FlEC: Enhancing QUIC with application-tailored reliability mechanisms

François Michel*, Alejandro Cohen†, Derya Malak‡, Quentin De Coninck*, Muriel Médard§, Olivier Bonaventure*

*UCLouvain, Belgium, {francois.michel,quentin.deconinck,olivier.bonaventure}@uclouvain.be
†Technion–Israel Institute of Technology, Haifa, Israel, alecohen@technion.ac.il
‡Ressenlaer Polytechnic Institute, New York, USA, malakd@rpi.edu
§Massachusetts Institute of Technology, Cambridge, USA, medard@mit.edu

***Abstract*—Packet losses are common events in today's networks. They usually result in longer delivery times for application data since retransmissions are the *de facto* technique to recover from such losses. Retransmissions is a good strategy for many applications but it may lead to poor performance with latency-sensitive applications compared to network coding. Although different types of network coding techniques have been proposed to reduce the impact of losses by transmitting redundant information, they are not widely used. Some niche applications include their own variant of Forward Erasure Correction (FEC) techniques, but there is no generic protocol that enables many applications to easily use them. We close this gap by designing, implementing and evaluating a new Flexible Erasure Correction (FlEC) framework inside the newly standardized QUIC protocol. With FlEC, an application can easily select the reliability mechanism that meets its requirements, from pure retransmissions to various forms of FEC. We consider three different use cases: (i) bulk data transfer, (ii) file transfers with restricted buffers and (iii) delay-constrained messages. We demonstrate that modern transport protocols such as QUIC may benefit from application knowledge by leveraging this knowledge in FlEC to provide better loss recovery and stream scheduling. Our evaluation over a wide range of scenarios shows that the FlEC framework outperforms the standard QUIC reliability mechanisms from a latency viewpoint.**

***Index Terms*—Networking, transport protocol, Forward Erasure Correction, RLNC, QUIC, protocol plugins.**

I. Introduction

The transport layer is one of the key layers of the protocol stack. It ensures the end-to-end delivery of application data through the unreliable network layer. There are two main families of transport protocols: the unreliable datagram protocols like UDP, DCCP [1], RTP [2] or QUIC datagrams [3], and the reliable ones such as TCP [4], SCTP [5] or QUIC [6]. During the last years, QUIC has attracted a growing interest thanks to its design. The QUIC specification was finalised in May 2021 [7] and as of November 2021, it is already supported by more than 30M domains on the Internet [8]. QUIC structures its control information and data into frames and supports stream multiplexing. Finally, QUIC includes the authentication and encryption functions of Transport Layer Security (TLS) [9]. By using the latter to encrypt and authenticate all data and most of the headers, QUIC prevents interference from middleboxes. Coupled with the availability of more than two dozens open-source implementations [10], QUIC has become a very interesting platform for transport layer research [11]–[15]. While QUIC leverages the loss recovery and congestion control techniques that are part of modern TCP implementations, other loss recovery mechanisms using FEC have recently been considered to recover earlier from packet losses [16]–[18].

Packet losses, either caused by congestion or transmission errors are frequent in today's networks and seriously considered for the design of modern transport protocols [6]. The first transport protocols relied on simple Automatic Repeat reQuest (ARQ) mechanisms [4], [19] to recover from losses. Over the years, a range of heuristics have been proposed. For TCP, this includes the fast retransmit heuristic [20], selective acknowledgements [21], the Eifel algorithm [22], recent acknowledgments [23], tail loss recovery techniques [24], [25], and others. Other reliable transport protocols have also benefited from this effort. SCTP and QUIC include many of the optimizations added to TCP over the years [6], [26].

While retransmissions remain the prevalent technique to recover from packet losses, coding techniques have been proposed in specific scenarios such as ATM networks [27], audio/video traffic [28] or multicast services [29], [30] where the cost of retransmissions grows with the group size. Some of these approaches are supported by RTP extensions [31], [32]. Several of these approaches have been applied to TCP [33], [34], usually by using a coding sublayer below TCP and hiding the coding functions from the transport layer [35]–[37].

Despite these efforts, many Internet applications still select either an unreliable transport (such as UDP) or TCP that forces in-order delivery and suffers from head-of-line blocking [6]. If an application developer needs another reliability model, she needs to implement the logic directly inside the application.

In this article we propose to revisit the reliability mechanisms in the transport layer. Our main contribution is that we enable applications to finely tune the reliability mechanism of the transport protocol to closely fit their needs. We implement our solution using QUIC and protocol plugins [12], but our ideas are generic and can be applied to other protocols as well. We evaluate the flexibility of our techniques by considering a range of applications and show that our application-tailored reliability mechanism outperforms a one-size-fits-all solution.

This paper is organised as follows. We first discuss the current reliability mechanisms (Section II) of transport protocols and see how flexibility is currently provided by existing solutions (Section III). We then propose Flexible Erasure Correction (*FlEC*), a novel reliability mechanism that can be easily redefined on a per-application basis (Section IV) to adapt the reliability mechanism to the application needs. We implement *FlEC* inside QUIC (Section V) and demonstrate the benefits of the approach by studying three different use-cases (Sections VI-VIII) with competing needs that can all improve their quality of experience using *FlEC*.

II. Related work

The QUIC protocol currently provides a reliable bytestream abstraction using *streams*. Application data transiting through QUIC streams is carried inside `STREAM` frames. Upon detection of a loss, the application data carried by the lost `STREAM` frames are retransmitted in new `STREAM` frames sent in new QUIC packets. QUIC [15] uses two thresholds to detect losses: a *packet-based* threshold and a *time-based* threshold. The packet-based threshold marks a packet as lost after packets with a sufficiently higher packet number have been received, indicating that pure re-ordering is unlikely. The specification recommends that an unacknowledged packet with number x is marked as lost when the acknowledgement of a packet with a number larger or equal to $x + 3$ has been received. This is similar to TCP's fast retransmit heuristic. The time-based threshold marks an unacknowledged packet as lost after a sufficient amount of time when a packet sent later gets acknowledged. The specification recommends that an unacknowledged packet sent at time t is marked as lost after $t+\frac{9}{8}*RTT$ if a packet sent later has already been acked. These two thresholds are only reached after at least one round-trip time, resulting in a late retransmission for delay-constrained applications. Such applications would benefit from a reliability mechanism that corrects packet losses *a-priori*.

There are ongoing discussions within the IETF [16] and the research community [12], [14], [17] to add Forward Erasure Correction (FEC) capabilities to QUIC. This mechanism consists in sending redundant information (Repair Symbols) before packets (Source Symbols) are detected as lost. It is especially useful for applications that cannot afford to wait for a retransmission, either due to strong delay requirements or connections suffering from long delays. Google experimented with a naive XOR-based FEC solution in early versions of QUIC [38]. The IRTF Network Coding research group explored alternative solutions [16]. Relying on a XOR code [38] does not enable sending several repair symbols to protect a window of packets, preventing the solution from recovering loss bursts. The standardisation work [16] does not provide any performance evaluation nor technique to schedule source and repair symbols. QUIC-FEC [17] proposes several redundancy frameworks and codes for the QUIC protocol. In this previous work, we studied file transfers with different codes such as XOR, Reed-Solomon and Random Linear Codes (RLC). We showed that FEC with QUIC can be beneficial for small file transfers but is harmful for longer bulk transfers compared to Selective-Repeat ARQ (SR-ARQ) mechanisms. One of the main limitations of QUIC-FEC [17] is that the code rate is fixed during the connection, leading to the sending of unnecessary coded packets. Pluginized QUIC (PQUIC) [12] proposes a FEC plugin equivalent to the RLC part of QUIC-FEC. Finally, rQUIC [18] presents an adaptive algorithm to regulate the code rate in function of the channel loss rate for QUIC communications. rQUIC has two main differences with our work. First, rQUIC assumes that isolated losses are not due to congestion. When it recovers isolated lost packets, rQUIC hides their loss signal to QUIC's congestion control to benefit from a larger bitrate. We follow the IRTF NWCRG recommendations [39]: we never hide any signal to the congestion control. If isolated losses are not due to congestion, then a specific congestion control ignoring isolated losses can be used instead of hiding the loss signal. The second difference is that we adapt the redundancy both to the channel conditions and the application requirements. Our solution will neither send the same amount of redundancy nor the same pattern of Repair Symbols for a bulk download and for a real-time video conferencing application.

Network coding has been considered for other transport protocols [40]–[42]. TCP/NC [33] adds network coding to TCP connections by applying a coding layer beneath the transport layer. It improves the TCP throughput by recovering from packets losses that block the TCP window. CTCP [34] pushes the idea further and proposes a revised congestion control algorithm for wireless communications. Tetrys [43] proposes a coding mechanism focused on real-time video applications and develops heuristics to adjust the coding rate to the sender's behaviour. RFC5109 [40] defines a standard RTP packet format to allow the use of FEC for RTP applications. An IETF draft [41] presents guidelines and requirements for the use of FEC for protecting video and audio streams in WebRTC. Minion [44] also uses coding to support unreliable data transfer above TCP. Existing work propose multi-path solutions to handle links with poor delay and fluctuating bandwidth [45]–[47] and use FEC to reduce head-of-line blocking. Unfortunately, none of the current solutions adapts the reliability mechanism to different classes of applications.

