `0x80`, and adds the *Source FEC Payload ID* field.

We add the *FEC Flag*. A QUIC packet with this flag set is a FEC-protected QUIC packet. A packet with this flag unset is a regular QUIC packet. While the *Source FEC Payload ID* field is present in the Public Header of a FEC-protected QUIC packet, the Public Header of a regular QUIC packet is identical to the QUIC Public Header without the FEC extension. This implies that the design of the FEC extension does not have side effects on the current Google-QUIC design: an implementation of Google-QUIC with the FEC extension will still be compatible with existing implementations of QUIC as soon as FEC is not used.

2.4.1 The Source FEC Payload ID field

The *Source FEC Payload ID* field is only present if the *FEC Flag* is set. It is an opaque field for the protocol that must be filled by the sender-side FEC Scheme and interpreted by the receiver-side FEC Scheme. Its purpose is to identify the packet as a FEC Source Symbol. No restriction is applied in the usage of this field except that it cannot reveal any information concerning the content of a Repair or Source Symbol, as it is not encrypted. As it is part of the Public Header, this value is authenticated and can thus be trusted by the receiver-side FEC Scheme.

2.4.2 The FEC Frame

The FEC Frame has been designed to contain one Repair Symbol. A new QUIC frame type byte has been reserved: the FEC Frame has the frame type `0x0a`. Hereunder is the format of a FEC Frame for Google-QUIC.

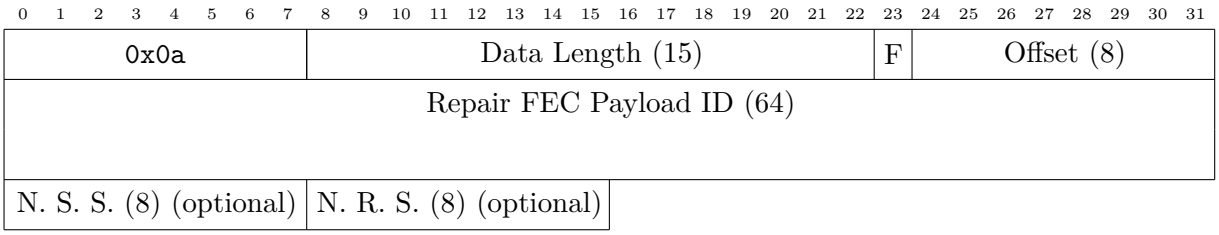


Figure 2.1: Google-QUIC FEC Frame format

2.4.2.1 FEC Frame fields

The *Data Length* field and the *FIN bit* (*F*) are encoded as a 16 bits word. We do not expect the length of a FEC Frame to exceed $2^{15} - 1$ bits as this is currently quite an uncommon size for a QUIC packet. If a Repair Symbol is longer than $2^{15} - 1$, it will have to be split in multiple FEC Frames.

The *Offset* field is a sequence number for the FEC Frames, increasing by 1 for each FEC Frame. It is present to handle the fragmentation of a Repair Symbol into multiple FEC Frames. Indeed, some Repair Symbols might be too large to fit into a single QUIC Packet. The FEC Frame could also be present in the packet along with other frames, preventing the FEC Frame to take the full QUIC packet space. In these cases, a Repair Symbol will be fragmented in pieces and will be reconstructed on the receiver side. We do not expect a Repair Symbol to be fragmented in more than 256 pieces and it must be avoided by an implementation of the FEC Extension. It must be noted that spreading a Repair Symbol into several FEC Frames that are sent in different QUIC packets lowers the probability of the Repair Symbol to be finally received. Indeed, a Repair Symbol can be reconstituted and delivered to the FEC Scheme only if all the FEC Frames concerning it have been successfully received. A Repair Symbol will not be reconstituted as soon as a packet containing one of its FEC Frames is lost. The probability of loosing a Repair Symbol will thus increase with the number of pieces in which it has been fragmented.

The *Repair FEC Payload ID* field is an opaque field for the protocol. It is used by the FEC Scheme to identify the Source Symbols concerned by the Repair Symbol contained in this FEC Frame. It is encoded on 64 bits instead of 32 bits because it can also contain informations related to the encoding and decoding procedures of the underlying FEC Scheme. As a FEC Frame is sent encrypted into the QUIC payload, there is no restriction about the usage of this field: it will be handled by the underlying FEC Scheme itself and can contain any needed information. This field is only required if the *Offset* field is equal to 0. It will be omitted otherwise.

The *Number of Source Symbols (N. S. S.)* field tells the receiver the total number of Source Symbols that have been used to generate this Repair Symbol.

The *Number of Repair Symbols (N. R. S.)* field tells the receiver the total number of Repair Symbols that have been generated along with this Repair Symbol. The number includes this

symbol itself.

The *N. S. S.* and *N. R. S.* fields must be present only if the *Offset* field is set to 0 and cannot be present otherwise. Once the FEC Frames have been received by the receiver, the different Repair Symbols are reconstituted from the FEC Frames and the FEC Scheme will be able to reconstruct the lost Source Symbols.

2.5 The FEC Framework

Even if there exists a wide variety of different FEC Schemes, it is possible to extract a common behaviour from FEC Schemes using block codes and another common behaviour for FEC Schemes using convolutional codes. We define here a FEC Framework that implements the common behaviour of these FEC Schemes in order to simplify the implementation of the FEC Schemes. It gives a structure for the Source and Repair FEC Payload ID which are opaque to the protocol. This FEC Framework is inspired from FECFRAME, defined in RFC6363 [21].

2.5.1 The Block FEC Framework

The Block FEC Framework proposes a standard way to exchange informations between block FEC Schemes and the transport protocol. It allows to abstract away the implementation of the different block erasure correcting codes and provides them with the informations required to perform encoding and decoding.

2.5.1.1 Sender-side Block FEC Framework

The sender-side Block FEC Framework receives Source Symbols to protect from the protocol before sending them. It adds a Source FEC Payload ID to these Source Symbols and forms a Source Block that is passed to the FEC Scheme once it is complete. The FEC Scheme generates the Repair Symbols attached to the Source Block. It will also generate a *FEC Scheme-specific value* for each Repair Symbol, containing informations regarding the symbol generation procedure. The FEC Framework then adds the Repair FEC Payload ID to each of these Repair Symbols.

The Source FEC Payload ID has the following format :

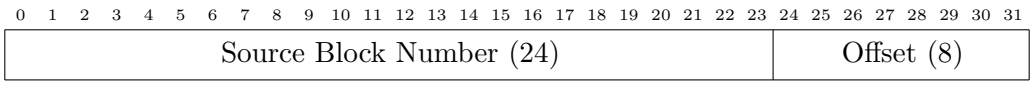


Figure 2.2: Source FEC Payload ID for Block FEC Schemes

The *Source Block Number* field identifies the Source Block to which the Source Symbol belongs. It is encoded on 24 bits. The eight remaining bits form the *FEC Block Offset*. It represents the position of the Source Symbol in the Source Block, often needed by block FEC Schemes to perform the decoding. Encoding a Source Block Number in the Source FEC Payload ID can be useful when using interleaving in the sequence of packets as it allows to have successive packets belonging to different Source Blocks.

The Repair FEC Payload ID has the following format :

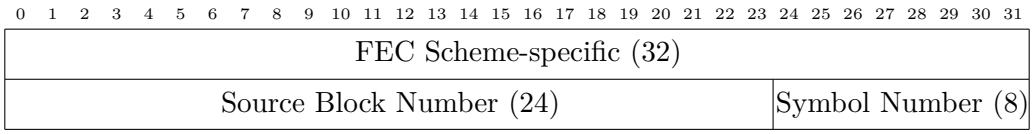


Figure 2.3: Repair FEC Payload ID for Block FEC Schemes

The *Source Block Number* field allows the FEC Framework to identify the Source Block to which the Repair Symbol belongs. The *Symbol Number* indicates the position of the Repair Symbol in the Source Block. The *FEC Scheme-specific* field allows the FEC Scheme to generate and communicate values to the receiver-side FEC Scheme. Each FEC Scheme can potentially encode any value in the *FEC Scheme-specific* field.

2.5.1.2 Receiver-side Block FEC Framework

The receiver-side Block FEC Framework is provided with all the non-erased Source and Repair Symbols. It extracts the *Source Block Number* and *Offset* from the Source FEC Payload IDs and the *FEC Block Number*, *Symbol Number* and *FEC Scheme-specific* fields from the Repair FEC Payload IDs. It then groups together the Source and Repair Symbols having the same *Source Block Number*. Once one or more Source Symbols can be recovered in a particular Source Block, the FEC Framework will provide the FEC Scheme with the Source and Repair Symbols of the Source Block to recover the lost symbols. The recovered Source Symbols will finally be passed to the protocol.

2.5.2 The Convolutional FEC Framework

As for the Block FEC Framework, the Convolutional FEC Framework proposes a standard way to exchange informations between convolutional FEC Schemes and the transport protocol.

2.5.2.1 Sender-side Convolutional FEC Framework

The sender-side Convolutional FEC Framework receives the Source Symbols to protect one-by-one. It adds a Source FEC Payload ID to these packets and places them on the encoding window for a potential Repair Symbols generation. When Repair Symbols generation is needed, the FEC Framework provides the FEC Scheme with the Source Symbols in the current encoding window. The FEC Scheme generates the Repair Symbols as well as the FEC Scheme-specific values if needed. Once the Repair Symbols are generated, the FEC Framework adds the Repair FEC Payload ID to them before sending.

Hereunder is the format of the Source FEC Payload ID for the Convolutional FEC Framework:

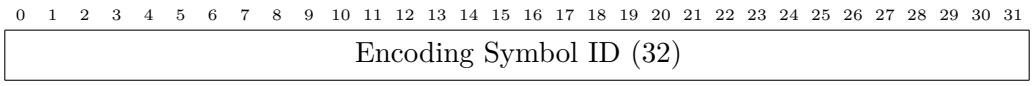


Figure 2.4: Source FEC Payload ID for Convolutional FEC Schemes

The *Encoding Symbol ID* directly identifies the attached Source Symbol. It starts at 0 and increases by 1 at each new Source Symbol. It falls back to 0 after the maximal 32-bits value is reached. This is inspired from the specification of the RLC FEC Scheme for FECFRAME [26] .

The format of the Source FEC Payload ID is described below :

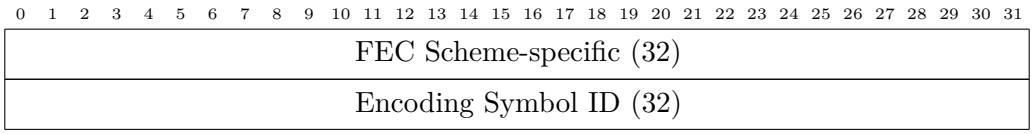


Figure 2.5: Repair FEC Payload ID for Convolutional FEC Schemes

The *Encoding Symbol ID* field represents the Encoding Symbol ID of the last Source Packet in the encoding window that generated the concerned symbol. The *FEC Scheme-specific* field, like for the Block FEC Framework, allows the FEC Scheme to encode any needed specific values related to the encoding/decoding procedure.

2.5.2.2 Receiver-side Convolutional FEC Framework

The receiver-side Convolutional FEC Framework replaces correctly the received Source Symbols in sequence using the *Encoding Symbol ID* of the Source Symbols. For each received Repair Symbol, the Source Symbols used to generate it will be identified by using the Encoding Symbol ID from the Repair FEC Payload ID. When lost Source Symbols can be recovered, the FEC Framework provides the FEC Scheme with the sequence of received Source Symbols and the received Repair Symbols with their associated FEC Scheme-specific values. Once recovered, the Source Symbols are passed to the protocol.

2.5.3 Interactions between the protocol and the FEC Framework

Figure 2.6 illustrates the interactions between QUIC and the FEC Framework. The protocol first sends the QUIC packets that must be protected to the FEC Framework (1). The FEC Framework then builds the FEC Payload ID, transforms this packet into a Source Symbol (i.e. a QUIC FEC-protected packet) and returns it to the protocol in order to send it as soon as possible (2). When necessary, the FEC Framework sends its Source Symbols to the FEC Scheme to generate Repair Symbols (3). For each Repair Symbol, the FEC Scheme also generates FEC Scheme-specific values if needed and gives it to the FEC Framework along with the Repair Symbols (4). The FEC Framework then generates the Repair FEC Payload ID's, attach it to the Repair Symbols and gives the Repair Symbols to the QUIC protocol for them to be sent (5).

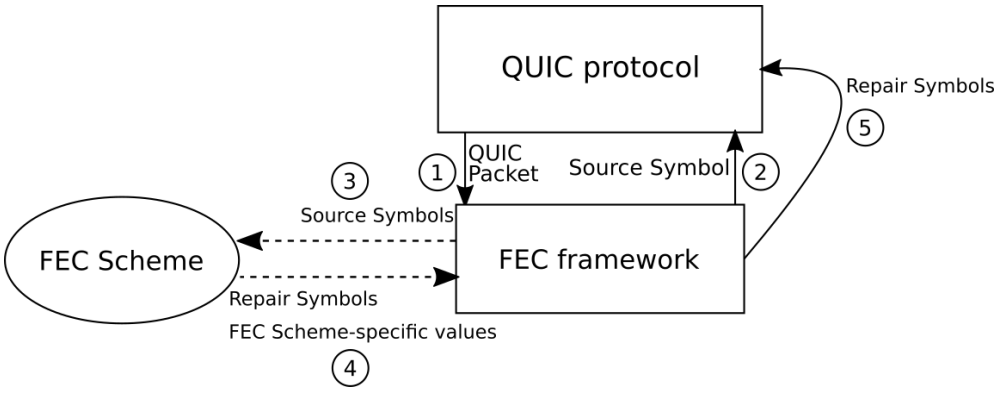


Figure 2.6: The FEC Framework interfaced with the protocol

2.6 FEC Scheme negotiation

As the FEC design allows to use different FEC Schemes, there needs to be a negotiation of the FEC Schemes that are used to encode and decode the Repair Symbols. The choice of

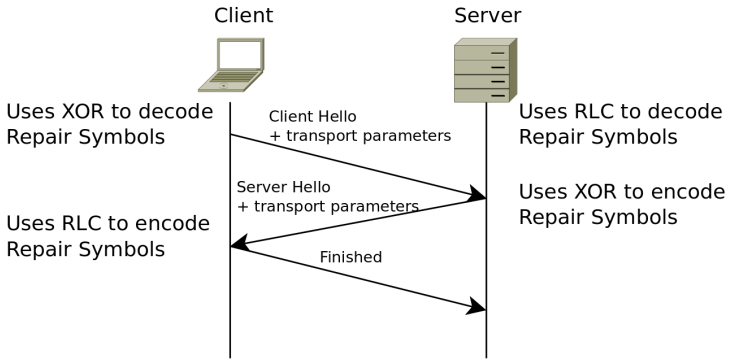


Figure 2.7: FEC Scheme negotiation during the cryptographic handshake

the FEC Scheme is unidirectional: the FEC Scheme used for the Repair Symbols sent to the client can be different from the FEC Scheme used for the Repair Symbols sent to the server. This makes sense in a configuration where the client has low computational capabilities but not necessarily the server. In that kind of configuration, it would be a good idea to choose a *server* $\rightarrow$ *client* FEC Scheme whose decoding procedure has a low computational complexity on the client and a potentially different *client* $\rightarrow$ *server* FEC Scheme whose encoding procedure has a low complexity on the client.