III. Tunable reliability mechanisms

Sending Repair Symbols for delay-sensitive applications is done at the cost of bandwidth when there is no loss to recover. This is why Repair Symbols should be sent carefully to avoid consuming bandwidth with no or low additional benefit.

Some adaptive FEC mechanisms have been proposed to adjust the redundancy overhead to the measured loss rate. Both CTCP [34] and TCP/NC [33] adjust the level of redundancy according to the measured loss rate. rQUIC [18] proposes a similar idea. While these approaches can show significant benefits compared to classical retransmission mechanisms, they still increase the overhead compared to the more efficient selective-repeat mechanisms in bulk download use-cases. By using a causal scheduling algorithm, we allow our solution

to react to the current channel condition and adopt similar behaviours to SR-ARQ when it is needed by the use-case. On the other-side, real-time applications cannot fully leverage the benefits of such transport-layer coding mechanisms because there is no way for an application to precisely express its requirements. The result is that such applications typically implement their own coding-enabled protocol [31], [48].

We reconcile strong application delay requirements and regular transport protocols by providing them with a tunable reliability mechanism that applications can adapt to their needs with small to no effort. We use QUIC to demonstrate our ideas, but they could be applicable to other transport protocols as well. QUIC stacks are mostly implemented as libraries that can be used by a wide range of applications. While QUIC can easily be tuned on the server-side to better fit the application requirements, obtaining such a flexibility on the end-user devices is more complicated, as the application wants to tune the underlying stack to meet its requirements. In the TCP/IP stack, this tuning is mainly done by using socket options or system-wide parameters. Socket Intents [49] and the ongoing work [50] within the IETF TAPS working group show that there is an interest to insert some knowledge from the application to the transport protocol. The QUIC specification [51] does not currently define a specific API between the application and the transport protocol but specifies a set of actions that could be performed by the application on the streams (e.g. reading and writing data on streams) and on the connection itself (e.g. switching on/off 0-RTT connection establishment or terminating the connection). The QUIC specification allows the application to pass information about the relative priority of the streams. However, it is unclear how, e.g., an application could express timeliness constraints.

On the other side, the current FEC specification for QUIC [16] does not guide the application to choose a code rate nor which parts of the application data should be FEC-protected and when coded symbols should be sent. Furthermore, different applications may require different strategies to send redundancy. In a video-conferencing application, Repair Symbols could protect a whole video frame. An IoT application [52] with limited buffers may want to protect the data incrementally to ensure a fast in-order delivery despite losses.

Contributions: Previously [53], we provided a joint coding scheme and algorithm in which one can theoretically manage the delay-rate tradeoff to get the required QoS. However, we did not cope with the complex and various requirements of real applications. We advocate that sending redundancy packets in transport protocols **should be done in adequacy with the application needs in order to provide satisfactory results.** In previous solutions, the FEC mechanism does not track both the channel condition and application requirements throughout the data transfer. In this work, we consider both the application's requirements and the network conditions to schedule the redundancy more efficiently. The contribution of this article is thus a complete redefinition of the reliability mechanism of the QUIC protocol by making it general and flexible. We introduce a general loss recovery framework able to implement both a classical SR-ARQ mechanism and FEC. We leverage the idea of protocol plugins [12] and implement our reliability mechanism as a framework exposing two anchors points to applications. Applications can redefine these anchors according to their delay-sensitivity and traffic pattern. We explore three different use-cases and show that adapting the reliability mechanism to the use-case can drastically improve the quality of the transmission. The first use-case is the bulk download scenario, discussed in Section VI. The second use-case discussed in Section VII is a scenario where the peer's receive window is small, resulting in the sender being regularly blocked by the flow control during loss events. The third use-case discussed in Section VIII is a scenario where the application sends messages that must arrive before a specific deadline. The three use-cases are described below.

A. Bulk file transfer

Bulk file transfer is the simplest use-case we consider. It consists in the download of a single file under the assumption that the receive buffer is large compared to the bandwidth-delay product of the connection. This is the classical use case for many transport protocols. Current open-source QUIC implementations use default receive window sizes that support such a use-case. The receive window starts at 2 MBytes for locally-initiated streams in `picoquic` [54]. The Chromium browser's implementation [55] starts with an initial receive window of 6MB per stream and 16MB for the whole connection. The metric that we minimize here is the total time to download the whole file. This includes REST API messages that often need to be completely transferred in order to be processed correctly by the application. As already pointed out [6], [17], [56], a packet loss during the last round-trip-time can have a high relative impact on the download completion time. The latter may indeed be doubled for small files due to the loss of a single packet. Protecting these tail packets can drastically improve the total transfer time at a cost significantly smaller than the cost of simply duplicating all these packets. On the other hand, protecting other packets than the tail ones with FEC can be harmful for the download completion time. The packet losses in the middle of the download can be recovered without FEC before any quiescence period provided that the receiver uses sufficiently large receive buffers.

B. Buffer-limited file transfers

In numerous network configurations, the available memory on the end devices is a limiting factor. It is common to see delays longer than 500 milliseconds in satellite communications, while their bandwidth is in the order of several dozens of Megabits per second [57], [58]. Furthermore, with the arrival of 5G, some devices will have access to bandwidth up to 10Gbps [59], [60]. While the edge latency of 5G infrastructures is intended to be in the order of a few milliseconds [61], the network towards the other host during an end-to-end transport connection may be significantly higher, partially due to the large buffers on the routers and the buffer-filling nature of currently deployed congestion control mechanisms. Packet

losses occurring on those high Bandwidth-Delay Product (BDP) network configurations imply a significant memory pressure on reliable transport protocols running on the end devices. To ensure an in-order delivery, the transport protocol running on these devices needs to keep the data received out-of-order during at least one round-trip-time, requiring receive buffer sizes to grow to dozens of megabytes for each connection. At the same time, QUIC is also considered for securing connections on IoT devices [52], [62]. Those embedded devices cannot dedicate large buffers for their network connections. Receive buffers that cannot bear the bandwidth-delay product of the network they are attached to are unable to fully utilize its capacity, even without losses. This typically occurs when the receive window is smaller than the sender's congestion window. Measurements show that TCP receivers frequently suffer from such limitations [6]. The problem gets even worse in case of packet losses as they prevent the receiver to deliver the data received out-of-order to the application. Those data will remain in memory, reducing the amount of new data that can be sent until the lost data is correctly retransmitted and delivered to the application. Sacrificing a few bytes of the receiver memory in order to handle repair symbols and protect the receive window from being blocked upon packet losses can drastically improve the transfer time, even in a file transfer use-case. In such cases, FEC can be sent periodically along with non-coded packets during the download and not just at the end of the transfer.

C. Delay-constrained messaging

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

FEC can significantly improve the quality of such transfers by recovering from packet losses without retransmissions, at the expense of using more bandwidth. Researchers have already applied FEC to video applications [42], [63]. Some [42] take a redundancy rate as input and allocates the Repair Symbols given the importance of the video frame. Others [63] propose a congestion control scheme that reduces the impact of isolated losses on the sending rate. They then use this congestion control to gather knowledge from the transport layer to the application in order to adapt the transmission to the current congestion. We propose the reverse idea: the application transfers its knowledge directly in the transport protocol to automatically adjust its stream scheduler and redundancy rate given the application's requirements.

IV. FlEC

In this section, we present the Flexible Erasure Correction (*FlEC*) framework. *FlEC* starts from a previous theoretical work, AC-RLNC [53]. This previous work proposes a decision mechanism to schedule repair symbols depending on the network conditions and the feedback received from the receiver. In this approach, repair symbols are sent in reaction to two

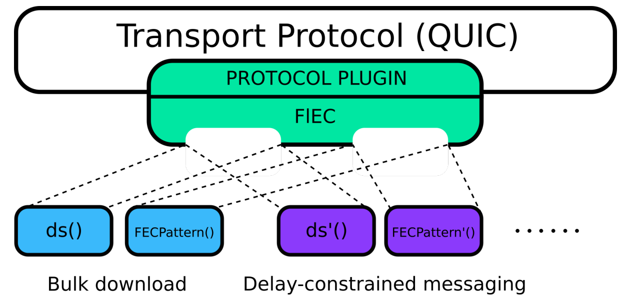

Figure 1. Design of the solution: a general framework with two pluggable anchors to redefine the reliability mechanism given the use-case.

thresholds: the first is triggered as a function of the number of missing degrees of freedom by the receiver, and the second threshold sends repair symbols *once every RTT*. The original goal of the proposed algorithm [53] is to trade bandwidth for minimizing the in-order delivery delay of data packets.