2.6.1 Negotiation procedure

Each peer announces to the other the FEC Scheme that it will use to decode the received Repair Symbols. This allows a peer with low computational capabilities to ensure that it will be able to recover the lost Source Symbols in a reasonable amount of time. The FEC Scheme is announced by each peer during the QUIC cryptographic handshake using the transport parameters. Figure 2.7 shows an example of the proposed FEC Scheme negotiation, with a client using a XOR FEC Scheme to decode the Repair Symbols sent by the server and a server that uses a Convolutional RLC FEC Scheme to decode the Repair Symbols sent by the client. Upon the reception of the transport parameters of its peer, the host sets up its sender-side FEC Scheme to encode the Repair Symbols. In this example, the Repair Symbols sent to the client are thus encoded with a XOR FEC Scheme while the Repair Symbols sent to the server are encoded with a Convolutional RLC FEC Scheme.

Chapter 3

Unreliable data transfer

The current design of Google-QUIC and IETF-QUIC offers a fully reliable data transfer. While it is relevant in most use-cases on the Internet, this mode of transmission assumes that the transferred data do not have an expiration deadline or have a very large one. This is typically the case for web browsing. This assumption is less solid for video-on-demand applications although the deadline problem is mitigated in practice as the data are generally transmitted with a higher transmission rate than the consumption rate. This mitigation is however not possible in the case of live streaming applications where the data transmission rate cannot be higher than the data generation rate. An unreliable data transfer can be useful in these use-cases. When the data expiration deadline is over, the retransmission of the data does not make sense anymore and will imply a waste of time and bandwidth. Figure 3.1 shows a reliable data transfer encountering a packet loss while figure 3.2 shows an unreliable data transfer encountering the same packet loss. As we can see, the receiver of the unreliable data transfer can deliver the data 5 sooner to the application but at the cost of a data loss.

In this chapter, we take a look at the design proposed to implement unreliable data transfers in QUIC. Combined with the Forward Erasure Correction extension, we expect to reach the data expiration deadline more often by avoiding retransmissions of lost data and recovering as much data as possible.

3.1 Unreliable Streams

In QUIC, the data are transmitted through the stream abstraction. The content of a stream transits in a stream frame. Each stream ensures that the data are delivered in order and without

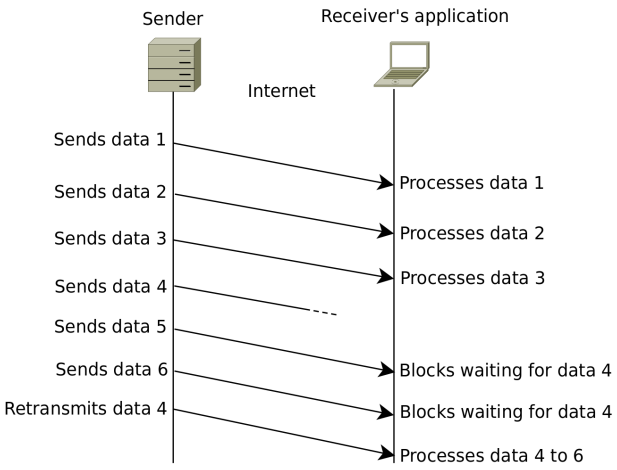


Figure 3.1: Reliable data transfer

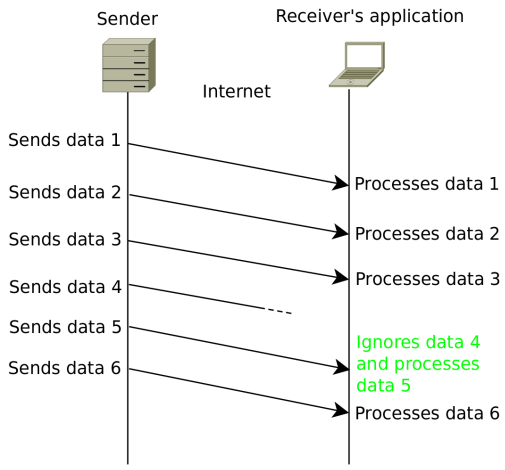


Figure 3.2: Unreliable data transfer

gap. For this abstraction to work correctly, the stream frames must be retransmitted if the packets containing them have been lost. If the data have an expiration deadline, it would be interesting to avoid retransmitting the stream frame when this deadline is expired. However, this would make the stream unreliable and the application should be prepared to receive possibly lossy data. Loosing data whose expiration deadline is passed should however not be a problem by definition.

This leads us to a new stream abstraction: the *Unreliable Stream*. The Unreliable Stream ensures that the data are provided in order but not that there isn't any gap in the data when the stream is processed.

3.1.1 Unreliable Stream frame

A stream frame should contain the information of whether a QUIC stream is unreliable of not. However, it is impossible to add an extra bit to the Google-QUIC stream frame without adding an extra byte to the frame header. The chosen solution is to give the Unreliable Stream frame another type byte to indicate that the stream is unreliable.

A new frame type is thus is used to transport unreliable stream frames. In section 3.1.2 are discussed the different possibilities to add an *unreliable* flag to the original QUIC stream frame.

The type byte of an Unreliable Stream frame is `0x0b`. The second byte of the Unreliable Stream frame is identical to the type byte of an original stream frame and contains the same informations (FIN bit, Data Length field presence, Offset length and Stream ID length for Google-QUIC). The header of an Unreliable Stream frame is thus exactly identical to the header of an original stream frame, with the byte `0x0b` in front of it.

Receiving an Unreliable Stream frame with a Stream ID corresponding to a reliable stream will make the stream become unreliable. The Stream ID's of the Unreliable Streams are thus not isolated from the Stream ID's of the reliable streams. Receiving a reliable stream frame on an Unreliable Stream does not make it reliable.

3.1.2 Avoiding the use of a new frame type

The solution to use a new type byte for the Unreliable Stream frames has been chosen to remain compatible with existing implementations of Google-QUIC when Unreliable Streams are not used. However, several strategies are possible in order to avoid using a whole new frame type for an Unreliable Stream frame. These solutions are described below.

3.1.2.1 Replacing the FIN bit by an Unreliable Bit

The first solution is also proposed by Tiesel *et al.* [30]. It consists in replacing the FIN Bit of a stream frame by an Unreliable Bit indicating if the stream is reliable or not. The closing of a stream could be indicated by another mean, like transmitting a stream frame with a data length of zero.

3.1.2.2 Making a stream becoming unreliable with a frame

The second idea would be to create every stream as a reliable stream and make it become unreliable by sending a specific frame, like a stream frame with a data length of zero and the FIN Bit unset. This solution has the drawback to hide to the receiver application the fact that a stream is opened with the purpose to make it unreliable.

3.1.2.3 Removing a bit from the Stream ID and using it as an Unreliable Flag

This solution is also proposed by Tiesel *et al.* [31]. It consists in removing a bit from the stream ID and reserving it as an Unreliable Bit.

3.1.2.4 Specific to IETF-QUIC: enlarge the stream frame type byte range

In the draft 11 of the IETF-QUIC design [1], the stream frame type byte takes the form `0b00010XXX` (i.e. the range from `0x10` to `0x17`). As no frame type is reserved above `0x17`, adding an unreliable flag to the type byte would be the easiest solution. The new stream frame type byte would now take the form `0b0001XXXX` (i.e. the range from `0x10` to `0x1f`). The new reserved bit would be used as an Unreliable Bit. This currently seems to be the best solution. However, this change is only applicable to an IETF-QUIC implementation.

3.1.3 Unreliable Stream frame retransmission

As said before, the data transmitted over an Unreliable Stream are typically subject to a short expiration deadline. However, the value of this deadline can change depending on the use-case and could be sometimes greater than the retransmission timeout. In such a case, retransmitting the lost frames could make sense if a lost stream frame has not been recovered by the Forward Erasure Correction.

This means that an Unreliable Stream frame is allowed to be retransmitted by the sender while its expiration deadline has not passed. As the expiration deadline changes over the cases, it must be defined by the application. The expiration deadline must be a per-stream value, as the data transmitted over different streams could be subject to different expiration deadlines. The expiration deadlines of the Unreliable Streams are sender-defined values and do not necessarily need to be transmitted to the receiver.

In order to provide a graceful Unreliable Stream closing, a stream frame containing the FIN bit cannot be unreliable. Unreliable Streams will thus be closed with regular stream frames.

3.1.4 Unreliable Stream frame and Forward Erasure Correction

An Unreliable Stream could benefit from Forward Erasure Correction as the Unreliable Stream frames are likely to have an expiration deadline shorter than the RTT. This is why an Unreliable Stream frame should be transmitted into a FEC-protected packet when possible.

Chapter 4

Implementation

Both the FEC and Unreliable Streams extension are implemented in `quic-go`. Besides the advantages of the language itself, `quic-go` has been chosen as there exists an implementation of multipath-quic for it [32] [33]. `quic-go` currently supports Google-QUIC and strongly evolves towards IETF-QUIC. This chapter first presents implementation details and benchmark tests of the three considered FEC Schemes. It then presents implementation details about the Unreliable Streams.

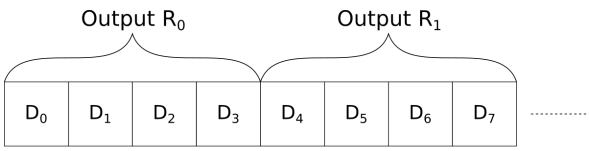
4.1 Forward Erasure Correction

In this section, we describe how the Forward Erasure Correction has been implemented in `quic-go`. This extension has been implemented on the basis of the FEC Framework defined in section 2.5. Three FEC Schemes have been implemented: *XOR*, *Reed-Solomon* and *Convolutional RLC*.

4.1.1 XOR FEC Scheme

The XOR FEC Scheme has been implemented by hand as its principle is quite simple: the Source Symbols are simply XORed with each other to generate a Repair Symbol. There is no need to encode data in the *FEC Scheme-Specific* field of the FEC Repair Payload ID of the Block FEC Framework. The implementation has been inspired from the implementation of the XOR function in the `crypto/cipher` Go package: when the underlying CPU architecture supports unaligned operations, the XOR is performed on batches of multiple bytes which fastens the computations, while it is computed byte-by-byte otherwise.

The XOR FEC Scheme itself can only recover the loss of one Repair Symbol. While this can be sufficient for a uniform loss model and a relatively low loss rate, this is insufficient when we consider high loss rates or when losses occur in bursts. This is why we consider the use of interleaving to recover from burst loss with the XOR FEC Scheme. Our implementation of the FEC Framework allows to interleave the FEC Blocks of successive packets. Sending successive packets in different FEC Blocks enables limited FEC Schemes such as XOR to better handle burst loss. Figure 4.1 shows the sequence of Source and Repair Symbols sent with a $(5, 4)$ XOR FEC Scheme without interleaving. We can see that only the loss burst of $R_0\, D_4$ can be recovered and any other loss of two symbols cannot. On the other side, Figure 4.2 shows the sequence of Source and Repair Symbols sent with a $(5, 4)$ XOR FEC Scheme, interleaving the packets with two different FEC Blocks. We can now see that losses of two consecutive symbols can be recovered by the FEC Scheme: the two lost symbols are spread in the two FEC Blocks as a single symbol loss.



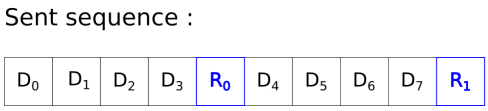


Figure 4.1: $(5, 4)$ block code used without interleaving: only one burst of 2 packets can be recovered

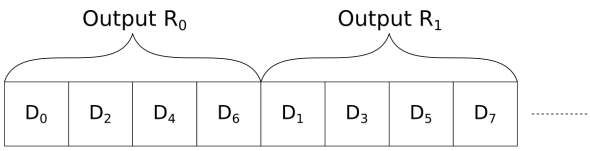


Figure 4.2: $(5, 4)$ block code used with interleaving: many bursts of 2 packets can be recovered

However, interleaving also has some drawbacks. First, it delays the computation and transmission of the repair symbols compared to the non-interleaved case. This leads to a delayed loss recovery compared to the non interleaved case when we consider the loss of a single symbol. In our example, the single loss of the D_2 symbol is recovered after the reception of D_3 and R_0 in Figure 4.1 while in Figure 4.2 it is recovered after the reception of $D_3\, D_4\, D_5\, D_6\, R_0$, which can induce a high delay when the symbols are spaced in time. Another drawback of interleaving is that although it can handle most of the burst losses of size b, it cannot handle all possible loss of b symbols. In our interleaved example, the loss of symbols D_1 and D_3 cannot be recovered while it would have been recovered with any $(10, 8)$ Block FEC Scheme that provides the same code rate as the interleaved XOR. Furthermore, there are some bursts that cannot be recovered. In our example, the loss of D_6 and R_0 in Figure 4.2 cannot be recovered. Our implementation allows thus to handle burst losses with a simple interleaved XOR FEC Scheme but its recovery capabilities are not optimal compared to other block FEC Schemes with a similar code rate. Using interleaved XOR can however be interesting as the XOR FEC Scheme is both easy to implement and fast to compute.

4.1.2 Reed-Solomon FEC Scheme

In addition to the XOR FEC Scheme, our implementation also allows to use a Reed-Solomon FEC Scheme to generate the redundancy. The implementation of the FEC Scheme relies on the `klauspost/reedsolomon` Reed-Solomon implementation available on GitHub[1]. It provides optimized encoders and decoders that exploit the instruction sets of different CPU architectures, with routines directly written in assembly. This implementation provides a systematic block code that can take up to 256 Source Symbols as input and generate up to 256 Repair Symbols. The FEC Scheme does not encode any data in the *FEC Scheme-Specific* field of the Repair FEC Payload ID.

As our Block FEC Framework does not settle the (n, k) parameters of the FEC Scheme in order to allow to adapt the level of redundancy to the connection characteristics, these parameters could potentially change frequently. In the `klauspost/reedsolomon` library, this requires to create a new Reed-Solomon encoder for each block with the correct parameters. This can slow down the encoding/decoding procedure drastically as the encoder construction requires intensive computations such as matrix inversion and multiplication. We thus added a caching mechanism that keeps in memory each Reed-Solomon encoder that has been built with a specific set of parameters. This mechanism can fasten the encoding and decoding process when the parameters of the FEC Scheme stay similar during the connection. However, this can require a large amount of memory when many encoders are created.

[1] https://github.com/klauspost/reedsolomon, available under the MIT license

4.1.3 Convolutional Random Linear Code FEC Scheme

As convolutional FEC Schemes provide different properties from Block FEC Schemes, our implementation also allows their use through the implementation of the Convolutional RLC FEC Scheme. The implementation has been written by hand given that the RLC error correcting code is rather simple and there is no known implementation in Go. Our implementation relies on the `alex-ant/gomath` linear equations system solver from GitHub[2]. It has been modified in order to handle symbols of arbitrary size and to extend its recovery capabilities to solve non-square systems that can contain solvable square subsystems. Our implementation is inspired from the *FECFRAME* RLC FEC Scheme draft [26] . For each Repair Symbol, it defines the following *FEC Scheme-Specific* fields used in the Convolutional FEC Framework :

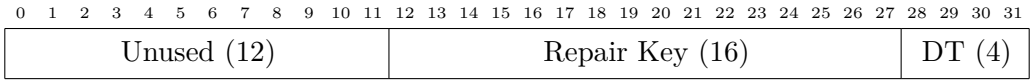


Figure 4.3: FEC Scheme-Specific field of the Convolutional RLC FEC Scheme

The *Repair Key* field is chosen by the FEC Scheme at the sender side. This value is used as a seed for the Pseudo-Random Number Generator of the FEC Scheme. As described in [26], our implementation uses the Park & Miller PRNG [34] to generate the random equation coefficients. The Park & Miller function used by our implemented FEC Scheme follows the draft and is thus the following :

$$I_{j+1} = 7^5 * I_j \pmod{(2^{31} - 1)}$$

with I_0 being the seed of the PRNG. As of writing this thesis, the fifth version of the draft has been released and advises to change the PRNG [35] . This will be done in the future.

The *Repair Key* field is a non-zero, monotonically increasing value. It must be different for all the repair symbols that have been generated from the same encoding window, otherwise, the generated equations will be co-linear as their coefficients will be identical. In our implementation, the *Repair Key* value increases by 1 for each Repair Symbol and loops back to 1 when the maximal value of the field is reached.

The *DT* field is the *Density Threshold* parameter of the FEC Scheme. It represents fraction of coefficients that are non-zero in the generated equation. Its value can go from 0 to 15 (15 meaning that all coefficients are non-zero). A high value provides better recovery capabilities but will make the recovery more intense from a computational point of view for large encoding windows.

The generation of the coefficients follows the algorithm defined in the section 3.5 of [26]. Our implementation generates coefficients in the Galois field $GF(2^8)$, i.e. coefficients between 0 and 255.

4.1.3.1 Optimizations at the receiver side

The receiver-side FEC Scheme must regularly solve systems of linear equations. Our solver library uses the Gaussian elimination method to echelon a matrix and thus solve a linear system. Gaussian elimination has a $\mathcal{O}(n^3)$ complexity for an $n \times n$ matrix which can become costly when we consider real-time communications that try to reduce the delay to recover packets. Furthermore, while the coefficients of the linear equations belong to $GF(2^8)$ and can be represented with one byte, the constant terms of the equations (i.e. the Repair Symbols) have the size of a QUIC packet, which can be more than one thousand bytes. Performing arithmetic operations on such

[2] https://github.com/alex-ant/gomath, available under the MIT license

large symbols also slows down the recovery process. This is why the FEC Scheme should avoid to echelon the matrix of the linear system if there is no chance to find a solution. The matrix representing the linear system will thus be put in echelon form only if one of the following holds :

- The system has n equations and concerns n variables with at least one non-zero coefficient.
- The system contains a subsystem with m equations, concerning m variables that have at least one non-zero coefficient.

For example, the following system does not trigger a Gaussian elimination process :

$$\begin{cases} 1d_1 + 2d_2 + 3d_3 = r_1 \\ 4d_1 + 5d_2 + 6d_3 = r_2 \end{cases}$$

On the other hand, the following system will trigger a Gaussian elimination process as there exists a subsystem of two equations concerning the two variables d_3 and d_4 :

$$\begin{cases} 1d_1 + 2d_2 = r_1 \\ 3d_3 + 4d_4 = r_2 \\ 5d_3 + 6d_4 = r_3 \end{cases}$$

4.1.4 Benchmarks

This section shows the results of benchmark tests for each implemented FEC Scheme to see their applicability in real-time scenarios. The benchmarks have been performed on a *Intel(R) Core(TM) i5-3337U* CPU with a maximal frequency of 2.7Ghz. This 5 years old laptop CPU can give an idea on how the FEC Schemes implementation behaves on rather modest laptops. The operating system is *Fedora 28* using the 4.16.6 *Linux* kernel. Each Source Symbol consists in a random array of 1000 bytes. The tool used to perform benchmark is the `go test` tool provided with the Go language.

We perform benchmarks for each FEC Scheme with different levels of redundancy and report the time in nanoseconds needed to perform one encoding/decoding operation as well as the coding speed in MegaBytes per second. The coding speed is computed by looking at how many bytes the FEC scheme can take as input per second. The coding speed metric must thus be taken carefully in the case of decoding. Erasing more symbols will indeed reduce the encoding speed in two ways: by reducing the size of the input and making the recovery procedure more difficult.

Each benchmark test consists in the recovery of a fixed number of erased symbols. The number of erased symbols is the maximum number of erasures that the FEC Schemes can handle. For the RLC FEC Scheme, it is set to $\lfloor \frac{1}{3} \rfloor L$ where L is the length of the window. Table 4.1 shows that the encoding operation can handle quite large windows while keeping the execution time in the order of a few microseconds. Furthermore, we can see that the execution time seems to evolve linearly with the length of the encoding window. The encoding operation can support a coding speed of more than 200 MB/s. As shown in Table 4.2, the RLC FEC Scheme symbols decoding is significantly more intense in a computational point of view when the decoding window grows. This is due to the use of Gaussian elimination to solve systems of linear equations whose constant terms are each 1000 bytes long. Gaussian elimination implies multiple linear combinations of the different Repair Symbols and can make the linear system solving quite slow when the number of equations and unknowns variables increase. The decoding operation is thus more intense to compute and its coding speed can decrease down to 10 MB/s

(n, k, L)	ns/op
$(3, 2, 4)$	21260
$(3, 2, 8)$	40275
$(3, 2, 20)$	97472
$(3, 2, 40)$	192123

Table 4.1: Encoding with RLC

(n, k, L)	ns/op
$(3, 2, 4)$	15332
$(3, 2, 8)$	41986
$(3, 2, 20)$	353654
$(3, 2, 40)$	2211082

Table 4.2: Decoding with RLC

Tables 4.3 and 4.4 show the results of the benchmark tests on the XOR FEC Scheme, while Tables 4.5 and 4.6 show the results with the batch optimization disabled. We can see that performing word-by-word instead of byte-by-byte operations can reduce drastically the time needed to encode and decode the symbols. The encoding and decoding operations can reach a coding speed of 2 GB/s with batch operations, while it can only reach a speed of 450 MB/s without it.

(n, k)	ns/op
$(5, 4)$	2031
$(9, 8)$	4282
$(21, 20)$	11431
$(41, 40)$	23454

Table 4.3: Encoding with XOR

(n, k)	ns/op
$(5, 4)$	1863
$(9, 8)$	3504
$(21, 20)$	8097
$(41, 40)$	16176

Table 4.4: Decoding with XOR

(n, k)	ns/op
$(5, 4)$	9013
$(9, 8)$	20196
$(21, 20)$	54198
$(41, 40)$	110603

Table 4.5: Encoding with XOR without batch operations

(n, k)	ns/op
$(5, 4)$	8958
$(9, 8)$	18939
$(21, 20)$	49672
$(41, 40)$	103315

Table 4.6: Decoding with XOR without batch operations

Tables 4.7 and 4.8 show the performance of the Reed-Solomon FEC Scheme with the encoder caching optimization, while Tables 4.9 and 4.10 show the results with caching disabled. This clearly shows that the initialization of a Reed-Solomon encoder is a heavy operation from a computational point of view. It is worth noting that the encoding/decoding operations of the Reed-Solomon error-correcting code are fast thanks to the use of optimized routines written in assembly. With caching enabled, the Reed-Solomon FEC Scheme can reach a coding speed of 2 GB/s for the encoder and 1.2 GB/s for the decoder. With caching disabled, it can only reach a speed of 900 MB/s for the encoder and 450 MB/s for the decoder.

(n, k)	ns/op
$(5, 4)$	1757
$(6, 4)$	2738
$(13, 10)$	6998
$(20, 15)$	14923
$(50, 30)$	101043

Table 4.7: Encoding with Reed-Solomon

(n, k)	ns/op
$(5, 4)$	3287
$(6, 4)$	4340
$(13, 10)$	8590
$(20, 15)$	16239
$(50, 30)$	145624

Table 4.8: Decoding with Reed-Solomon

(n, k)	ns/op
$(5, 4)$	4522
$(6, 4)$	5756
$(13, 10)$	21458
$(20, 15)$	54276
$(50, 30)$	99370

Table 4.9: Encoding with Reed-Solomon without caching

(n, k)	ns/op
$(5, 4)$	8444
$(6, 4)$	9660
$(13, 10)$	28685
$(20, 15)$	66868
$(50, 30)$	431735

Table 4.10: Decoding with Reed-Solomon without caching

4.1.5 Example usage of the FEC extension

This section presents an example of usage of the Forward Erasure Correction extension in `quic-go`. The example opens a QUIC session and specifies the Reed-Solomon FEC Scheme as receiver-side FEC Scheme. The FEC Scheme is directly defined at the opening of the session as a configuration parameter. The code in the example finally defines a constant level of redundancy. The underlying application can also define its own redundancy controller to update the parameters of the FEC Scheme dynamically after each packet loss.

```
// initialize the QUIC config
quicConfig := &quic.Config{
              FECScheme: quic.ReedSolomonFECScheme,
              /* insert here any other needed QUIC
                 config parameters */
          }
// open the session
session, err := quic.DialAddr(addr, TLS_CONFIG, quicConfig)
if err != nil {
        return err
}
// define the level of redundancy used for this communication
rc := fec.NewConstantRedundancyController(NUMBER_OF_SOURCE_SYMBOLS,
                                          NUMBER_OF_REPAIR_SYMBOLS,
                                          NUMBER_OF_INTERLEAVED_BLOCKS,
                                          WINDOW_STEP_SIZE)
session.SetRedundancyController(rc)
```

4.2 Unreliable Streams

We describe here our implementation of the Unreliable Streams in `quic-go`. Unreliable Streams are designed for the delivery of delay-sensitive data. They guarantee in-order delivery of the data, but will skip missing data when their expiration deadline has expired. Unreliable Streams in `quic-go` can be used in a similar way to normal QUIC Streams.

4.2.1 Buffering

In order to be able to perform data recovery, a delay should be added between the reception of stream data and their delivery to the application. This takes the form of a buffer at the stream level. An Unreliable Stream does not deliver the byte at offset o to the application while its buffer has not received the byte at offset $o + d$ or greater. d is the *desynchronization* in

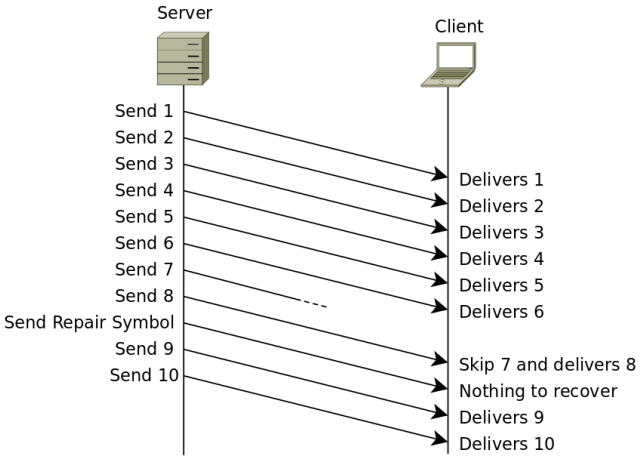


Figure 4.4: Unreliable transmission without buffering

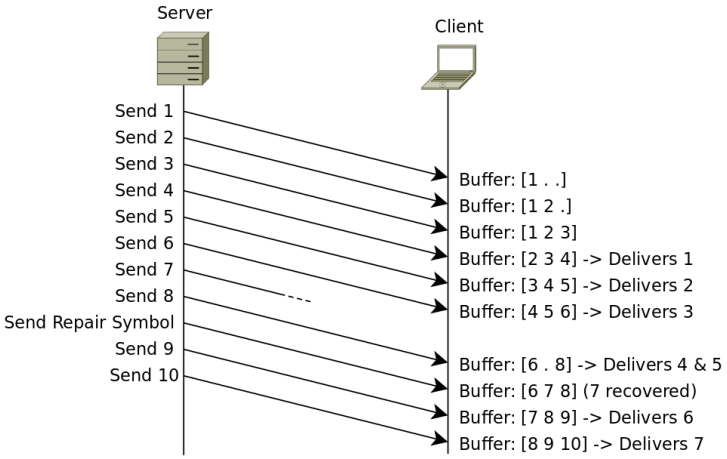


Figure 4.5: Unreliable transmission with buffering

bytes induced by the buffer and is also the size of the buffer. The application can now use the stream buffering to induce a delay that is sufficient to recover lost frames, given the bitrate of the data stream and some time desynchronization constraints that are specific to the application.

Figure 4.4 shows an unreliable transmission that does not use buffering, while Figure 4.5 shows an unreliable transmission using buffering. The receiver of Figure 4.4 receives the data earlier as it does not need to fill its buffer. The perceived delay is thus lower. However, if data are lost, the receiver won't be able to recover it excepted if the Repair Symbol(s) arrive immediately after the lost data. This is the problem of the transmission of Figure 4.4 where symbol 7 is lost and won't be delivered to the application. In Figure 4.5, the receiver will deliver the data later to the application because of the buffer. The perceived delay is thus higher. On the other hand, the receiver will be able to recover the lost Source Symbol thanks to the received redundancy. Buffering is thus necessary to recover lost data but imply a higher perceived delay. Upon the reception of a stream frame with the FIN bit set, the buffer authorizes to read all the remaining data to empty the stream buffer.

4.2.2 Retransmission and reliability deadlines

Unreliable Streams can receive a retransmission deadline that represents the time during which their data is worth to be retransmitted. The QUIC sender will retransmit every Unreliable Stream frame whose retransmission deadline is not over. The Unreliable Streams thus behave as normal QUIC streams while the retransmission deadline of their data is not over.

The application can also give a reliability deadline to a stream. On the receiver-side, an Unreliable Stream will not skip unreceived data upon a `Read()` call before the reliability deadline has passed from the moment of the call. The application can thus adjust the reliability deadline to the circumstances: if the application considers that it can stand 1 second before receiving the next data, it can set the reliability deadline to 1 second and then call the `Read` function. It will thus deliver the data as soon as possible while avoiding to skip the missing data before it has arrived.

4.2.3 Message mode

In most cases, an application will send structured data through a stream instead of a raw stream of unstructured data. An Unreliable Stream skipping an arbitrary number of bytes could confuse and mislead the application in its parsing of the structure by skipping for example some

important encoded fields. This is why our implementation of the Unreliable Streams allows to set an Unreliable Stream in *message mode*. Upon a call to the `Write()` function, a stream in message mode will consider the provided payload as a whole message. The stream frames of a stream in message mode always contain one entire message. This implies that a message will always be either entirely received or entirely lost. This allows the application to define a structure for each message and deal more easily with message losses, without being misled in their parsing of the messages.

4.2.4 Example usage of the Unreliable Streams

Hereunder is an example usage of `quic-go` with Unreliable Streams. This example uses all the features described in the Section 4.2. It opens an Unreliable Stream and sends two messages. Finally, it outputs every received message and closes the stream. All packets containing an Unreliable Stream frame are FEC-protected.

```
var session quic.Session = ...
stream, err := session.OpenStreamSync()
if err != nil {
        panic(err)
}
// make the stream unreliable
stream.SetUnreliable(true)
// the lost stream frames will be retransmitted during 500ms from
// their first transmission
stream.SetRetransmissionDeadline(500*time.Millisecond)
// If we call Read() and there is no data, the call will block
// during at maximum 1 second. Unreceived data will be skipped
// after 1 second of blocking.
stream.SetReliabilityDeadline(1*time.Second)
// set the stream in message mode
stream.SetMessageMode(true)
// Send two messages. The receiver will receive either both messages,
// one of them or none of them.
stream.Write([]byte("hello"))
stream.Write([]byte("world"))

var receivedData [1200]byte
for {
        n, err := stream.Read(receivedData[:])
        if err != nil && err != io.EOF {
                panic(err)
        }
        fmt.Printf("received %+v", receivedData[:n])
        if err == io.EOF {
                stream.Close()
                break
        }
}
```

Chapter 5

Experiments

The design described in this thesis is currently implemented and experiments have been performed to assess its performances. Those are explained in this chapter.