We start from this idea of tracking the sent, seen and received degrees of freedom as a first step to propose a redundancy scheduler for the transport layer. However, while this first idea provides a general behaviour, this may be insufficient for real applications with tight constraints that cannot be expressed with AC-RLNC's parameters. For example, a video-conferencing application may prefer to maximize bandwidth over low-delay links and therefore rely on retransmissions only, while FEC is needed over high-delay links as such retransmissions cannot meet the application's delay constraints. Instead of proposing configurable constant thresholds to tune the algorithm, we make it dynamic by proposing two redefinable functions: $ds()$ (for "*delay-sensitivity*") and $FECPattern()$. These two functions can be completely redefined to instantiate a reliability mechanism closely corresponding to the use-case. This allows having completely different FEC behaviours for use-cases with distinct needs such as HTTP versus video-conferencing. The $ds()$ threshold represents the sensitivity of the application to the in-order delivery delay of the data sent. In AC-RLNC, the FEC scheduler sends redundancy once per RTT. In *FlEC*, the $FECPattern()$ dynamic function allows triggering the sending of FEC at specific moments of the transfer depending on the use-case. Sending FEC for every RTT may deteriorate the application performance, especially when the delay is low enough to rely on retransmissions only. Having a dynamic $FECPattern()$ function avoids this problem. For instance, in a bulk download scenario, it can trigger FEC at the end of the download only and rely on retransmissions otherwise. For video transfer, it can trigger FEC after each video frame is sent. Figure 1 illustrates the idea of *FlEC*. The regular QUIC reliability mechanism is based on SR-ARQ. In *FlEC*, the SR-ARQ mechanism is a particular case among many other possibilities. Algorithm 1 shows our generic framework and Table I defines the variables used by our algorithms. We implement *FlEC* using PQUIC [12] and define $FECPattern()$ and $ds()$ as protocol operations. However, the same principles can be applied without PQUIC with the application redefining the

$\hat{l}$	the estimated uniform loss rate
$\hat{r}$	the estimated receive rate
$\hat{G_p}$, (resp. $\hat{G_r}$)	the estimated transition probability from the GOOD to the BAD (resp. BAD to GOOD) state of a Gilbert loss model [64]
md	missing degrees of freedom
ad	added degrees of freedom
$ds()$	customizable threshold eliciting Repair Symbols given the application's delay sensitivity
$FECPattern()$	customizable condition to send FEC using the application's traffic pattern

Table I
DEFINITION OF THE DIFFERENT SYMBOLS.

operations natively thanks to the user-space nature of QUIC.

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

Require: $\hat{l}$
Require: $feedback$, the most recent feedback received from the peer
Require: W, the current coding window
1: $\hat{r} \leftarrow 1 - \hat{l}$
2: $ad \leftarrow computeAd(W)$
3: $md \leftarrow computeMd(W)$
4: **if** $feedback = \varnothing$ **then**
5: **if** FECPattern() **then**
6: **return** $NewRepairSymbol$
7: **else**
8: **return** $NewData$
9: **end if**
10: **else**
11: updateLossEstimations(feedback)
12: **if** FECPattern() **then**
13: **return** $NewRepairSymbol$
14: **else if** $\hat{r} - \frac{md}{ad} < ds()$ **then**
15: **return** $NewRepairSymbol$
16: **else**
17: **return** $NewData$
18: **end if**
19: **end if**

The $computeMd$ function computes the number of missing degrees (md) of freedom (i.e. missing source symbols) in the current coding window. The $computeAd$ function computes the number of added degrees (ad) of freedom (i.e. repair symbols) that protect at least one packet in the current coding window. Compared to AC-RLNC, we only consider in-flight repair symbols in ad to support retransmissions when repair symbols are lost. The higher the value returned by $ds()$, the more likely it is to send repair symbols prior to the detection of a lost source symbol and the more robust is the delay between the sending of the source symbols and their arrival at the receiver. The extra cost is the bandwidth utilization. Sending repair symbols *a priori* occupies slots in the congestion window and is likely to increase the delay between the generation of data in the application and its actual transmission. Setting $ds()$ to $-\hat{l}$ triggers the transmission of repair symbols only in reaction to a newly lost source symbol, implementing thus a behaviour similar to regular

Use-case	$ds()$	$FECPattern()$
Bulk transfer (SR-ARQ)	$-\hat{l}$	$false$
AC-RLNC [53]	$c \cdot \hat{l}$	$true$ every RTT
Bulk transfer	$-\hat{l}$	$allStreamSent()$
Buffer-limited bulk	$c \cdot \hat{l}$	Algorithm 2
Messaging	$-\hat{l}$	Algorithm 4

Table II
DEFINITION OF $ds()$ AND $FECPattern()$ FOR THE CONSIDERED USE-CASES.

QUIC retransmissions. In this work, retransmissions are done using repair symbols to illustrate that the approach is generic. However, regular uncoded retransmissions can be used for better performance without loss of generality. $FECPattern()$ allows regulating the transmission of *a priori* repair symbols regardless of the channel state, in contrast with AC-RLNC [53] where this threshold is triggered once per RTT.

Table II describes how $ds()$ and $FECPattern()$ can be redefined to represent reliability mechanisms that fit the studied use-cases. The first row of the table shows how to implement the classical Selective-Repeat ARQ mechanism used by default in QUIC. The second one implements the behaviour of AC-RLNC [53]. $FECPattern$ is triggered once every RTT according to the EW parameter of AC-RLNC. The third one is tailored for the bulk use-case: $ds()$ is set to send Repair Symbols only when there are missing symbols at the receiver and $FECPattern()$ sends Repair Symbols when there is no more data to send. The two other rows are explained in details in the next sections. In this Table, c is a non-negative user-defined constant. The higher c is, the more sensitive we are to a variance in the loss rate.

A. Comparing FlEC and previous work

The origin of *FlEC* comes from the shortcomings of AC-RLNC [53] and QUIC-FEC [17]. As said earlier in this section, *FlEC* shares with AC-RLNC the idea of tracking the state of the communication in terms of received, seen and lost symbols. However, it adds the tight and diverse application requirements to the loop in order to adopt a correct behaviour for use-cases where FEC can be beneficial. It also adds all the transport-layer considerations such as staying fair to the congestion control of the protocol upon loss recovery.

FlEC also builds upon QUIC-FEC as it integrates similar transport layer considerations. For instance, *FlEC* uses as similar wire format as well as the concept of `RECOVERED` frame in order to differentiate packet acknowledgements from symbols recoveries. However, QUIC-FEC was designed without any care of the application traffic pattern or channel condition: the packet redundancy was not adaptive at all.

V. IMPLEMENTATION

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

We implement our *FlEC* framework inside PQUIC [12]. We implement the behaviours of the three use-cases discussed in this article by redefining $FECPattern()$ and $ds()$ to support the adequate reliability mechanism for each of them. Similarly to previous works [17], [53], we rely on random linear codes for the encoding and decoding of the symbols. This choice is made out of implementation convenience although other error correcting codes can be used as encoding/decoding tools of our work with only little adaptation. We advocate that even simpler codes such as Reed-Solomon can provide benefits for the considered use-cases although the benefit may be lower (e.g. such simple block codes cannot mix the repair symbols of different generations conversely to random linear codes).

We re-implemented the FEC plugin originally proposed in PQUIC [12] to match the latest design of the FEC extension for QUIC [16]. We enhanced the $GF(2^8)$ RLC implementation to use dedicated CPU instructions and adding an online system solver for faster symbols recovery. Most of the *FlEC* protocol operations consist in monitoring the current packet loop and providing a shim layer between the PQUIC design and the *FlEC* symbols scheduling algorithm. While we propose the *FlEC* framework as a protocol plugin, it can also be implemented natively and provided by default with the protocol implementation. The application can also provide its native implementation for the $ds()$ and $FECPattern()$ operations. The whole *FlEC* framework implementation takes 8200 lines of code. It adds a complete FEC extension to the QUIC protocol with the RLC error correcting code using PQUIC protocol plugins. This code is generic and does not have to be redefined by any application. The codes needed to define $ds()$ and $FECPattern()$ for the bulk and buffer-limited use-case have been written with respectively 57 and 97 lines of C code while the code for the messaging use-case takes 335 lines of C code. These two small functions are the parts that can be redefined by the application to stick to their use-case. Applications can also use our implementations for the three use-cases explored in this paper.

VI. Bulk file transfers

We here present the implementation and evaluation of the reliability mechanism proposed for bulk file transfers. The metric to minimize is the total download completion time. Sending unneeded repair symbols reduce the goodput and increase the download completion time. The expected behaviour is therefore similar to SR-ARQ with tail loss protection. The Repair Symbols are always sent within what is allowed by the congestion window, meaning that *FlEC* does not induce any additional link pressure.

A. Bulk reliability mechanism

For a file transfer, we set the delay-sensitivity threshold to be equal to $-\hat{l}$.

$$\hat{r} - md/ad < -\hat{l} \rightarrow sendRepairPacket() \quad (1)$$

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

$$md/ad > 1 \rightarrow sendRepairPacket() \quad (2)$$

so that we send Repair Symbols only when a packet is detected as lost and it has not been protected yet. The transmission of a Repair Symbol triggered by this threshold increases ad by 1 until ad becomes equal to md. Using the threshold defined in Equation 1 ensures a reliable delivery of the data but does not improve the download completion time in the case of tail losses. md only increases after a packet is marked as lost by the QUIC loss detection mechanism. The $FECPattern()$ operation controls the *a priori* transmission of Repair Symbols. In contrast with the previous solution [53], we redefine $FECPattern()$ and set it to $true$ only when all the application data has been sent instead of setting it to $true$ once per RTT. This implies that only the last flight of packets will be protected. All the previous flights will be recovered through retransmissions. Indeed, given the fact that the receive window is large enough compared to the congestion window, there will be no silence period implied by any packet loss except for the last flight of packets. The total download completion time will thus not be impacted by any loss before the last flight of packets. Without using FEC, the loss of any packet in the last flight will cause a silence period between the sending of that lost packet and its retransmission. We track the loss conditions throughout the download and trigger the $FECPattern()$ threshold according to the observed loss pattern. This loss-rate-adaptive approach is especially beneficial when enough packets are exchanged to accurately estimate the loss pattern. This occurs when the file is long or when loss information is shared among connections with the same peer. When a sufficient number of repair symbols are sent to protect the expected number of lost source symbols, the algorithm keeps slots in its congestion window to transmit new data. Another approach would be to define $FECPattern()$ to use all the remaining space in the congestion window to send repair Symbols, with the drawback of potentially consuming more bandwidth than needed.

B. Evaluation

We now evaluate *FlEC* with the $ds()$ and $FECPattern()$ protocol operations defined for the bulk use-case.

1) Experimental setup: We base our implementation on the PQUIC [12] pluginized QUIC implementation on commit *68e61c5* [65]. PQUIC is itself based on the `picoquic` [66] QUIC implementation. We perform numerous experiments and compare it with the regular QUIC without our plugins. We use ns-3 [67] version 3.33 with the Direct Code Execution (DCE) [68] module. The DCE module allows using ns-3 with the code of a real implementation in a discrete time environment. This means that the actual code of the QUIC and *FlEC* implementation is running and that the underlying network used by the implementation is simulated by ns-3, making the experiments fully reproducible while running real code. Figure 2 shows the experimental setup. We use ns-3's *RateErrorModel* to generate reproducible loss patterns with different seeds and configure the network queues to 1.5 times the bandwidth-delay product. We run the system into a Ubuntu

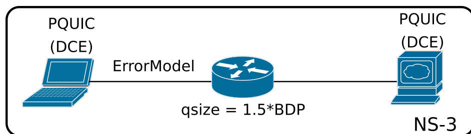

Figure 2. Experimental topology using NS-3 with Direct Code Execution.

16.04 Linux system with 20GB of RAM, using 16 cores *Intel(R) Xeon(R) Silver 4314* CPUs.

Although the congestion control is orthogonal to our proposed reliability mechanism, the Reno [69] and CUBIC [70] congestion control algorithms supported by PQUIC suffer from bandwidth underestimation under severe loss conditions. We thus perform experiments using the BBR [71] congestion control algorithm. BBR avoids underestimating the network bandwidth upon packet losses by looking at the receive rate and delay variation during the transfer. While not being explored in this paper, other congestion control algorithms [72], [73] use other signals than packet losses to detect congestion.

2) Experimental design: We evaluate the bulk use-case by sending files of several sizes and first see how *FlEC* compares with QUIC using its regular reliability mechanism. For this evaluation, we rely on an experimental design [74]. This approach consists in defining ranges of parameters instead of choosing precise values in order to mitigate the experimentation bias and explore network configurations showing the limits of the presented solution. We use the WSP [75] space-filling algorithm to cover the parameter space with 94 points. One experiment is run for each point in the parameter space.

Figure 3 shows the cumulative distribution function (CDF) of the Download Completion Time (DCT) ratio between *FlEC* and `picoquic` [66] used as our reference QUIC implementation. The experiments consist in the download files of size 10kB, 40kB, 100kB, 1MB and 10MB. For each file size, 95 experiments are run using experimental design. 40kB and 100kB are the average response sizes for Google Search on mobile and desktop devices [6]. The parameter space is described on top of the Figure. The loss rate varies between 0.1% and 8% to cover both small loss rates and loss rates experienced under intense network conditions such as In-Flight Communications [76]. The round-trip-time varies between 10ms and 200ms to experience both low delays and large delays such as those encountered in satellite communications. As shown in the Figure, $ds()$ and $FECPattern()$ implement here a bulk-friendly reliability mechanism. By automatically protecting the tail of the downloaded file, we obtain similar results as previous works [12], [17]. A few of the experiments with 40 and 100kB files provided poorer results compared to QUIC. With those file sizes, *FlEC* uses one more stream frame to transmit the data, needing in some rare cases one additional round-trip to transmit this additional packet. While not shown graphically in this article, replacing BBR by Cubic [70] provides similar results. These experiments are provided in the artefacts that come with the article upon publication.

Figure 4 compares *FlEC* with an implementation of AC-RLNC [53] following Table II. We observe that *FlEC* still outperforms AC-RLNC as sending repair symbols every RTT consumes too much bandwidth for the bulk use-case, while *FlEC* only sends repair symbols *a priori* for the last flight of packets, relying on retransmissions for all the other packets as their retransmission arrives before the end of the download.

3) Experimenting with a real network: We now extend our study and analyze the benefits of *FlEC* over a real network between a regular QUIC and *FlEC* server on a Ubuntu 18.04 server located at UCLouvain and a client wired to a Starlink access point located in Louvain-la-Neuve (Belgium). We performed a total of 20150 uploads of 50kB from the client to the server. Among those 20150 uploads, 430 encountered at least one packet loss during the transfer. Figure 5 shows the CDF of the download completion time for these 430 uploads. The median download completion time for these uploads is 247ms for *FlEC* and 272ms for regular QUIC. The average download completion time is 340ms for *FlEC* and 393ms for QUIC. Unsurprisingly, *FlEC* improves the download completion time for the transfers where the loss events occur during the RTT.

4) CPU performance: While it has been demonstrated that PQUIC protocol plugins deteriorate noticeably the performance [12], we analyze the CPU impact of the *FlEC* framework by transferring 1GB files on the loopback interface. Without *FlEC*, we achieved a throughput of 650 Mbps. With *FlEC* configured for the bulk use-case (i.e. sending Repair Symbols at the end of the transfer only), it dropped to 300 Mbps. This is inline with earlier observations on PQUIC performance. We believe that with a native implementation, the impact of the *FlEC* framework would be barely noticeable. We also analyzed the throughput sending one Repair Symbol every ten Source Symbols and obtained a throughput (i.e. not goodput) of 280 Mbps, meaning that the encoding and decoding of Repair Symbols implies only a small overhead compared to the framework in itself.

VII. BUFFER-LIMITED FILE TRANSFERS

We here present and evaluate the reliability mechanism for buffer-limited file transfers. In this setup, the receive window (*rwin*) is relatively small compared to the congestion window (*cwin*) of the sender, making every loss event potentially blocking and increasing the download completion time. In addition to protect the download from tail losses, we protect every window of packets to avoid stalling due to lost packets blocking the stream flow-control window.

A. Reliability mechanism

For this use-case, $ds()$ returns $\hat{l}$ to ensure that ad stays larger than md, according to the estimated loss rate. $FECPattern()$ behaves as shown in Algorithm 2. We spread the Repair Symbols along the sent Source Symbols in order to periodically allow the receiver to unblock its receive window by recovering the lost Source Symbols and deliver the stream data in-order to the application. More precisely, the $FECPattern()$ operation sends one Repair Symbol every $\frac{1}{\hat{l}}$ Source Symbols. The algorithm needs three loss statistics. The first is the estimated uniform loss rate $\hat{l}$. The two others are the $\hat{G}_p$ and $\hat{G}_r$ parameters of the Gilbert loss model. The Gilbert

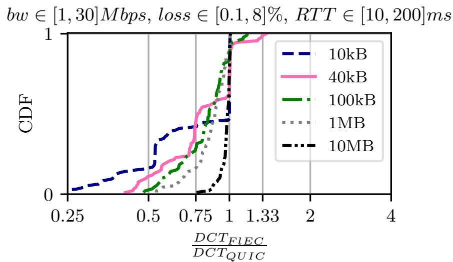

Figure 3. DCT ratio for bulk use-case using BBR. $FECPattern()$ and $ds()$ ensure that Repair Symbols only protect the tail of the file.

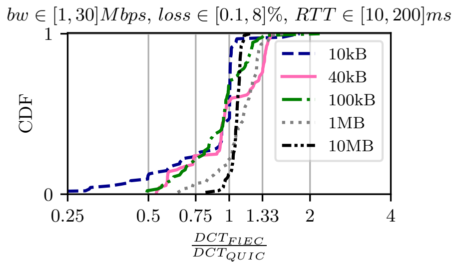

Figure 4. DCT ratio between *FlEC* and AC-RLNC [53] for regular bulk use-case using the BBR congestion control.

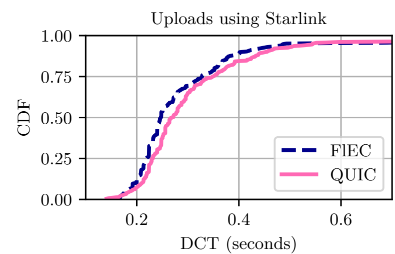

Figure 5. DCT comparing *FlEC* and the regular QUIC for downloads with at least one packet loss, performed on a real Starlink network access.

model [64] is a two-states Markov model representing the channel, allowing representing network configurations where losses occur in bursts. These loss patterns cannot be easily recovered by a simple XOR error correcting code as shown in the original QUIC article [6] but can be recovered by the random linear codes used by *FlEC*. In the *GOOD* state of the Gilbert model, packets are received while the packets are dropped in the *BAD* state. $\hat{G_p}$ is the transition probability from the GOOD to the BAD state while $\hat{G_r}$ is the transition probability from the BAD to the GOOD state. In order to estimate the loss statistics $\hat{l}$, $\hat{G_p}$ and $\hat{G_r}$, we implement a *loss monitor* that estimates the loss rate and Gilbert model parameters over a QUIC connection.