5.1 Design of the experiments

This section describes the experiments that have been performed on the solutions we developed. In order to emulate different network conditions, we used the Mininet tool [36], taking the benefits of OS-level virtualisation to allow network emulation on constrained resources.

5.1.1 Loss Model

Experimenting with networks is not new and there currently exists a wide variety of scientific papers on the subject. Some of them consider either uniform random losses, short burst sizes or pre-defined uniform loss rate variation to represent loss bursts, such as [37], [38], [39] or [40] (which considers an average loss burst length of 2). Others consider a burst loss model with potentially long loss bursts [24] [41] [42] . While small burst lengths or uniform losses can already give an idea on the efficiency of a solution, we advocate for looking at longer burst lengths in order to evaluate the full capacity of our solution and to compare different types of codes under different conditions. The study of the efficiency of Forward Erasure Correction in a real-time scenario would benefit from using a loss model allowing potentially long error bursts. It indeed has a direct impact on the delay induced by the level of redundancy needed to handle the bursts.

The first loss model that we consider in our experiments is a two-states Markov model, depicted in Figure 5.1. The two states are the *Good* (G) and *Bad* (B) states. In the G state, the transmitted packet arrives correctly to the receiver. In the B state, the packet is lost and does not reach the receiver. In the model, p denotes the transition probability from the G to the B state and r the transition probability from the B to the G state. This model allows us to represent a lossy channel in which the losses occur in bursts where the average burst length is $\frac{1}{r}$. A transition is performed for each packet sent on the network. This model has the advantage to represent a lot of possible configurations while being simple with only 2 varying parameters.

A generalization of this model is known as the Gilbert-Elliott [43] model, shown in Figure 5.2. This model adds two parameters, k and h. In the G state, the transmitted packet arrives correctly with probability k. In the B state, the packet is lost with probability $1 - h$. The two-states Markov model of figure 5.1 is identical to a Gilbert-Elliott model with $k = 1$ and $h = 0$.

As Mininet only provides a uniform loss model, we patched it to use the Gilbert-Elliott loss model provided by the `netem` Linux tool [44] .

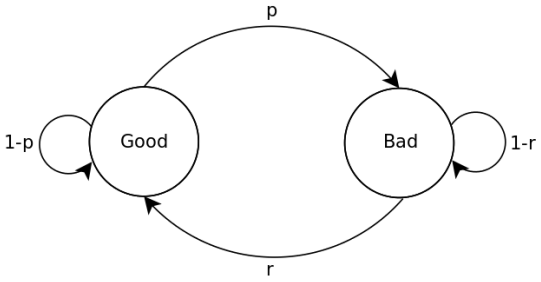


Figure 5.1: Two-states Markov loss model

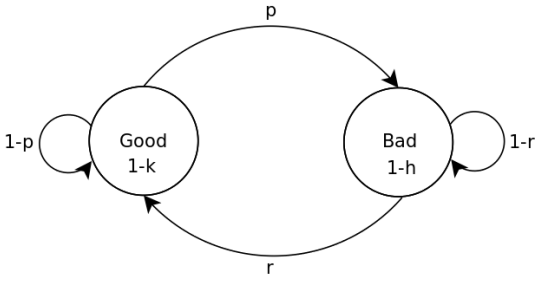


Figure 5.2: Gilbert-Elliott model

5.1.2 Experiment parameters

Although the protocol can control different parameters during the transmission, such as the amount of redundancy sent by the FEC Framework, some parameters are uncontrolled and have a direct impact on the quality of the transmission. These parameters can typically be the delay of the connection and the loss probability distribution. In order to have a global confidence in the tested design, these parameters should be tested either with a set of relevant values when possible, and otherwise with a large range of possible values. This is why we use the *experimental design* approach for our experiments.

5.1.3 The experimental design approach

The *experimental design* allows to address the problem of having a lot of uncontrolled parameters, without knowing exactly their typical real-life values. It consists in defining ranges of possible values for each parameter and performing a series of experiments with random values chosen within these ranges. This will sub-sample the ranges of values and give a global overview of the possible values taken by all the parameters. In addition to providing a general confidence concerning the performances of the tested implementation, it mitigates the bias in the parameters selection by the experimenter. This bias could have led to selecting unrealistic parameters values or avoiding special cases in which the implementation would perform poorly. Using the experimental design approach for this work has also permitted to locate different bugs that were appearing only with some specific parameter values.

While the litterature provides numerous examples of experimental frameworks, many of them did not choose the approach of experimental design [37] [38] [45] . It has however grown in popularity during the last few years: more and more articles perform experiments that cover a wide range of parameters values [33] [39] [46] [47] . The major drawback of the experimental design is the need of a high amount of experiments in order to sample correctly the ranges of parameters values. This problem can however be mitigated by using space-filling algorithms that ensure a well-distributed selection of parameters values even with a rather small number of tests.

We thus perform an overall analysis of the performances of our implementation, using experimental design along with the WSP space-filling algorithm [48] to avoid running too heavy experiments.

5.1.4 Chosen parameters ranges

Table 5.1 shows the chosen parameters ranges for the different monitored parameters. Besides the Gilbert-Elliott model and the simplified two-states model, some experiments have been performed with a uniform loss model, whose loss probability lies in the range [0, 0.03], in line with the range proposed by Paasch *et al.* [46]. It must be noted that the h and k values of the Gilbert-Elliott model have not been chosen to have an average amount of losses equivalent to the simplified model, but to represent a wider range of network conditions. In particular, the values chosen for h and k result in more losses on average compared to the simplified two-states

Parameter	Smallest value	Highest value
p	0	0.01
r	0.025	0.5
k (when Gilbert-Elliott is used)	0.97	1
h (when Gilbert-Elliott is used)	0	0.4
loss rate (when considering uniform losses)	0	0.03
one-way delay (milliseconds)	0	100

Table 5.1: Parameter ranges for the experimental design

Markov model. The one-way delay ranges from 0 to 100 milliseconds, leading to a maximum round-trip-time of 200 milliseconds. This is not completely in line with the measurement-based study of Zhang *et al.* [49]. In this study, although the majority of the measured round-trip-times is between 0 and 200 milliseconds, some of them can reach 400 milliseconds in some cases. We thus perform in this chapter a univariate analysis targeted on the delay to show whether high delays have a negative influence on our solutions.

When it is not otherwise specified, the chosen FEC Scheme is the Reed-Solomon FEC Scheme and the level of redundancy is set to $(30, 20)$. This ensures a code rate of $\frac{2}{3}$ and a burst recovery capability of 10 symbols. By looking at the possible values for the r parameter in Table 5.1, we can see that we will often be in configurations where the average burst length is longer than 10 packets, possibly leading to poor performances of our solution. This is done on purpose for multiple reasons. First, with experimental design, we want to assess the global performances of our solution, even for the cases that are not in favour of it. Second, when using the Gilbert-Elliott model, an average burst of n packets in the B state does not imply that the average number of packets lost in the burst will be n too. We thus want a reasonable margin to reach the undesirable cases even with a Gilbert-Elliott model. Finally, having an average burst length longer than the recovery capabilities of our solution will allow us to take a look at the benefits of multipath in terms of symbols recovery. Although the majority of experiments focus on these parameter values, we also explore other values for some experiments.

5.1.5 Behaviour of a test

A test simulates a real-time transmission between a sender and a receiver with bursty traffic. Its conception has first been inspired by looking at network traces produced by real-time video communication applications such as the *Facebook* video calls. It revealed that the traffic of such applications tends to be bursty by sending video frames at regular intervals instead of a traffic only composed with equally-spaced packets. We then simplify the traffic by ignoring sound and video frames compression, as it facilitates the simulation implementation and the interpretation of the results. We finally base our model on what is provided by `ffmpeg` [50] with a relatively high bitrate and framerate in order to send a sufficient amount of packets to encounter losses in a reasonably low amount of time with our Markov models. Using `ffmpeg` with the following parameters will reproduce the traffic we consider with a good fidelity, sending it over UDP at the IP address `127.0.0.1` on port `1234` :

```
ffmpeg -re -i video.mp4 -framerate 30 -s 640x480 -c:v mpeg4 -b:v 2M \
-maxrate 2M -f avi -an -g 1 udp://127.0.0.1:1234?pkt_size=1000
```

We choose a maximum payload size of 1000 bytes to prevent the fragmentation of our Repair Symbols into multiple FEC Frames, as advised in section 2.4.2.1. In order to avoid any confusion between video frames and QUIC frames, we will use the word *application message* to refer to video frames.

The sender regularly sends application messages composed of several QUIC packets at a specified number of application messages per second. The sender waits for the correct amount of time between each message. In our experiments, we use 30 messages per second, with 8 QUIC packets per message, which implies that the application messages are spaced by ~ 33 milliseconds. Figure 5.3 illustrates the shape of the traffic in our experiments. Each experiment runs during 25 seconds, in order to encounter losses even with low loss probabilities.

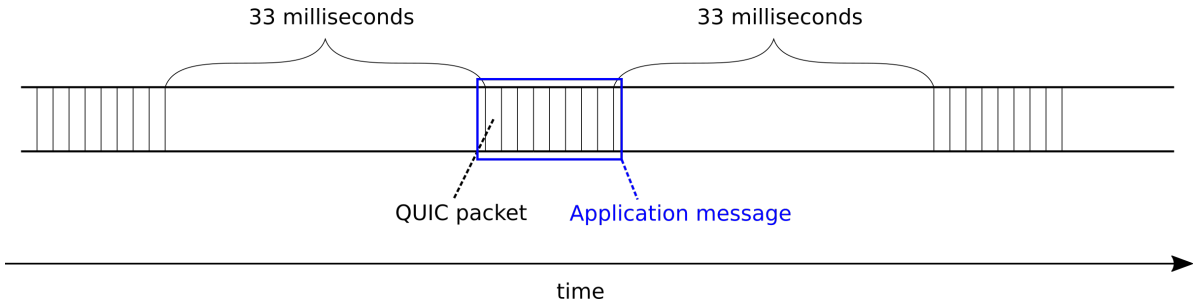


Figure 5.3: The behaviour of a test.

On the other side, the receiver, which is aware of the number of messages per second, reads the application messages at the same rate and report statistics about the user experience. A message is considered well-received when all its parts have been received. It is considered as corrupted if one or more parts are missing. When the receiver cannot read a message on time, either because it is received too late or corrupted, the receiver reports a re-buffering. The duration of the re-buffering depends on the number of successive messages that could not be delivered.

In order to directly focus on the recovery capabilities of our solutions, the congestion control has been disabled for the experiments. Indeed, a pacing-based congestion control as the one used by QUIC [51] is inappropriate for real-time communications [52] as it will delay packets although they have a short expiration deadline. A better way to avoid congestion would be to let the application adapt its sending rate (e.g. by compressing its video frames for a video application) when congestion appears.

A constant desynchronization is applied by the receiver in order to have enough time to recover the lost data. This is done by using the added stream buffering capabilities in `quic-go`: if the buffer does not contain stream data having a sufficiently high byte offset to respect the de-synchronization, it will be blocking (Figure 5.4). When using an unreliable transfer, the lost data at the head of the buffer will be skipped if we already received data with a sufficiently high byte offset (Figure 5.5). Unless otherwise specified, the experiments use a desynchronization buffer with the size of 4 application messages. The application uses the buffer in such a way that it will be able to read one full application message as soon as the first bytes of the fourth application message are present in the buffer. The buffer will thus induce a desynchronization of 100 milliseconds. Of course, we also analyse the impact of the desynchronization buffer on the recovery capabilities of our solutions. In order to be fair, the same desynchronization is also applied for the experiments with UDP and the fully reliable QUIC.

5.1.6 Reported metrics

For each test, different metrics are measured and reported in a database as a result of the test. Hereunder are described the reported metrics for each test. Each test is run three times and the reported value is the median value between the three runs.

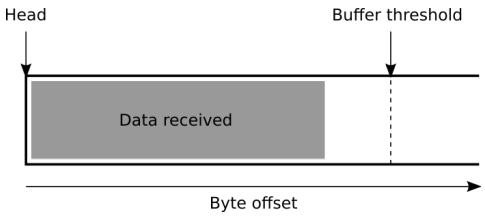


Figure 5.4: Example of blocking buffer: there is no stream data whose byte offset is above the threshold. The desynchronization is thus not respected. The buffer is blocking until new data are received.

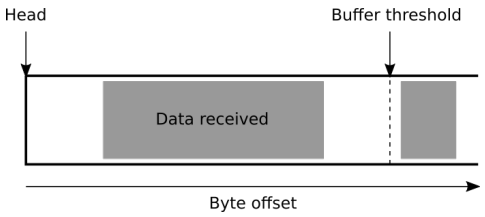


Figure 5.5: Example of non-blocking buffer: there is stream data whose byte offset is above the threshold. The missing data at the head will be skipped, and both the head and the buffer threshold will move to the right upon a `Read()`.

5.1.6.1 Total re-buffering time

This metric is a good indicator of the quality of experience perceived by the end-user as it represents the total wasted time during the connection. With a rate of x application messages per second, a lost or corrupted application message will imply a re-buffering of $\frac{1}{x}$ seconds (33 milliseconds in our experiments). When using an unreliable data transfer, the total re-buffering time of a connection thus indicates how many application messages have been lost or corrupted.

5.1.6.2 Amount of received application data

While in the case of fully reliable transfer, we can expect to finally receive all the sent data, in the unreliable case (e.g. when using Unreliable Streams or UDP), some parts of the data could be completely lost. The Forward Erasure Correction will increase the amount of received data over an Unreliable Stream, although it does not give the guarantee to receive the totality of the sent data. This metric reports the amount of data received by the application from its data stream. If the application only receives a part of an application message, this part will still be taken into account in the amount of data received. A re-buffering will however be reported in that case, as the application message has been corrupted. Having a scenario where the majority of the sent data has been received and a high total re-buffering time has been reported indicates that many application messages have been corrupted with only a small part of missing data. For example, a transmission during which only one packet composing an application message is lost will result in the same total re-buffering time as a transmission during which one full application message is lost. In this case, the total re-buffering time will be the same but the first transmission has received more data than the second one. This is thus an interesting metric to observe in order to compare transfers with and without FEC or fully and partially reliable transfers.

5.2 Overhead comparison of the different solutions

In order to be more aware of the behaviour of each solution, we want to compare the overhead of data that they need to send. We thus performed experiments using our simplified two-states Markov model of Figure 5.1. Every sent byte is reported, including the bytes of the QUIC header. Figure 5.6 shows the Empirical CDF (ECDF) of the ratio of the total amount of data sent by a specific solution to the amount of data sent by the UDP solution, which only has a small packet header. We compare QUIC-FEC with two FEC Schemes having a code rate of $\frac{2}{3}$: a $(30, 20)$ Reed-Solomon and a $(3, 2, 20)$ convolutional RLC FEC Scheme.