When the sender is blocked by the QUIC stream flow control, $FECPattern()$ sends more Repair Symbols to recover from the remaining potentially lost Source Symbols. While spreading the Repair Symbols along the coding window helps to recover the lost Source Symbols more rapidly compared to a block approach where all the repair symbols are sent at the end of the window, this also potentially consumes more bandwidth. Indeed, the Repair Symbols do not protect the entire window. This means that with an equal number of losses, some specific loss patterns will lead to Repair Symbols protecting a portion of the window with no loss and portions of the window requiring more Repair Symbols to be recovered.

Algorithm 2 FECPattern for buffer-limited use-case

Require: $last$, the ID of the last symbol present in the coding window when $FECPattern()$ was triggered the last time
Require: $nTriggered$, the number of times FECPattern() has already been triggered since no new symbol was added to the window.
Require: $maxTrigger$, the maximum number of times we can trigger this threshold for the same window
Require: $nRSInFlight$, the number of Repair Symbols currently in flight
Require: W, the current coding window.
Require: $FCBlocked()$, telling us if we are currently blocked by flow control.
Require: $\hat{l}$, $\hat{G_p}$, $\hat{G_r}$, see Table I.
1: **if** $nRSInFlight \geq 2 * \lceil |W| * \hat{l} \rceil$ **then**
2: **return** $false$ ▷ Wait for feedback before sending new RS
3: **end if**
4: $nUnprotected \leftarrow W.last - last$
5: $n \leftarrow min(\frac{1}{\hat{G_p}}, |W|)$
6: $protect \leftarrow nUnprotected = 0 \vee nUnprotected \geq n \vee FCBlocked()$
7: **if** $protect \wedge nUnprotected \neq 0$ **then** ▷ *Start Repair Symbols sequence*
8: $nTriggered \leftarrow 1$
9: $last \leftarrow W.last$
10: $maxTrigger \leftarrow \lceil max(\hat{l} * nUnprotected, \frac{1}{\hat{G_r}}) \rceil$
11: **else if** $protect$ **then**
12: **if** $FCBlocked() \vee nTriggered < maxTrigger$ **then**
13: $nTriggered \leftarrow nTriggered + 1$ ▷ *Continue sending symbols*
14: **else**
15: $protect \leftarrow false$ ▷ *Enough symbols have been sent*
16: **end if**
17: **end if**
18: **return** $protect$

B. Evaluation

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

1) FlEC for SATCOM: We choose the satellite communications (SATCOM) use-case where the delay can easily reach several hundreds of milliseconds [57], [58]. In those cases, end-hosts need a large receive buffer in order to reach the channel capacity. If they do not use a sufficiently large buffer, packet losses can have a significant impact on the throughput, preventing the sender to send new data as long as the data at the head of the receive buffer have not been correctly delivered to the application. The studied network configuration has a round-trip-time of 400 milliseconds and a bandwidth of 8 Mbps. Those are lower-bound values compared to current deployments [57], [58]. The bandwidth-delay product is thus 400kB. Higher BDP configurations are studied in the experimental design analysis of the next section. We study the benefits brought by *FlEC* with several receive window sizes.

a) Download completion time and throughput: Figure 6 shows the download completion time ratio between *FlEC* and regular QUIC with a 5 MB file and 0.5% of packet loss. Each box in the graph is computed from 95 runs with different seeds for the ns-3 rate error model. The bandwidth is set to 8 Mbps and the congestion control is BBR. For each transfer using *FlEC*, we decrease the receive window by 5% at the receiver in order to store the received repair symbols in the remaining

space. With receive windows smaller than the BDP (ranging from 70 kB to 400 kB), the sender is flow-control-blocked once per RTT during a time proportional to the $\frac{rwin}{cwin}$ ratio. This implies that the download completion time with small receive windows is large even without any packet loss. When losses occur, the repair symbols sent *a priori* help to unblock the receive window at the receiver-side and avoid blocking the data transfer for more than one RTT. For the 70 kB receive window, the 5% reduction to store the repair symbols is significant compared to the benefit of FEC and has a negative impact on the goodput. With the 400 kB receive window, the sender only blocks in the presence of losses during the round-trip. The earlier the loss occurs during the round-trip, the longer the sender will be blocked by the flow control for the next round-trip, since it needs to retransmit the data to unblock the receive window. Sending *a priori* Repair Symbols for these configurations allows reducing or completely avoiding those blocking situations, at the price of a small reduction in goodput. The transmission of Repair Symbols in a sliding-window manner (i.e. interleaved with the Source Symbols) as described in Algorithm 2 helps to recover from losses earlier compared to sending all the Repair Symbols at once in a block fashion. The price to pay compared to a block pattern is an goodput reduction as some loss patterns might require more Repair Symbols to be recovered with this method. For the large receive windows, sending Repair Symbols *a priori* does not unblock the window but still helps to recover from tail losses. With such a high RTT, the impact of a tail loss relative to the download completion time is still significant.

Figure 7 shows the result of our experiments with a 2% packet loss rate. It is thus more common that the sender becomes flow-control blocked. This makes the approach worth even for smaller receive window sizes such as 70kB as the sender will be slowed down a lot more often.

b) Delay-bandwidth tradeoff: Figure 8 illustrates the delay-bandwidth tradeoff operated when using *FlEC* instead of regular QUIC. Each point on the figure concerns a single experiment and represents the download completion time and the bytes overhead of the solution. The bytes overhead is computed by dividing the total amount of bytes of UDP payload sent by the server by the size of the file transferred (5MB). For this graph, the experiments use a small receive window of 150kB and the loss rate is 2%. As the receive window is small, sending FEC unblocks the receive window upon losses and allows drastically lowering the download completion time. The price to pay is an additional bytes overhead compared to the regular QUIC solution. In this rwin-limited scenario, the available bandwidth is generally larger than what is used due to the rwin restriction.

Figure 9 shows experiments results with the opposite scenario: the receive window is 6MB large, which is larger than both the file to transfer and the bandwidth-delay product of the link. This case is similar to the bulk use-case of section VI. We can see that *FlEC* leads to stable latency results at the expense of a larger bytes overhead. As the receive window is larger than the file to transfer, the sender will never be flow-control blocked during the download. In this case, *FlEC* minimizes the latency essentially by recovering from tail losses.

2) Experimental design analysis: Figure 10 shows the aggregated results of simulations using experimental design. We show the CDF of the download completion time ratio between *FlEC* and `picoquic` [66]. Each CDF on the figure is built from 95 experiments with parameters selected from the ranges depicted on top of Figure 10. Each CDF curve corresponds to downloads using the receive window size specified in the legend. The congestion control used is still BBR. We observe positive results using *FlEC* for the majority (75%) of the network configurations, especially for smaller receive window sizes (80% positive results for windows smaller or equal to 400kB). Some configurations still expose negative results using *FlEC*, even for smaller receive window sizes. These configurations are those whose bandwidth-delay product is small compared to the receive window. To verify this, we computed the average $\frac{BDP}{rwin}$ ratio on all the experiments for which *FlEC* took more time to complete than `picoquic`, whose value is 0.48. For the experiments where the *FlEC* download was faster, the average value of this ratio is 1.53.

Let us now assess the performance of our solution using a bursty loss model in order to see whether *FlEC* stays robust even in presence of loss bursts. Figure 11 shows the results of an experimental design analysis with a Gilbert loss model with $G_{\hat{p}}$ ranging from 0.1% to 1.5% and $G_{\hat{r}}$ set to 33% (i.e. an expected burst size of 3 packets) and a maximum burst size of 5 packets. Loss events thus occur less often compared to Figure 10, leading to fewer blocking periods for QUIC during the experiments but with a higher probability of loosing several packets in a row. We can see that Algorithm 2 still offers benefits in the presence of bursty losses. Similarly to Figure 10, *FlEC* especially improves the results for experiments with a large $\frac{cwin}{rwin}$ ratio.

VIII. Delay-constrained messaging

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

A. Reliability mechanism

We consider an application sending variable-sized messages, each having its own delivery deadline. To convey these deadlines, we extend the transport API (Section VIII-A1). Furthermore, we replace the QUIC stream scheduler to leverage application information (Section VIII-A2). This can be done easily since applications are bundled with their QUIC implementation and are able to easily extend it. We then discuss and evaluate a specific use-case in Section VIII-B.

1) Application-specific API: We propose the following API enabling an application to send deadline-constrained messages.

a) `send_fec_protected_msg(msg, deadline)`: The application submits its deadline-constrained messages. The QUIC protocol already supports

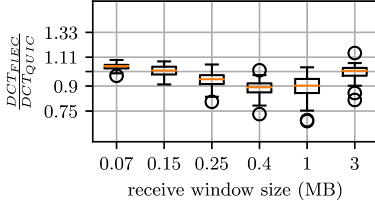

Figure 6. DCT ratio, 0.5% losses.

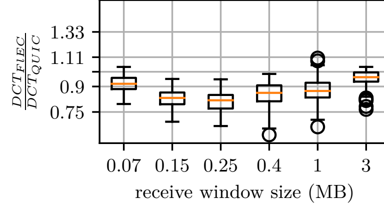

Figure 7. DCT ratio, 2% losses

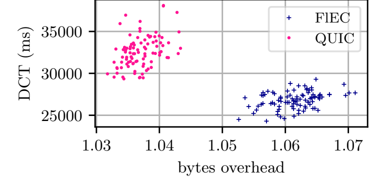

Figure 8. Time-bandwidth tradeoff, 2% loss.

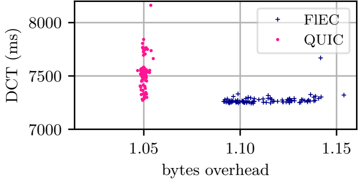

Figure 9. Time-bandwidth tradeoff with a 0.5% loss link and a 6MB receive window.

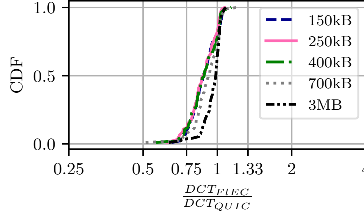

Figure 10. Experimental design analysis for several receive window configurations.

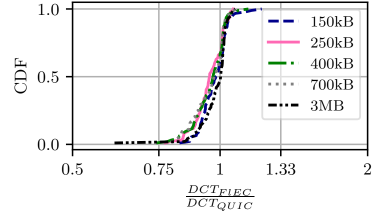

Figure 11. Experimental design analysis using Gilbert model with bursts of 1 to 5 packets.

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

b) `next_message_arrival(arrival_time)`*:* This API call allows the application to indicate when it plans to submit the next message. While this API function is not useful for all kinds of unreliable messaging applications, applications having a constant message sending rate such as video-conferencing might benefit from providing such information.

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

The $closestDeadlineStream()$ function searches among all the available streams attached to a deadline to find the stream having the closest expiration deadline while still having chances to arrive on-time given the current one-way delay. The scheduler chooses the non-flow-control blocked stream that is the closest to expire while still having a chance to be delivered on-time to the destination. Our implementation

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

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

1: $scheduledStream \leftarrow \varnothing$
2: $currentDeadline \leftarrow now + \hat{OWD}$ ▷ Initialization
3: **while** $scheduledStream = \varnothing$ **do**
4: $\quad candidate \leftarrow closestDeadlineStream(\mathcal{S}, currentDeadline)$
5: $\quad$ **if** $candidate = \varnothing$ **then break**
6: $\quad$ **end if**
7: $\quad$ **if** $\neg FCBlocked(candidate)$ **then**
8: $\quad\quad scheduledStream \leftarrow candidate$
9: $\quad$ **else**
10: $\quad\quad \mathcal{S} \leftarrow \mathcal{S} \setminus \{candidate\}$
11: $\quad\quad currentDeadline \leftarrow candidate.deadline$
12: $\quad$ **end if**
13: **end while**
14: **if** $scheduledStream = \varnothing$ **then**
15: $\quad$ **return** $defaultStreamScheduling(\mathcal{S})$ ▷ Fallback
16: **end if**

estimates the one-way delay as $\frac{RTT}{2}$. Other methods exist [77], [78]. Recent versions of `picoquic` include a mechanism for estimating the one-way delay [66] when the hosts clocks are synchronized. In the absence of clock synchronization, the estimated one-way delay can only be interpreted relatively, which helps to estimate the one-way delay variation but not for decision thresholds such as the one used in Algorithm 3.

3) FECPattern() and ds() for delay-constrained messaging: We now describe how our application redefines *FlEC*. Our application is sensitive to the delivery delay of entire messages more than the in-order delivery delay of individual packets.

We thus set the $ds()$ threshold to $-\hat{l}$ as it is useful to retransmit non-recovered lost packets that can still arrive on-time. $FECPattern()$ is described in Algorithm 4. The algorithm triggers the sending of Repair Symbols to protect as many messages as possible according to the messages deadline and the next expected message timestamp if provided by the application. The rationale is the following. If the unprotected messages can wait for new messages to arrive before being protected, $FECPattern()$ does not send Repair Symbols and waits for the arrival of new messages. Otherwise, Repair Symbols are sent to protect the entire window until it is considered fully protected. This idea of waiting for new messages before protecting comes from the fact that the messages can be small and sending Repair Symbols for each sent message can lead to a high overhead. By doing so, $FECPattern()$ adapts the code rate according to the application needs.

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

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

1: $nextDL \leftarrow closestDL(\mathcal{S}, max(now + \hat{OWD}, last.deadline))$
2: $protect \leftarrow (nextDL = \varnothing \vee now + \hat{OWD} + nextMsg + \epsilon \geq nextDL)$
3: $nUnprotected \leftarrow W.last - last$
4: **if** $protect \wedge nUnprotected \neq 0$ **then** ▷ *Start Repair Symbols sequence*
5: $\quad nTriggered \leftarrow 1$
6: $\quad last \leftarrow W.last$
7: $\quad maxTrigger \leftarrow \lceil max(\hat{l} * nUnprotected, \frac{1}{\hat{G_r}}) \rceil$
8: **else if** $protect$ **then**
9: $\quad$ **if** $nTriggered < maxTrigger$ **then**
10: $\quad\quad nTriggered \leftarrow nTriggered + 1$ ▷ *Continue sending*
11: $\quad$ **else**
12: $\quad\quad protect \leftarrow false$ ▷ *Enough symbols have been sent*
13: $\quad$ **end if**
14: **end if**
15: **return** $appLimited() \wedge protect \wedge \frac{cwin}{bif} > \theta$

B. Evaluation

We evaluate *FlEC* under the messaging use-case using an application sending video frames as messages. We set the deadline to 250 milliseconds, meaning that each frame must be delivered within this time. We use 86 seconds of the video recording from the Tixeo video-conference application [79]. The framerate and bitrate are adjusted by the application. This video recording starts at 15 frames per second during the first 6 seconds and runs at 30 images per second afterwards. For frame, we record its delivery delay between when the application sends it and when it is delivered at the receiver. We send each video frame in a different QUIC stream to avoid head-of-line blocking across frames upon packet losses. The regular QUIC solution uses the default round-robin scheduler provided by PQUIC. In the first set of experiments, we set the bandwidth to 8 Mbps and observe the performance of *FlEC* in the presence of losses. For each experiment, the delay is sampled in the $[5, 200]ms$ range. We then perform an experimental design analysis over a wider parameters space.

Figure 12 and Figure 13 show the 95th and 98th percentiles of the message delivery times for each experiment. We can see that while 95% of the video frames are delivered successfully in every experiment, regular QUIC struggles to deliver 98% of the submitted frames on time (i.e. before 250 milliseconds) with a one-way delay above 75 milliseconds. Indeed, with a one-way delay above 75 milliseconds, the lost frames are retransmitted after more than 150 milliseconds and take more than 75 milliseconds to reach the receiver. Note that QUIC's loss detection mechanism takes a bit more than one RTT to consider a packet as lost to avoid spurious retransmissions due to reordering [15]. These retransmitted frames thus arrive a few milliseconds before the deadline in the best case. As we can see on Figure 13, only a few experiments without *FlEC* have more than 98% of the frames arriving on-time while *FlEC* can cope with one-way delays up to 200 milliseconds. Figure 14 shows that no experiment with regular QUIC succeeded to deliver 99% of the video frames on time with a one-way delay above 75ms, while *FlEC* succeeded in every experiment.

Note that the $FECPattern()$ algorithm plugged in this use-case tries to protect as many messages as possible with the same number of Repair Symbols by delaying the sending of Repair Symbols when new messages are expected soon. This lazy Repair Symbol scheduling explains the plateau present around the 250ms delivery time in Figure 14 and why the frame delivery time is larger than the one-way delay. In order to send as few Repair Symbols as possible, *FlEC* delays the sending of Repair Symbols to the last possible moment while ensuring that lost data can be recovered before the deadline.

Figure 15 shows the ratio between the number of messages received on-time by *FlEC* and by the regular QUIC implementation. In order to isolate the effects of the *FlEC* API, the Figure also shows *FlEC* results without leveraging the application knowledge brought by the API functions ($FlEC_{NO-API}$ on the Figure). It thus uses `picoquic`'s default stream scheduler and sends repair symbols for each newly sent message. As it is only a simplified version of Algorithm 4, we do not show the $FECPattern()$ algorithm of this second solution. As we can see, nearly no experiment ended with fewer messages received on-time using the API-enabled *FlEC* compared to QUIC. A similar gain compared to the regular QUIC is present for both *FlEC* versions. However, the interest of the *FlEC* API resides in the redundancy it needs to obtain those results.

We now analyze the redundancy overhead of our solution.

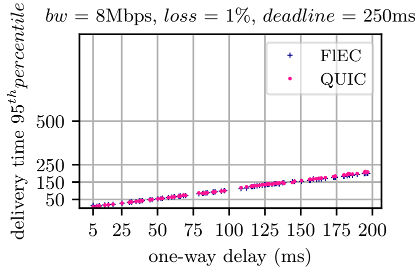

Figure 12. Message delivery time 95th percentile, comparing *FlEC* with API and the regular QUIC.

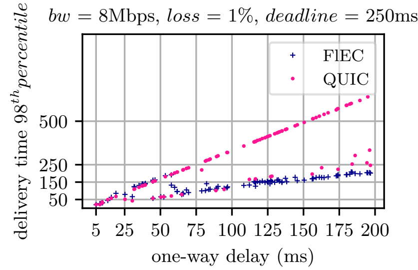

Figure 13. Message delivery time 98th percentile, comparing *FlEC* with API and the regular QUIC.