The total amount of data sent by the reliable QUIC solution is always higher than UDP. This is due to the overhead of the QUIC Public Header and the header of the stream frames

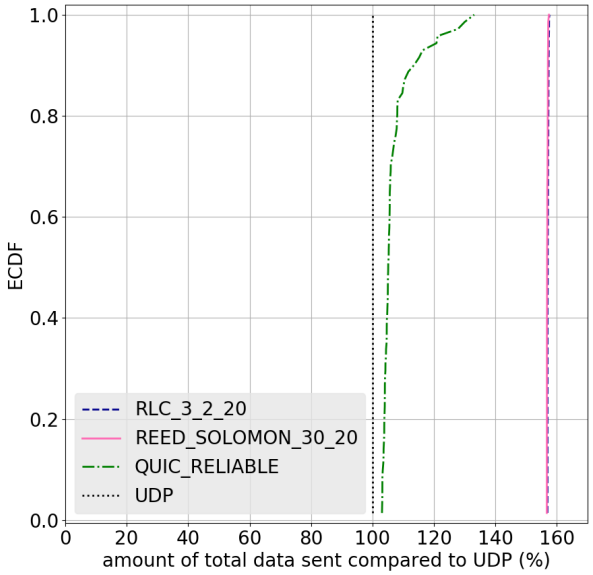


Figure 5.6: UDP VS Reed-Solomon VS RLC VS Reliable QUIC: amount of data sent compared to the UDP solution

that contain the application data. It also seems that the overhead of the reliable QUIC solution varies in the experiments. By looking at our results, this overhead is high when either the r parameter of our loss model is low or the p parameter is high. This overhead variation is indeed due to the retransmissions of the lost data that have been sent by the reliable QUIC.

We can see on the same figure that the overhead of QUIC-FEC with Reed-Solomon and Convolutional RLC stays constant, with an overhead between 50% and 60% compared to UDP. This is a little bit more than the theoretical overhead of 50% represented by the code rate of our FEC Schemes ($\frac{2}{3}$ for each FEC Scheme). This difference is due to the additional FEC-related bytes present in the Public Header and in the FEC Frame header.

5.3 The benefits of QUIC-FEC compared to UDP

In this section, we evaluate the interest of using Forward Erasure Correction with an unreliable data transfer by comparing QUIC-FEC + Unreliable Streams with UDP with our Gilbert-Elliott model. The ECDF of the amount of data received for UDP and QUIC-FEC + Unreliable Streams are shown in Figure 5.7. We can see a clear advantage of QUIC-FEC in terms of the amount of data received compared to UDP. Furthermore, as we can see in Figure 5.8, not only the amount of data received is significantly higher with FEC but the total re-buffering time is drastically reduced, meaning that fewer application messages are delayed, lost or corrupted. Figure 5.9 shows the ratio between the amount of data received with QUIC-FEC and the amount of data received with UDP.

Although the vast majority of the ratios is higher than 1, some points indicate that some experiments ended with more data received with UDP than QUIC-FEC. By looking at the results, this always occurs in parameter configurations with $r \leq 0.06$, which gives an average burst length significantly longer than the number of packets that can be recovered with our $(30, 20)$ block code. The loss bursts were thus too long to be recovered by FEC, making QUIC-FEC comparable to

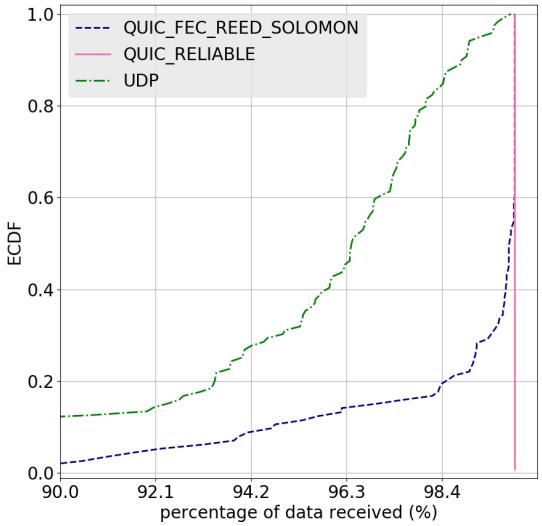


Figure 5.7: UDP VS Reed-Solomon VS Reliable QUIC: amount of data received with our Gilbert-Elliott model

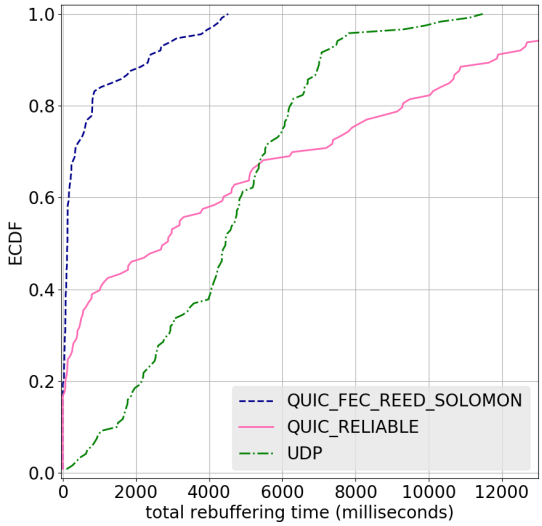


Figure 5.8: UDP VS Reed-Solomon VS Reliable QUIC: total re-buffering time with our Gilbert-Elliott model

UDP, with potentially worse results in some cases.

5.4 The benefits of FEC and partial reliability compared to reliable transfer

In this section, we evaluate the benefits of QUIC FEC + Unreliable Streams compared to a fully reliable transport in a real-time communication scenario. Figures 5.7 and 5.8 show the ECDF of the amount of data received and the total re-buffering time for a reliable QUIC data transfer. Unsurprisingly, we can see in Figure 5.7 that all the sent data are received by the reliable receiver. However, by looking at Figure 5.8, we can see that the reliable QUIC transfer encounters a lot more re-bufferings than QUIC-FEC with our Reed-Solomon block code. This means that although all application messages are received thanks to the retransmissions, an important part of the messages are received too late and will imply a re-buffering on the receiver application. Furthermore, not only these application messages are received too late, they also prevent the other application messages from being read on time, even if they have not been lost.

Another point that is worth to be noted is that the curves of reliable QUIC and UDP cross at some point in Figure 5.8. After taking a closer look at our results, the samples on the reliable QUIC curve that are to the right of the UDP curve concern parameters configurations with a high delay. In that case, the desynchronization of 4 application messages used by the application may not be sufficient to receive the retransmission of the lost data on time. We will analyse the behaviour of reliable QUIC with different delays more in details in the next section. On the other side, the UDP receiver simply skips the lost frames without blocking furthermore, which gives a smaller re-buffering compared to reliable QUIC when the delay is high. QUIC-FEC tries to gather the best of both worlds in a real-time scenario by trying to recover the lost Unreliable Stream frames with FEC and skipping the Unreliable Stream frames that could not be recovered on time. This confirms that a fully reliable data transfer may be problematic in a real-time transmission use-case.

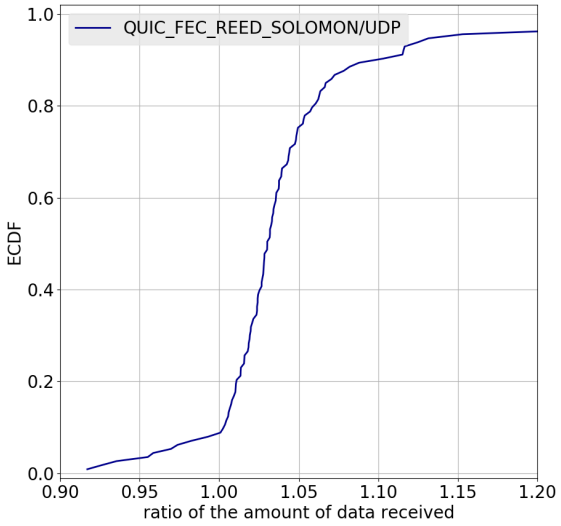


Figure 5.9: UDP VS Reed-Solomon: $\frac{\text{QUIC-FEC}}{\text{UDP}}$ ratio of the amount of data received with our Gilbert-Elliott model.

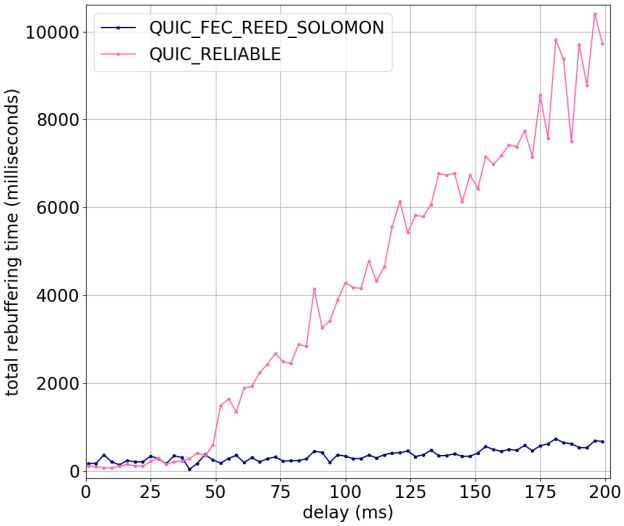


Figure 5.10: QUIC-FEC VS reliable QUIC: total rebuffering time in function of the delay with our Gilbert-Elliott model.

5.4.1 Univariate analysis: delay

Thanks to the experimental design, we have seen in the results of the previous experience that the reliable QUIC solution performs badly when the delay is high. In order to better analyse the impact of the delay on reliable QUIC and QUIC-FEC, we performed a univariate experiment. All parameters have been fixed except the delay. The value of the parameters can be seen on Table 5.2.

Parameter	Value
p	0.005
r	0.25
k	0.98
h	0.05
one-way delay (milliseconds)	0 to 200

Table 5.2: Parameter ranges for the univariate analysis on the delay

Figure 5.10 shows the total re-buffering time of reliable QUIC and QUIC-FEC in function of the delay. We can see that when the delay is low, both reliable QUIC and QUIC-FEC have a low re-buffering time. When the delay becomes higher than 45 milliseconds, the total re-buffering time of reliable QUIC increases a lot. `quic-go` considers by default that a packet has been lost after waiting $\frac{9}{8} * \text{RTT}$ since its transmission without receiving an acknowledgement for it. This means that the `quic-go` receiver will have to wait an additional time of $\frac{9}{8} * \text{RTT}$ to receive a the retransmisson of a lost stream frame. Our buffer delivers data as soon as it contains 4 application messages, which ensures a desynchronization of around 100 milliseconds. With a delay higher than 45 milliseconds, the receiver will thus have to wait more than 100 additional milliseconds before receiving the retransmission of the lost data. With a buffer of 4 application messages only providing a desynchronization of 100 milliseconds, it will imply re-bufferings at the receiver side. We can see that QUIC-FEC does not suffer from this problem as it does not

rely on retransmissions to recover lost data and skips the lost data at the head of the buffer. It is thus robust against high delays. There is however a slight increase in the total re-buffering time for QUIC-FEC with high delays. This is due to the QUIC flow control. With high delays, the sender will be blocked at the beginning of the connection, as the receive window of the receiver is full. The window extension will take more time to arrive to the sender with high delays, which increases the time during which the sender cannot send anything. This problem can be mitigated by increasing the initial receive window of the QUIC receiver. However, it will still stay vulnerable to higher delays. The only way to avoid this problem completely would be to disable the flow control.

5.5 Interleaved $(n, n-1)$ XOR VS (n, k) Reed-Solomon

In this section, we compare our interleaved $(3, 2)$ XOR FEC Scheme and our $(30, 20)$ Reed-Solomon FEC Scheme. In order to have similar burst handling capabilities, the XOR FEC Scheme uses an interleaving of 10 different Source Blocks. We thus have two block codes with the same code rates ($\frac{2}{3}$) and similar burst recovery capabilities.

5.5.1 Uniform losses

In Figure 5.11, we can see that the interleaved XOR block code recovers quite less data than the Reed-Solomon block code when we consider uniform losses. This is one of the inconvenients of interleaving: while the (n, k) Reed-Solomon can recover any loss of l packets such that $l \leq n - k$, this is not the case of our block code using interleaving. If a loss occurs on two packets of the same Source Block, it won't be recovered. This also implies a higher total re-buffering time, as we can see in Figure 5.12.

5.5.2 Gilbert-Elliott model

We can see in Figure 5.13 that the interleaved XOR FEC Scheme also seems to recover less data than our Reed-Solomon FEC Scheme when considering a Gilbert-Elliott model. Furthermore, by looking at Figure 5.14, it also seems to encounter re-bufferings more often. This is because our interleaved XOR FEC Scheme can not recover every loss burst of 10 packets, while our Reed Solomon FEC Scheme can. This implies the loss of more packets as well as more re-bufferings.

5.6 Convolutional codes VS block codes

Here, we evaluate the interest of convolutional and block codes when used for Forward Erasure Correction. Roca *et al.* [24] already perform comparisons between convolutional and block codes. We however differ from this study by sending messages composed of multiple packets instead of sending equally-spaced packets. Our traffic is thus more bursty.

In this case, we compare a $(3, 2, 20)$ convolutional Random Linear Codes (RLC) FEC Scheme and a $(30, 20)$ Reed-Solomon FEC Scheme. Both schemes have a code rate of $\frac{2}{3}$ and should be able to handle loss bursts of size 10.

5.6.1 Uniform losses

Figure 5.11 shows the ECDF of the amount of data received and Figure 5.12 shows the number of rebufferings when considering uniform losses. Both Reed-Solomon and RLC FEC Schemes have a similar result with almost every lost packet being recovered. We can note that the RLC FEC Scheme encounters fewer rebufferings than the Reed-Solomon FEC Scheme. A rebuffering caused by the loss of a unique packet is possible when the lost packet is at the beginning of a Source Block. In that case, the FEC Scheme may not be able to recover the lost

data on time, as one Source Block of 20 packets can potentially spend 4 application messages, as they are composed of 8 packets. In that case, the receiver would thus need to receive 4 application messages to be able to recover the lost data. This problem is not present in the case of the Convolutional RLC FEC Scheme thanks to the fact that convolutional FEC Schemes send the Repair Symbols interleaved with the Source Symbols: with our $(3, 2, 20)$ Convolutional RLC FEC Scheme, it is likely to have a Repair Symbol directly following the lost packet. In our configuration, we could easily increase the code rate of convolutional RLC with uniform losses without affecting much the total re-buffering time. However, we can see that there are also re-bufferings with convolutional RLC. When looking at the results, it is always when the delay is high (> 50 milliseconds). These re-bufferings are due to the QUIC flow control. When the delay is high, the sender is blocked at the beginning of the connection by the Stream-level flow control before it receives a message from the receiver to increase its receiving window.

5.6.2 Gilbert-Elliott model

Figure 5.13 shows the ECDF of the amount of data received and Figure 5.14 shows the ECDF of the total rebuffering time when considering a Gilbert-Elliott loss model. Figure 5.15 shows the ratio of the amount of data received between QUIC-FEC with the RLC FEC Scheme and QUIC-FEC with the Reed-Solomon FEC Scheme. We can see that the Reed-Solomon FEC Scheme seems to recover more data in the majority of the cases. By looking at our results for cases where the ratio is above 1.005, it concerns parameter configurations with low values for the r parameter ($< 10\%$), low values for the h parameter ($< 5\%$) or low values for both. These configurations are close to the simplified Markov Model in which all the packets are lost during a loss burst.