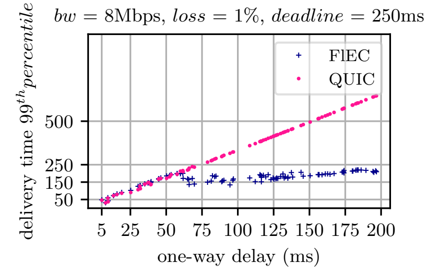

Figure 14. Message delivery time 99th percentile, comparing *FlEC* with API and the regular QUIC.

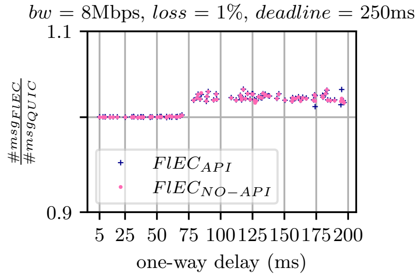

Figure 15. Messages received on-time comparing QUIC and *FlEC* with and without API, using BBR.

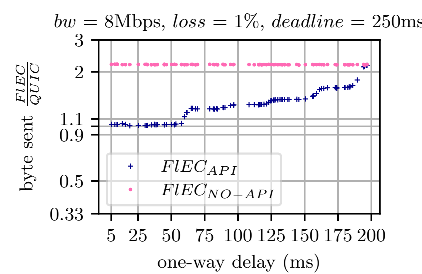

Figure 16. Bytes sent by the server, comparing QUIC and *FlEC* with and without API, using BBR.

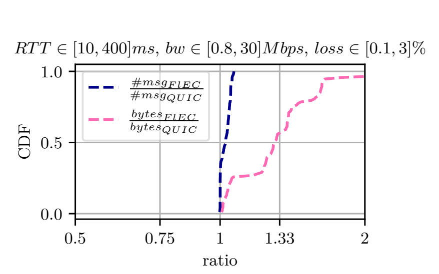

Figure 17. Experimental design analysis for the delay-constrained messaging use-case using BBR.

Figure 16 shows the ratio of bytes sent by the server between regular QUIC and *FlEC* with and without the API defined in Section VIII-A1. The results of *FlEC* without the API show that protecting every message blindly is very costly in terms of bandwidth. Indeed, for this video-conferencing transfer, many video frames sent by the application are smaller than the size of a full QUIC packet. The QUIC `REPAIR` frames sent by *FlEC* contain additional metadata. In this case where the application traffic is thin, protecting every message may double the volume of sent data as shown on Figure 16. Using *FlEC* with the message-based API can save a lot of bandwidth by using application-aware stream and redundancy schedulers. Note the portion of the graph between 5ms and 70ms one-way delays. For those configurations, no Repair Symbol is sent by *FlEC*. Indeed, the messages are acknowledged by the peer before $FECPattern()$ triggers the sending of repair symbols. *FlEC* thus naturally uses SR-ARQ when redundancy is not needed to meet the messages deadlines.

Experimental design analysis: Figure 17 shows the results of an experimental design analysis using the parameters depicted on the top of the Figure. The Figure shows CDFs for the amount of bytes sent by the server and the number of messages received within the deadline. In a few cases, the *FlEC* solution using the application-tailored API sends a similar amount of bytes to regular QUIC. This is due to the fact that for some configurations, the delay was sufficiently low to send no or a few repair symbols. We can also see that none of the experiments revealed a lower amount of on-time received messages compared to regular QUIC, showing the robustness of *FlEC* under various network conditions.

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

IX. Conclusion

In this paper, we redefine the QUIC reliability mechanism and enable its per-use-case customization. Flexible Erasure Correction (*FlEC*) allows efficiently combining retransmissions and Forward Erasure Correction. Applications can either use a standard Selective-Repeat ARQ mechanism or tailor a Forward Erasure Correction mechanism that fits their own traffic pattern and sensitivity to delays. Our *FlEC* implementation leverages the PQUIC protocol plugins to enable the application to insert its own algorithm to select the level of redundancy and the stream scheduling decisions. We customize *FlEC* for three different use-cases. We evaluate and demonstrate that *FlEC* can be configured with small to no effort by applications to significantly enhance the quality of experience compared to the existing QUIC loss recovery mechanisms. *FlEC* is currently a single-path implementation. In the future, we plan to study how *FlEC* can be used together with several network interfaces to improve the transfer for the considered use-case and go further than reducing head-of-line blocking using a tailored redundancy and path scheduler.

Artefacts

Simulation scripts and the code of *FlEC* are publicly available from https://github.com/francoismichel/flec.

References

[1] E. Kohler, M. Handley, and S. Floyd, "Designing DCCP: Congestion control without reliability," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 4, pp. 27–38, 2006.

[2] A.-V. T. W. Group, H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications," RFC 1889 (Proposed Standard), RFC Editor, Fremont, CA, USA, Jan. 1996, obsoleted by RFC 3550. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1889.txt

[3] T. Pauly, E. Kinnear, and D. Schinazi, "An Unreliable Datagram Extension to QUIC," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-datagram-10, Feb. 2022, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-quic-datagram-10

[4] J. Postel, "Transmission Control Protocol," RFC 793 (Internet Standard), RFC Editor, Fremont, CA, USA, Sep. 1981, updated by RFCs 1122, 3168, 6093, 6528. [Online]. Available: https://www.rfc-editor.org/rfc/rfc793.txt

[5] R. Stewart (Ed.), "Stream Control Transmission Protocol," RFC 4960 (Proposed Standard), RFC Editor, Fremont, CA, USA, Sep. 2007, updated by RFCs 6096, 6335, 7053, 8899. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4960.txt

[6] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar *et al.*, "The QUIC transport protocol: Design and internet-scale deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 183–196.

[7] J. Iyengar (Ed.) and M. Thomson (Ed.), "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000 (Proposed Standard), RFC Editor, Fremont, CA, USA, May 2021. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9000.txt

[8] J. Zirngibl, P. Buschmann, P. Sattler, B. Jaeger, J. Aulbach, and G. Carle, "It's over 9000: Analyzing early QUIC deployments with the standardization on the horizon," *Proceedings of the 2021 ACM SIGCOMM conference on Internet measurement conference*, 2021.

[9] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446 (Proposed Standard), RFC Editor, Fremont, CA, USA, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8446.txt

[10] "Quic implementations," https://github.com/quicwg/base-drafts/wiki/Implementations, accessed: 2022-03-23.

[11] A. M. Kakhki, S. Jero, D. Choffnes, C. Nita-Rotaru, and A. 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*, 2017, pp. 290–303.

[12] Q. De Coninck, F. Michel, M. Piraux, F. Rochet, T. Given-Wilson, A. Legay, O. Pereira, and O. Bonaventure, "Pluginizing QUIC," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 59–74.

[13] M. Polese, F. Chiariotti, E. Bonetto, F. Rigotto, A. Zanella, and M. Zorzi, "A survey on recent advances in transport layer protocols," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 4, pp. 3584–3608, 2019.

[14] P. Garrido, I. Sánchez, S. Ferlin, R. Agüero, and O. Alay, "rQUIC: Integrating FEC with QUIC for robust wireless communications," in *IEEE Globecom*, 2019.

[15] J. Iyengar and I. Swett, "QUIC loss detection and congestion control," RFC 9002, May 2021. [Online]. Available: https://rfc-editor.org/rfc/rfc9002.txt

[16] I. Swett, M.-J. Montpetit, V. Roca, and F. Michel, "Coding for quic," Working Draft, IETF Secretariat, Internet-Draft draft-swett-nwcrg-coding-for-quic-03, July 2019, http://www.ietf.org/internet-drafts/draft-swett-nwcrg-coding-for-quic-03.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-swett-nwcrg-coding-for-quic-03.txt

[17] F. Michel, Q. De Coninck, and O. Bonaventure, "QUIC-FEC: Bringing the benefits of Forward Erasure Correction to QUIC," *IFIP Networking 2019*, 2019.

[18] M. Zverev, P. Garrido, F. Fernández, J. Bilbao, Ö. Alay, S. Ferlin-Reiter, A. Brunström, and R. Agüero, "Robust QUIC: Integrating practical coding in a low latency transport protocol," *IEEE Access*, 2021.

[19] D. Bertsekas and R. Gallager, *Data networks*. Prentice-Hall International New Jersey, 1992.

[20] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm," RFC 6582 (Proposed Standard), RFC Editor, Fremont, CA, USA, Apr. 2012. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6582.txt

[21] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Options," RFC 2018 (Proposed Standard), RFC Editor, Fremont, CA, USA, Oct. 1996. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2018.txt

[22] R. Ludwig and R. H. Katz, "The eifel algorithm: making TCP robust against spurious retransmissions," *ACM SIGCOMM Computer Communication Review*, vol. 30, no. 1, pp. 30–36, 2000.

[23] Y. Cheng, N. Cardwell, N. Dukkipati, and P. Jha, "The RACK-TLP Loss Detection Algorithm for TCP," RFC 8985, Feb. 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc8985

[24] M. Rajiullah, P. Hurtig, A. Brunstrom, A. Petlund, and M. Welzl, "An evaluation of tail loss recovery mechanisms for TCP," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 1, pp. 5–11, 2015.

[25] M. Allman, K. Avrachenkov, U. Ayesta, J. Blanton, and P. Hurtig, "Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)," RFC 5827 (Experimental), RFC Editor, Fremont, CA, USA, May 2010. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5827.txt

[26] Ł. Budzisz, J. Garcia, A. Brunstrom, and R. Ferrús, "A taxonomy and survey of SCTP research," *ACM Computing Surveys (CSUR)*, vol. 44, no. 4, pp. 1–36, 2012.