This reveals one of the drawbacks of convolutional FEC Schemes. As the Repair Symbols are interleaved with the Source Symbols, the Repair Symbols following a loss burst will be received progressively along with the Source Symbols, while it will arrive all at once with our block FEC Scheme. With long loss bursts, it may thus take longer for a convolutional FEC Scheme to gather all the Repair Symbols that are necessary to recover the lost Source Symbols. RLC seems however to recover more data when the average loss bursts are shorter or when some packets are received during the bursts, implying that fewer Repair Symbols will be necessary to proceed to the recovery.

5.7 Affording low desynchronizations with FEC

An interesting feature of the convolutional codes is the fact that Repair Symbols are interleaved with Source Symbols. In the case of short loss bursts, the Repair Symbols will follow directly after the lost Source Symbols. It can thus potentially provide a low-delay recovery. In this section, we perform experiments to see how the FEC Schemes behave with a shorter desynchronization.

5.7.1 Uniform losses

Figures 5.16 and 5.17 illustrate this property by showing the results of the experiments with a uniform loss model and desynchronization of two application messages instead of four. We compare the results with those obtained with UDP. With our rate of 30 application messages per second, a desynchronization of 2 messages corresponds to around 33 milliseconds. As we can see in Figures 5.16 and 5.17, this desynchronization is not sufficient for our Reed-Solomon FEC Scheme which needs to receive more packets than those provided by the desynchronization before finally receiving the Repair Symbols. The performances of Reed-Solomon in this case are not so far from UDP, with the FEC Scheme only recovering the packets that are lost at the end of a Source Block. On the other hand, the RLC FEC Scheme is nearly not affected by this change

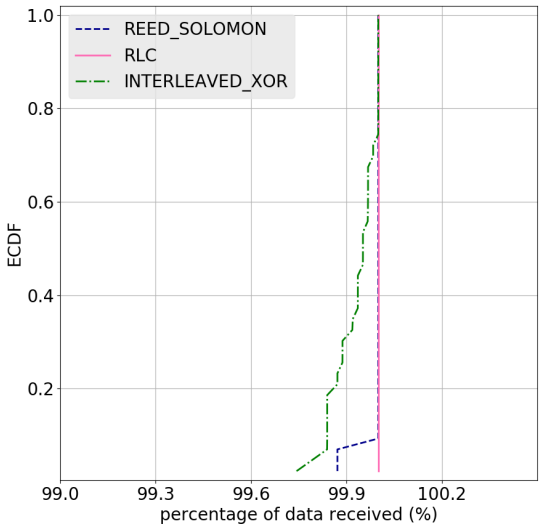


Figure 5.11: Convolutional RLC VS Reed-Solomon VS XOR: amount of data received with uniform losses

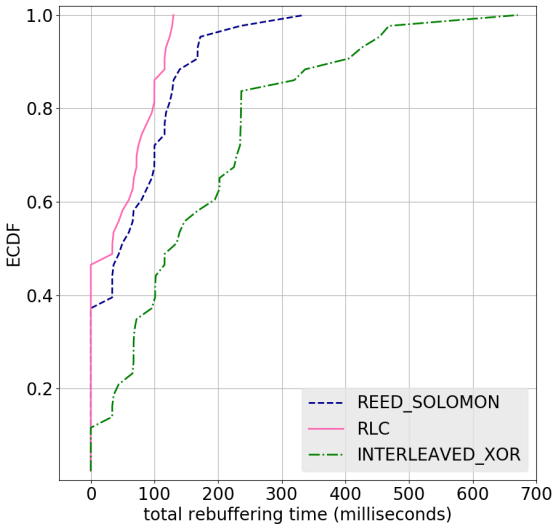


Figure 5.12: Convolutional RLC VS Reed-Solomon VS XOR: total rebuffering time with uniform losses

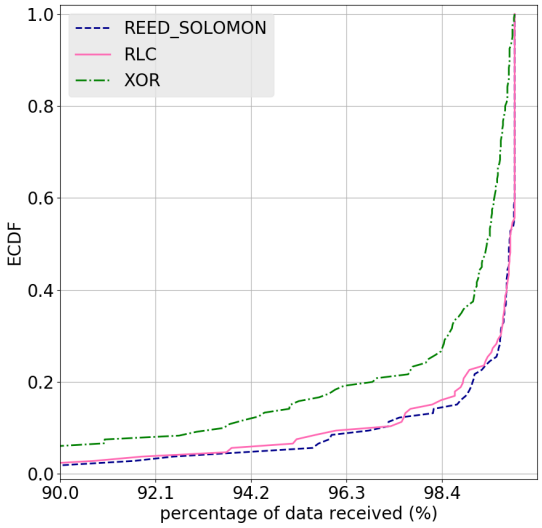


Figure 5.13: Convolutional RLC VS Reed-Solomon VS XOR: amount of data received with our Gilbert-Elliott model

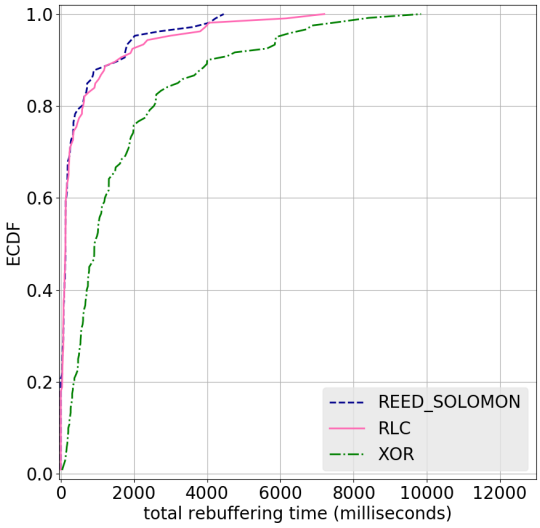


Figure 5.14: Convolutional RLC VS Reed-Solomon VS XOR: total rebuffering time with our Gilbert-Elliott model

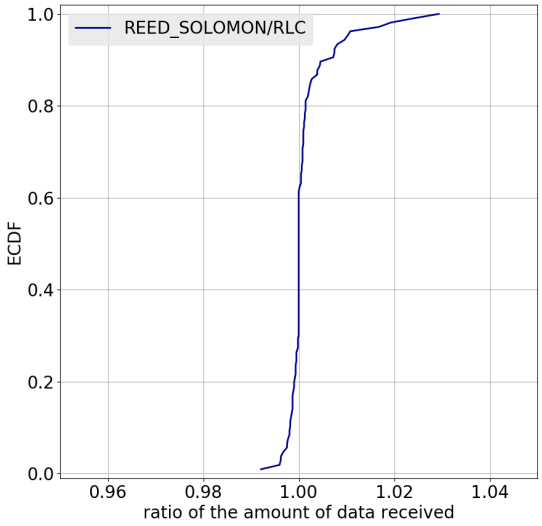


Figure 5.15: Reed-Solomon VS Convolutional RLC: ratio of amount of data received with our Gilbert-Elliott model

and will recover every lost packet in the vast majority of the cases, encountering only a small number of re-bufferings. The problem of the Reed-Solomon FEC Scheme could be mitigated by reducing drastically the block size while keeping the same code rate. However, doing so reduces the burst handling capabilities of the code. In the next section, we will see if the advantage of RLC is still present with a bursty loss model.

5.7.2 Gilbert-Elliott loss model

We now compare the two FEC Schemes with a desynchronization of 2 frames under our Gilbert-Elliott loss model. Figures 5.18 and 5.19 respectively show the ratio of the amount of data received between the two FEC Schemes and the ECDF of the total re-buffering time. Although RLC still seems to recover more data and encounter fewer re-bufferings, the difference is less clear than with the uniform loss model. Figure 5.18 shows that Reed-Solomon recovers more data in some cases. As for the results of section 5.6.2, these cases are where the r and h parameters are very low.

5.8 The potential benefits of Multipath with Forward Erasure Correction

When considering loss bursts, sending the data on multiple paths can help to recover more packets with a same level of redundancy compared to single-path transmission. An example can be shown in Figure 5.20. In a single-path configuration, a loss burst of 3 symbols will be recovered if the burst begins at R_1, R_2, R_3 or R_4 and will be lost in every other case (if the burst begins at any of the 8 other symbols). It will thus be recovered in 1/3 of the cases. With a multi-path configuration using a round-robin scheduler and considering that losses occurring on both paths at the same time are rare, a loss burst of 3 symbols on Path 1 will be recovered if the burst begins at P_3, R_1, P_7 or R_3 and will be lost if the burst begins at P_1 or P_5. It will thus be

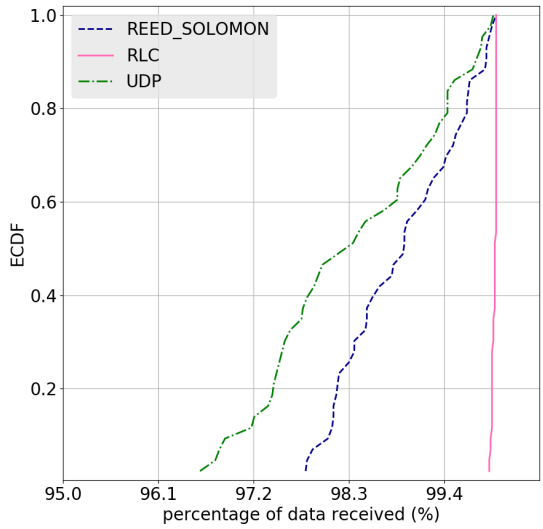


Figure 5.16: Convolutional RLC VS Reed-Solomon VS UDP: amount of data received with uniform losses and a desynchronization of 2 application messages

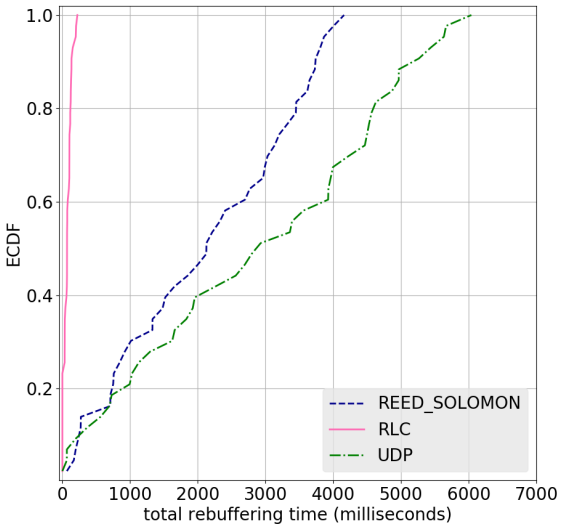


Figure 5.17: Convolutional RLC VS Reed-Solomon VS UDP: total rebuffering time with uniform losses and a desynchronization of 2 application messages

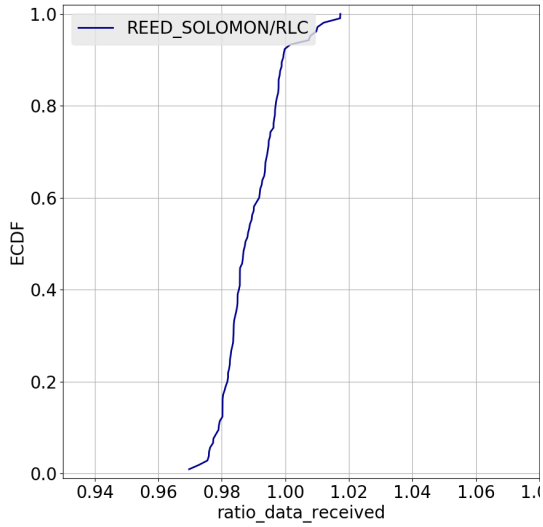


Figure 5.18: Convolutional RLC VS Reed-Solomon: ratio of the amount of data received with our Gilbert-Elliott model and a desynchronization of 2 application messages

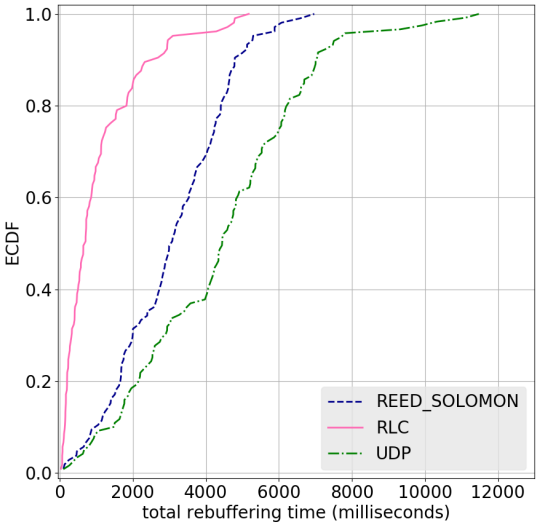


Figure 5.19: Convolutional RLC VS Reed-Solomon VS UDP: total rebuffering time with our Gilbert-Elliott model and a desynchronization of 2 application messages

recovered in 2/3 of the cases. This is because when a loss burst affects on one path the packets of one Source Block, some packets of the Source Block will still be received as they will be sent on the other path. Interleaving the packets will thus spread the losses on multiple Source Blocks that will be handled independently.

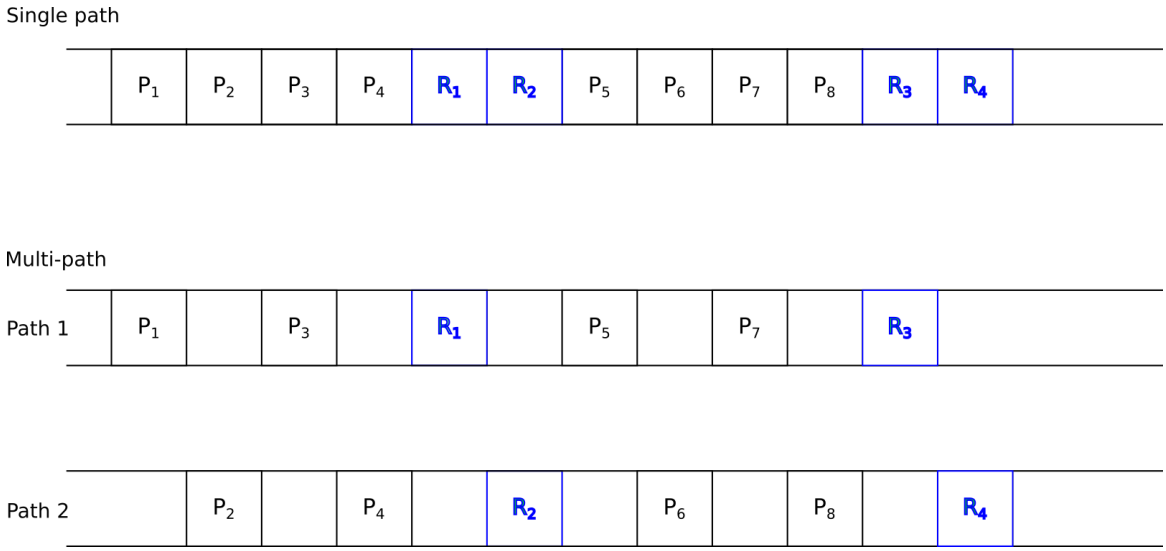


Figure 5.20: Bursts with single-path VS multipath with a round-robin scheduler

5.9 Multipath experiments with a block code

In order to see whether multipath could improve the efficiency of Forward Erasure Correction, experiments have been performed with a multipath configuration using the multipath implementation of `quic-go` [33] . The network configuration of the multipath experiments is shown in Figure 5.21. This configuration offers two network interfaces to the client and only one for the server. This results in two different possible paths, the path *Client-R1-R3-Server* that we call $path_1$ and the path *Client-R2-R3-Server* that we call $path_2$. Our single-path client always contacts the server using $path_1$.

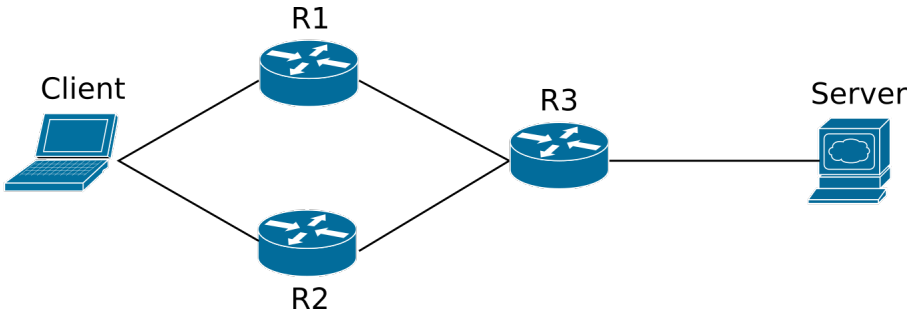


Figure 5.21: Multipath network configuration for our experiments

We perform experiments on homogeneous and heterogeneous paths, in order to have a good overview of the results in different configurations. In this section, all the experiments are performed with our simplified two-states Markov model (i.e. Gilbert-Elliott model with $h = 0$ and $k = 1$), to avoid having too heavy experiments coming from the potentially important number of parameters that vary. The delay and loss model are both applied on the links R1-R3 and R2-R3. It implies that the losses occurring on one link will be independent from the losses occurring on the other link. The parameters ranges used for the experimental design are the

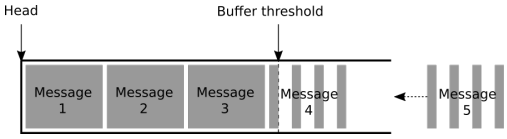


Figure 5.22: Message 4 is partially lost by the burst of 8 packets: the buffer is not blocking but the burst is spread on two messages.

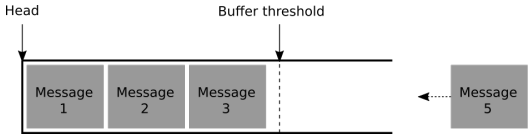


Figure 5.23: Message 4 is completely lost by the burst of 8 packets: the buffer is blocking but the burst is concentrated on one message.

same as those described in Table 5.1. For each scenario, we use a $(30, 20)$ Reed-Solomon block code protecting the QUIC packets.

5.9.1 Homogeneous paths

We first perform experiments on paths that have the same characteristics. The two paths have the same delay and parameter values for their loss model.

5.9.1.1 Round-Robin scheduler compared to single-path

Figures 5.24, 5.25 and 5.26 respectively show the ECDF of the amount of data received, the ratio of the amount of data received and the ECDF of the total re-buffering time for the multipath and single-path configurations. We can see in Figure 5.25 that, as expected, the total amount of received data is often higher in the multipath case. When looking at our results, we saw that for all the cases where single-path performed better with a ratio above 1.001, it was either with parameter configuration with a high value for the r parameter (≥ 0.3) or a low value for both the r and p parameters, making both approaches having quite similar results. However, we can see in Figure 5.26 that the total re-buffering time can be higher with multipath compared to single-path. By looking at our results, the multipath scenario has a smaller total rebuffering time when the mean loss burst is short and a higher total rebuffering time when the mean loss burst is long. This is the drawback of interleaving the packets on the paths. A burst of n packets on one particular path will be perceived as the loss of one packet over two during $2n$ packets. Figures 5.22 and 5.23 compare a burst of the size of an application message between multipath with a round-robin scheduler and single-path.

As we can see in Figure 5.22, the interleaving implies that the losses will be spread over more application messages compared to the single path case. If such a loss burst cannot be recovered, it will imply more corrupted application messages and thus more re-bufferings compared to the single-path scenario. However, with loss bursts that can be recovered, interleaving the packets on two paths can be beneficial for the re-bufferings. Indeed, while an entire application message is lost in the single-path case of Figure 5.23, only one packet over two are lost in the multipath case of Figure 5.22 if the other path is not in a loss burst as well. At the single-path receiver-side, the desynchronization buffer will be blocking. The receiver will thus have to wait for the next application message before delivering its data. In the multipath case, a part of the message is received and the buffer won't block as the desynchronization is respected.

We can thus encounter some cases where the amount of data received is higher with multipath (because there are more recoverable bursts thanks to the interleaving) but the total re-buffering time is higher too (because the non-recoverable bursts have been spread on more application messages).

The benefit of multipath is thus not completely clear in our use-case in terms of the total re-buffering time. However, the amount of data received is still an interesting metric as it can be determining for applications whose messages are not longer than one packet. In the upcoming sections, we focus on increasing the total amount of data received using multipath.

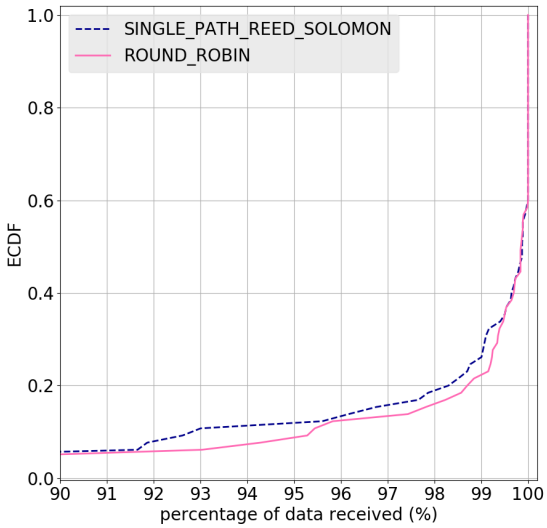


Figure 5.24: Single-path VS Round-Robin: amount of data received

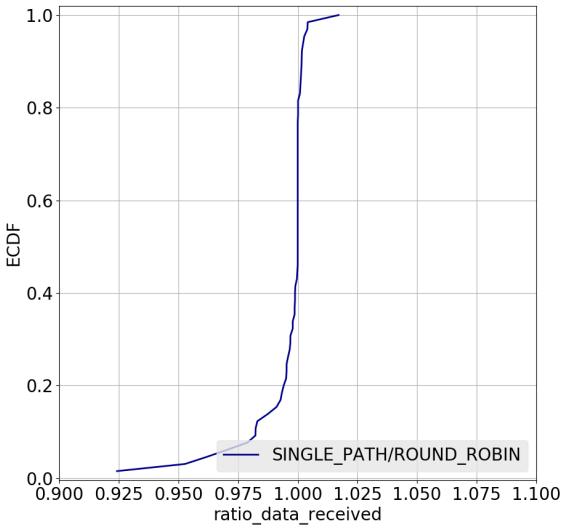


Figure 5.25: Single-path VS Round-Robin: ratio $\frac{\text{SinglePath}}{Round-Robin}$ of the amount of data received

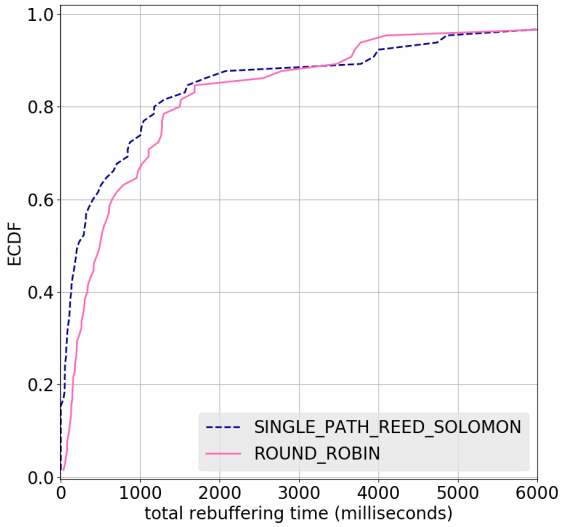


Figure 5.26: Single-path VS Round-Robin: total rebuffering time

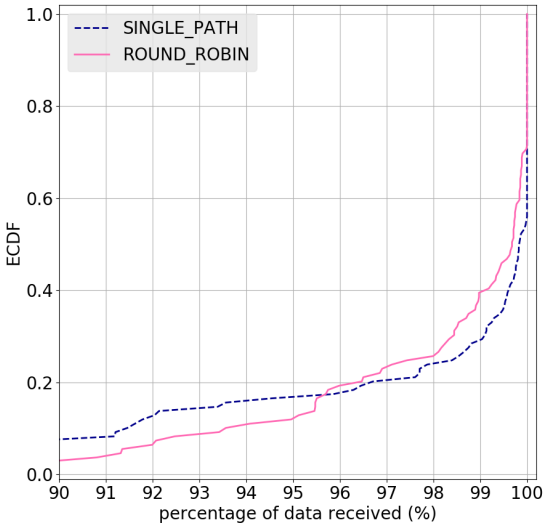


Figure 5.27: Single-path VS Round-Robin: amount of data received on paths with different losses

5.9.2 Heterogeneous paths: different losses, same delays

In this section, we consider paths with different loss parameters and identical delays. Figure 5.27 shows the amount of data received on configurations whose two paths have potentially different loss parameters values. We can see that the curves of the round-robin and the single-path cases cross at some point. By looking at our results, we saw that for the cases where the round-robin scheduler received more data, the p parameter of $path_1$ is greater than the p parameter of $path_2$ by 0.0018 and the r parameter is lower by 0.19 on average. This explains why the round-robin scheduler is better in these cases. When $path_2$ is better than $path_1$ in terms of losses, using both paths is better than only using the worse. We also have the same phenomenon for the cases where the single-path scenario received more data: in these cases, the p parameter of $path_1$ is lower by 0.0019 and r is greater by 0.15 on average compared to $path_2$, meaning that only using $path_1$ is better than using both paths.

5.9.3 The MAXRB scheduler

We can conclude from the previous section that a scheduler has an interest in being adaptive to the loss rate when the two paths have different loss characteristics. It will avoid using a bad path diminishing the quality of the transmission. However, we still would like to have the benefits of path interleaving with Forward Erasure Correction. We thus need to define a new type of scheduler whose purpose is to perform interleaving when possible, and use only one path when using both could be harmful. We define the HIGHRB scheduler as a scheduler that selects a path randomly, using the number of remaining bytes of the congestion window as weights of the random selection. The scheduler is described in Algorithm 1.

Algorithm 1 HIGHRB scheduler

Require: $\mathcal{P}$, the set of available paths
Require: $BytesInFlight(p)$, the number of sent, non-acknowledged bytes of p, $\forall p \in \mathcal{P}$
Require: $Cwin(p)$, the congestion window of p, $\forall p \in \mathcal{P}$
$RemainingBytes(p) \leftarrow Cwin(p) - BytesInFlight(p)$, $\forall p \in \mathcal{P}$
$Total \leftarrow \sum_{p \in \mathcal{P}} RemainingBytes(p)$
if $Total \neq 0$ **then**
    $w(p) \leftarrow \frac{RemainingBytes(p)}{Total}$, $\forall p \in \mathcal{P}$
else
    $w(p) \leftarrow \frac{1}{|\mathcal{P}|}$, $\forall p \in \mathcal{P}$
end if
select p^* randomly from $\mathcal{P}$ with weights w
return p^*

The purpose of this scheduler is to gather informations from a congestion control algorithm that decreases its congestion window when losses occur on a path. The congestion window of lossy paths will be smaller than the congestion window of other paths, implying a lower number of remaining bytes on average and a lower probability to select this path. Our implementation of HIGHRB in `quic-go` relies on the congestion window provided by the OLIA congestion control algorithm [53]. HIGHRB uses a weighted random selection instead of always selecting the path with the highest number of remaining bytes because we focus on the real-time use-case and real-time applications do not often use all the available bandwidth. Selecting always the path with the highest number of remaining bytes could thus lead to only use one path instead of doing interleaving if the sending rate of the application is not large enough. Finally, the random selection also allows to continue to regularly probe the non-preferred paths to check if it could become a interesting path to use.

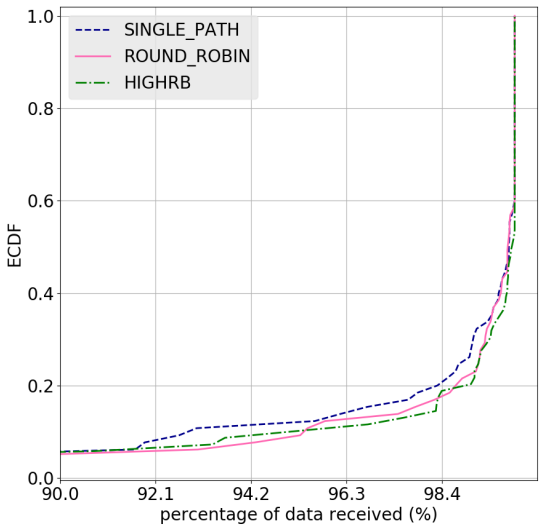


Figure 5.28: Single-path VS Round-Robin VS HIGHRB: amount of data received on homogeneous paths

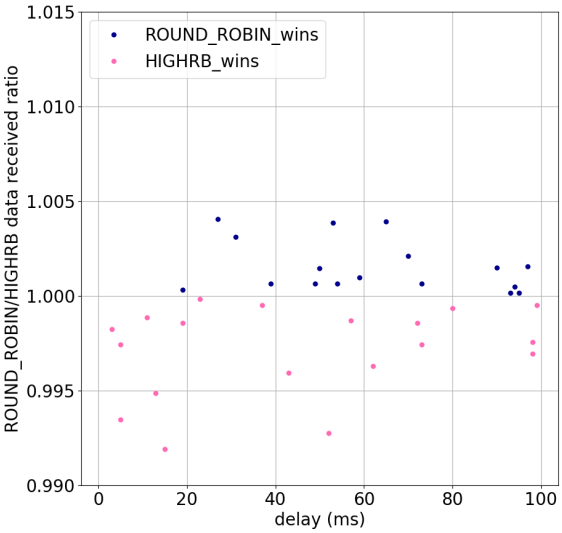


Figure 5.29: Round-Robin VS HIGHRB: amount of data received on homogeneous paths

We first take a look at the performances of HIGHRB when the paths are homogeneous, shown at Figure 5.28. We can see that HIGHRB seems to perform quite smiliarly than the round-robin scheduler. When looking more closely at our data and comparing HIGHRB and the round-robin scheduler, HIGHRB however received more data in every configuration with a low delay ($\leq 15ms$).

Figure 5.29 shows the ratio $\frac{\text{Round-Robin}}{\text{HIGHRB}}$ of the amount of data received in function of the delay from our experiments (we removed dots where the ratio was 1). We can see that for low delays, the round-robin scheduler never won a test. This is an advantage of using the remaining bytes of the congestion window as a metric to choose a path. As we send one application message every 33 milliseconds, a long loss burst that spans two application messages will be detected by the `quic-go` receiver before its end when the round-trip-time is short. It will thus lower the congestion window and the number of remaining bytes before the end of the burst. The scheduler will choose the other path on average to avoid to loose additional packets, while the round-robin scheduler will continue to use both paths.

We now take a look at the performances of HIGHRB when the paths can have different loss parameters. Figure 5.30 shows us the ECDF of the amount of data received compared to single-path and the round-robin scheduler. We can see that HIGHRB seems to outperform both other alternatives. We found however some cases where the single-path solution performed sensibly better than HIGHRB. These cases were cases for which the r parameter of $path_1$ was strongly higher than for $path_2$. In that case, although HIGHRB is adaptive, only using the first path gives better results.

5.9.4 Heterogeneous paths: different losses, different delays

The last case that we want to analyse is the case where both the delays and the loss parameters values can differ for the two paths. Figure 5.31 shows that HIGHRB seems less perturbed than the round-robin scheduler, but often performs badly compared to the single-path case. It is typically the case when either the $path_1$ has a r parameter strongly larger than the $path_2$ or when the difference of delays is high (> 70 milliseconds of difference). This is one of the advantages of

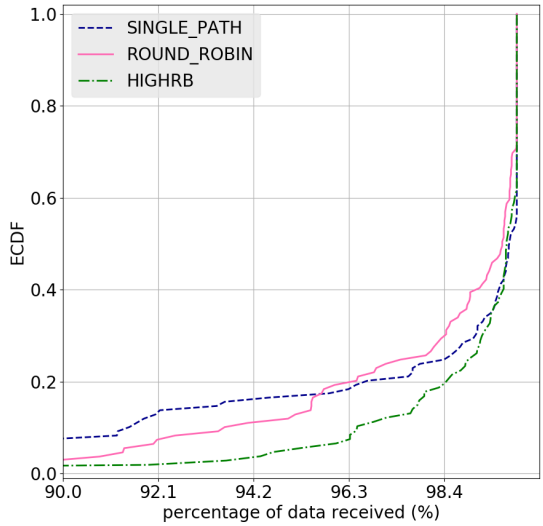


Figure 5.30: Single-path VS Round-Robin VS HIGHRB: amount of data received on paths with different losses

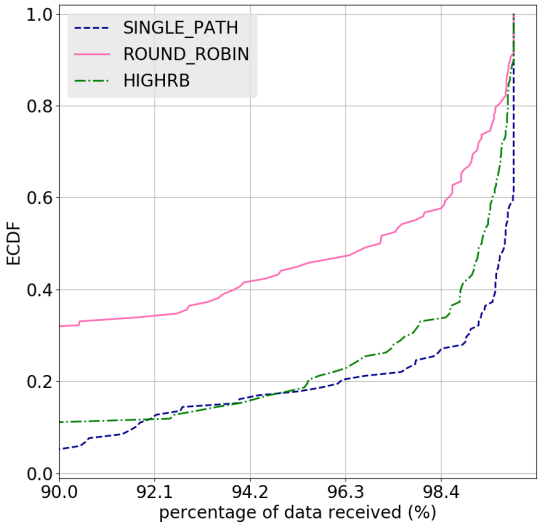


Figure 5.31: Single-path VS Round-Robin VS HIGHRB: amount of data received on paths with different losses and delays

single-path: by always selecting the same path, it avoids packets re-ordering when the delays of the two paths are strongly different. A large difference in the delays will imply an important packet re-ordering: the packets sent on one path will arrive later than the packets sent on the other path. In that case, the buffer on the receiver-side will have difficulties in maintaining a constant desynchronization between the two peers and the packets sent on the path having a high delay will arrive too late to be read. However, HIGHRB still performs well in the case where the r parameter is strongly larger on the $path_2$ than on the $path_1$, for lower delay differences (< 70 milliseconds). In this case, HIGHRB will mostly use the $path_2$ and no or little re-ordering will occur, while the single-path configuration will be forced to use a worse path. We can thus conclude that while HIGHRB is adaptive to lossy paths, it can be more beneficial to only use one path when the delays are too different.

Conclusion

In this thesis, we studied how to enlarge the possible use-cases of the QUIC protocol, by considering the real-time transfer use-case. For this purpose, we designed and implemented two extensions of the QUIC protocol. First, the Forward Erasure Correction extension allows to recover lost data packets without paying the price of a retransmission. We created a design to support several different FEC Schemes. We then implemented our design and added three different FEC Schemes to our implementation: two block FEC Schemes and one convolutional FEC Scheme. We also designed and implemented the Unreliable Stream extension, that allows to skip data whose expiration deadline is passed. We performed experiments in order to assess the performances of our extensions and to show that real-time communications have a benefit from using Forward Erasure Correction and an unreliable data transfer. Finally, we considered the benefits that could be brought by a multipath communication to a real-time use-case using Forward Erasure Correction. We designed and implemented a new multipath scheduler that interleaves the data on multiple paths when it could be beneficial in order to break long loss bursts. We performed comparisons to assess its performances and showed that although interleaving the data on multiple paths can have some drawbacks in terms of total re-buffering time, it can perform better than a single-path transmission concerning the total amount of data received.

Bibliography

[1] Jana Iyengar and Martin Thomson. Quic: A udp-based multiplexed and secure transport. Internet-Draft draft-ietf-quic-transport-11, IETF Secretariat, April 2018. http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-11.txt.

[2] Mike Bishop. Hypertext transfer protocol (http) over quic. Internet-Draft draft-ietf-quic-http-11, IETF Secretariat, April 2018. http://www.ietf.org/internet-drafts/draft-ietf-quic-http-11.txt.

[3] Christian Huitema, Melinda Shore, Allison Mankin, Sara Dickinson, and Jana Iyengar. Specification of dns over dedicated quic connections. Internet-Draft draft-huitema-quic-dnsoquic-03, IETF Secretariat, January 2018. http://www.ietf.org/internet-drafts/draft-huitema-quic-dnsoquic-03.txt.

[4] Domenico Ferrari. Client requirements for real-time communication services. 1990.

[5] Jon Postel. Rfc793: Transmission control protocol. 1981.

[6] Moufida Feknous, Thierry Houdoin, Bertrand Le Guyader, Joseph De Biasio, Annie Gravey, and Jose Alfonso Torrijos Gijón. Internet traffic analysis: A case study from two major european operators. In *Computers and Communication (ISCC), 2014 IEEE Symposium on*, pages 1–7. IEEE, 2014.

[7] Jerry Chu, Arvind Jain, Yuchung Cheng, and Sivasankar Radhakrishnan. Rfc7413: Tcp fast open. 2014.

[8] Randall Stewart. Rfc4960: Stream control transmission protocol. 2007.

[9] Jon Postel. Rfc768: User datagram protocol. Technical report, 1980.

[10] Van Jacobson, Ron Frederick, Steve Casner, and H Schulzrinne. Rfc3550: Rtp: A transport protocol for real-time applications. 2003.

[11] Alan Ford, Costin Raiciu, Mark Handley, and Olivier Bonaventure. Rfc6824: Tcp extensions for multipath operation with multiple addresses. Technical report, 2013.

[12] Benjamin Hesmans, Fabien Duchene, Christoph Paasch, Gregory Detal, and Olivier Bonaventure. Are tcp extensions middlebox-proof? In *Proceedings of the 2013 workshop on Hot topics in middleboxes and network function virtualization*, pages 37–42. ACM, 2013.

[13] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? designing and implementing a deployable multipath tcp. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 29–29. USENIX Association, 2012.

[14] Martin Thomson. Version-independent properties of quic. Internet-Draft draft-ietf-quic-invariants-01, IETF Secretariat, March 2018. http://www.ietf.org/internet-drafts/draft-ietf-quic-invariants-01.txt.

[15] Quic wire layout specification. https://docs.google.com/document/d/1WJvyZflAO2pq77yOLbp9NsGjC1CHetAXV8I0fQe-B_U/edit. Accessed: 2018-04-24.

[16] Arash Molavi Kakhki, Samuel Jero, David Choffnes, Cristina Nita-Rotaru, and Alan Mislove. Taking a long look at quic: an approach for rigorous evaluation of rapidly evolving transport protocols. In *Proceedings of the 2017 Internet Measurement Conference*, pages 290–303. ACM, 2017.

[17] Phillip Rogaway. Authenticated-encryption with associated-data. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 98–107. ACM, 2002.

[18] Quic crypto. https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g/edit. Accessed: 2018-06-04.

[19] Adam Bergkvist, Daniel C Burnett, Cullen Jennings, Anant Narayanan, and Bernard Aboba. Webrtc 1.0: Real-time communication between browsers. *Working draft, W3C*, 91, 2012.

[20] Justin Uberti. Webrtc forward error correction requirements. Internet-Draft draft-ietf-rtcweb-fec-08, IETF Secretariat, March 2018. http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-fec-08.txt.

[21] Vincent Roca, Mark Watson, and Ali C. Begen. Forward Error Correction (FEC) Framework. RFC 6363, October 2011.

[22] Amine Bouabdallah, Mathieu Cunche, Vincent Roca, Kazuhisa Matsuzono, and Jerome Lacan. Rfc6865: Simple reed-solomon forward error correction (fec) scheme for fecframe. 2013.

[23] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[24] Vincent Roca, Belkacem Teibi, Christophe Burdinat, Tuan Tran, and Cédric Thienot. Less latency and better protection with al-fec sliding window codes: A robust multimedia cbr broadcast case study. In *Wireless and Mobile Computing, Networking and Communications (WiMob),*, pages 1–8. IEEE, 2017.

[25] Vincent Roca and Ali Begen. Forward error correction (fec) framework extension to sliding window codes. Internet-Draft draft-ietf-tsvwg-fecframe-ext-01, IETF Secretariat, March 2018. http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-fecframe-ext-01.txt.

[26] Vincent Roca and Belkacem Teibi. Sliding window random linear code (rlc) forward erasure correction (fec) schemes for fecframe. Internet-Draft draft-ietf-tsvwg-rlc-fec-scheme-02, IETF Secretariat, March 2018. http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-rlc-fec-scheme-02.txt.

[27] Vincent Roca, Belkacem Teibi, Christophe Burdinat, Tuan Tran-Thai, and Cédric Thienot. Block or convolutional al-fec codes? a performance comparison for robust low-latency communications. 2017.

[28] Ryan Hamilton, Janardhan Iyengar, Ian Swett, and Alyssa Wilk. Quic: A udp-based secure and reliable transport for http/2. Internet-Draft draft-hamilton-early-deployment-quic-00, IETF Secretariat, July 2016. http://www.ietf.org/internet-drafts/draft-hamilton-early-deployment-quic-00.txt.

[29] Quic-fec. https://datatracker.ietf.org/meeting/99/materials/slides-99-nwcrg-08-swett-quic-fec-00. Accessed: 2018-06-02.

[30] Philipp Tiesel, Mirko Palmer, Balakrishnan Chandrasekaran, Anja Feldmann, and Joerg Ott. Considerations for unreliable streams in quic. Internet-Draft draft-tiesel-quic-unreliable-streams-00, IETF Secretariat, September 2017. http://www.ietf.org/internet-drafts/draft-tiesel-quic-unreliable-streams-00.txt.

[31] Philipp Tiesel, Mirko Palmer, Balakrishnan Chandrasekaran, Anja Feldmann, and Joerg Ott. Considerations for unreliable streams in quic. Internet-Draft draft-tiesel-quic-unreliable-streams-01, IETF Secretariat, October 2017. http://www.ietf.org/internet-drafts/draft-tiesel-quic-unreliable-streams-01.txt.

[32] Quentin De Coninck and Olivier Bonaventure. Multipath extension for quic. Internet-Draft draft-deconinck-quic-multipath-00, IETF Secretariat, March 2018. http://www.ietf.org/internet-drafts/draft-deconinck-quic-multipath-00.txt.

[33] Quentin De Coninck and Olivier Bonaventure. Multipath quic: Design and evaluation. In *13th International Conference on emerging Networking EXperiments and Technologies (CoNEXT 2017). http://multipath-quic.org*, 2017.

[34] Stephen K. Park and Keith W. Miller. Random number generators: good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, 1988.

[35] Vincent Roca and Belkacem Teibi. Sliding window random linear code (rlc) forward erasure correction (fec) schemes for fecframe. Internet-Draft draft-ietf-tsvwg-rlc-fec-scheme-05, IETF Secretariat, May 2018. http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-rlc-fec-scheme-05.txt.

[36] Brandon Heller Bob Lantz and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[37] Gaetano Carlucci, Luca De Cicco, and Saverio Mascolo. Http over udp: an experimental investigation of quic. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 609–614. ACM, 2015.

[38] Yong Cui, Lian Wang, Xin Wang, Hongyi Wang, and Yining Wang. Fmtcp: A fountain code-based multipath transmission control protocol. *IEEE/ACM Transactions on Networking (ToN)*, 23(2):465–478, 2015.

[39] Enhuan Dong, Mingwei Xu, Xiaoming Fu, and Yu Cao. Lamps: A loss aware scheduler for multipath tcp over highly lossy networks. In *Local Computer Networks (LCN), 2017 IEEE 42nd Conference on*, pages 1–9. IEEE, 2017.

[40] Simone Ferlin, Stepan Kucera, Holger Claussen, and Özgü Alay. Mptcp meets fec: Supporting latency-sensitive applications over heterogeneous networks. *IEEE/ACM Transactions on Networking*.

[41] Pierre Ugo Tournoux, Emmanuel Lochin, Jérôme Lacan, Amine Bouabdallah, and Vincent Roca. On-the-fly erasure coding for real-time video applications. *IEEE Transactions on Multimedia*, 13(4):797–812, 2011.

[42] Ahmed Badr, Ashish Khisti, Wai-Tian Tan, and John Apostolopoulos. Streaming codes for channels with burst and isolated erasures. In *INFOCOM, 2013 Proceedings IEEE*, pages 2850–2858. IEEE, 2013.

[43] Edwin O Elliott. Estimates of error rates for codes on burst-noise channels. *The Bell System Technical Journal*, 42(5):1977–1997, 1963.

[44] netem. https://wiki.linuxfoundation.org/networking/netem. Accessed: 2018-05-31.

[45] Péter Megyesi, Zsolt Krämer, and Sándor Molnár. How quick is quic? In *Communications (ICC), 2016 IEEE International Conference on*, pages 1–6. IEEE, 2016.

[46] Christoph Paasch, Ramin Khalili, and Olivier Bonaventure. On the benefits of applying experimental design to improve multipath tcp. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 393–398. ACM, 2013.

[47] Christoph Paasch, Simone Ferlin, Ozgu Alay, and Olivier Bonaventure. Experimental evaluation of multipath tcp schedulers. In *Proceedings of the 2014 ACM SIGCOMM workshop on Capacity sharing workshop*, pages 27–32. ACM, 2014.

[48] J Santiago, M Claeys-Bruno, and M Sergent. Construction of space-filling designs using wsp algorithm for high dimensional spaces. *Chemometrics and Intelligent Laboratory Systems*, 113:26–31, 2012.

[49] Bo Zhang, TS Eugene Ng, Animesh Nandi, Rudolf Riedi, Peter Druschel, and Guohui Wang. Measurement-based analysis, modeling, and synthesis of the internet delay space. 2006.

[50] Ffmpeg. https://ffmpeg.org. Accessed: 2018-06-05.

[51] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[52] Colin Perkins. Rtp and the datagram congestion control protocol (dccp). 2010.

[53] Ramin Khalili, Nicolas Gast, Miroslav Popovic, Utkarsh Upadhyay, and Jean-Yves Le Boudec. Mptcp is not pareto-optimal: performance issues and a possible solution. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 1–12. ACM, 2012.

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