[27] E. W. Biersack, "Performance evaluation of forward error correction in ATM networks," *ACM SIGCOMM Computer Communication Review*, vol. 22, no. 4, pp. 248–257, 1992.

[28] G. Carle and E. W. Biersack, "Survey of error recovery techniques for IP-based audio-visual multicast applications," *IEEE Network*, vol. 11, no. 6, pp. 24–36, 1997.

[29] J. Gemmell, T. Montgomery, T. Speakman, and J. Crowcroft, "The PGM reliable multicast protocol," *IEEE network*, vol. 17, no. 1, pp. 16–22, 2003.

[30] M. Luby, L. Vicisano, J. Gemmell, L. Rizzo, M. Handley, and J. Crowcroft, "Forward Error Correction (FEC) Building Block," RFC 3452 (Experimental), RFC Editor, Fremont, CA, USA, Dec. 2002, obsoleted by RFCs 5052, 5445. [Online]. Available: https://www.rfc-editor.org/rfc/rfc3452.txt

[31] H. Schulzrinne and J. Rosenberg, "An RTP Payload Format for Generic Forward Error Correction," RFC 2733, Dec. 1999. [Online]. Available: https://rfc-editor.org/rfc/rfc2733.txt

[32] M. Luby, M. Watson, and T. Stockhammer, "RTP Payload Format for Raptor Forward Error Correction (FEC)," RFC 6682, Aug. 2012. [Online]. Available: https://rfc-editor.org/rfc/rfc6682.txt

[33] J. K. Sundararajan, D. Shah, M. Médard, M. Mitzenmacher, and J. Barros, "Network coding meets TCP," in *IEEE INFOCOM 2009*. IEEE, 2009, pp. 280–288.

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

[35] M. Médard, S. Katti, D. Katabi, W. Hu, H. Rahul, and J. Crowcroft, "XORs in the past and future," *SIGCOMM Comput. Commun. Rev.*, vol. 49, no. 5, p. 77–81, Nov. 2019. [Online]. Available: https://doi.org/10.1145/3371934.3371959

[36] J. K. Sundararajan, D. Shah, M. Médard, S. Jakubczak, M. Mitzenmacher, and J. Barros, "Network coding meets TCP: Theory and implementation," *Proceedings of the IEEE*, vol. 99, no. 3, pp. 490–512, 2011.

[37] Y. Cui, L. Wang, X. Wang, H. Wang, and Y. Wang, "FMTCP: A fountain code-based multipath transmission control protocol," *IEEE/ACM Transactions on Networking*, vol. 23, no. 2, pp. 465–478, 2014.

[38] I. Swett, "QUIC WG charter: FEC initially out of scope," Presentation at IETF99, 2017.

[39] N. Kuhn, E. Lochin, F. Michel, and M. Welzl, "Coding and congestion control in transport," Internet Engineering Task Force, Internet-Draft draft-irtf-nwcrg-coding-and-congestion-09, Jun. 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-irtf-nwcrg-coding-and-congestion-09

[40] A. Li (Ed.), "RTP Payload Format for Generic Forward Error Correction," RFC 5109 (Proposed Standard), RFC Editor, Fremont, CA, USA, Dec. 2007. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5109.txt

[41] J. Uberti, "WebRTC forward error correction requirements," Tech. Rep.

[42] B. Cavusoglu, D. Schonfeld, and R. Ansari, "Real-time adaptive forward error correction for MPEG-2 video communications over rtp networks,"

in *2003 International Conference on Multimedia and Expo. ICME'03. Proceedings (Cat. No. 03TH8698)*, vol. 3. IEEE, 2003, pp. III–261.
[43] P. U. Tournoux, E. Lochin, J. Lacan, A. Bouabdallah, and V. Roca, "On-the-fly erasure coding for real-time video applications," *IEEE Transactions on Multimedia*, vol. 13, no. 4, pp. 797–812, 2011.
[44] M. F. Nowlan, N. Tiwari, J. Iyengar, S. O. Amin, and B. Ford, "Fitting square pegs through round pipes: Unordered delivery wire-compatible with {TCP} and {TLS}," in *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, 2012, pp. 383–398.
[45] N. Kuhn, E. Lochin, A. Mifdaoui, G. Sarwar, O. Mehani, and R. Boreli, "Daps: Intelligent delay-aware packet scheduling for multipath transport," in *2014 IEEE international conference on communications (ICC)*. IEEE, 2014, pp. 1222–1227.
[46] F. Chiariotti, A. Zanella, S. Kucera, K. Fahmi, and H. Claussen, "The hop protocol: Reliable latency-bounded end-to-end multipath communication," *IEEE/ACM Transactions on Networking*, vol. 29, no. 5, pp. 2281–2295, 2021.
[47] A. Garcia-Saavedra, M. Karzand, and D. J. Leith, "Low delay random linear coding and scheduling over multiple interfaces," *IEEE Transactions on Mobile Computing*, vol. 16, no. 11, pp. 3100–3114, 2017.
[48] H. Schulzrinne, S. L. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications," RFC 3550, Jul. 2003. [Online]. Available: https://rfc-editor.org/rfc/rfc3550.txt
[49] P. S. Schmidt, T. Enghardt, R. Khalili, and A. Feldmann, "Socket intents: Leveraging application awareness for multi-access connectivity," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, 2013, pp. 295–300.
[50] T. Pauly, B. Trammell, A. Brunstrom, G. Fairhurst, C. Perkins, P. S. Tiesel, and C. A. Wood, "An Architecture for Transport Services," Internet Engineering Task Force, Internet-Draft draft-ietf-taps-arch-06, Dec. 2019, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-taps-arch-06
[51] J. Iyengar and M. Thomson, "QUIC: A UDP-based multiplexed and secure transport," RFC 9000, May 2021. [Online]. Available: https://rfc-editor.org/rfc/rfc9000.txt
[52] L. Eggert, "Towards securing the internet of things with QUIC," EasyChair, Tech. Rep., 2020.
[53] A. Cohen, D. Malak, V. B. Bracha, and M. Médard, "Adaptive causal network coding with feedback," *IEEE Transactions on Communications*, vol. 68, no. 7, pp. 4325–4341, 2020.
[54] C. Huitema *et al.*, "Minimal implementation of the quic protocol," https://github.com/private-octopus/picoquic/blob/master/picoquic/quicctx.c, 2021, commit: 7f49f62ff7f3938eb1a0f49dfc551d7ed189454c.
[55] "Quiche," https://quiche.googlesource.com/quiche/+/refs/heads/main, file quic/tools/quic_client_base.cc, 2021, commit: 98966fd9b7183bcdb42ce78e58be40bcf6d68493.
[56] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan, "Reducing web latency: the virtue of gentle aggression," in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, 2013, pp. 159–170.
[57] L. Thomas, E. Dubois, N. Kuhn, and E. Lochin, "Google QUIC performance over a public SATCOM access," *International Journal of Satellite Communications and Networking*, vol. 37, no. 6, pp. 601–611, 2019.
[58] N. Kuhn, G. Fairhurst, J. Border, and S. Emile, "Quic for satcom," Working Draft, IETF Secretariat, Internet-Draft draft-kuhn-quic-4-sat-03, January 2020, http://www.ietf.org/internet-drafts/draft-kuhn-quic-4-sat-03.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-kuhn-quic-4-sat-03.txt
[59] T. S. Rappaport, S. Sun, R. Mayzus, H. Zhao, Y. Azar, K. Wang, G. N. Wong, J. K. Schulz, M. Samimi, and F. Gutierrez, "Millimeter wave mobile communications for 5G cellular: It will work!" *IEEE access*, vol. 1, pp. 335–349, 2013.
[60] M. Agiwal, A. Roy, and N. Saxena, "Next generation 5G wireless networks: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 3, pp. 1617–1655, 2016.
[61] 3GPP, "Release description; Release 15," 3rd Generation Partnership Project (3GPP), Technical Report (TR) 21.915, 10 2019, version 15.0.0. [Online]. Available: http://www.3gpp.org/DynaReport/21915.htm
[62] P. Kumar and B. Dezfouli, "Implementation and analysis of QUIC for MQTT," *Computer Networks*, vol. 150, pp. 28–45, 2019.
[63] R. Puri, K. Ramchandran, K.-W. Lee, and V. Bharghavan, "Forward error correction (FEC) codes based multiple description coding for internet video streaming and multicast," *Signal Processing: Image Communication*, vol. 16, no. 8, pp. 745–762, 2001.
[64] E. O. Elliott, "Estimates of error rates for codes on burst-noise channels," *The Bell System Technical Journal*, vol. 42, no. 5, pp. 1977–1997, 1963.
[65] "Pluginized QUIC," https://github.com/p-quic/pquic, commit: 68e61c5496d8d3ef9b39e7bd5d60a14b9789e977.
[66] C. Huitema *et al.*, "Minimal implementation of the QUIC protocol," https://github.com/private-octopus/picoquic, 2020.
[67] G. F. Riley and T. R. Henderson, "The ns-3 network simulator," in *Modeling and tools for network simulation*. Springer, 2010, pp. 15–34.
[68] D. Camara, H. Tazaki, E. Mancini, T. Turletti, W. Dabbous, and M. Lacage, "DCE: Test the real code of your protocols and applications over simulated networks," *IEEE Communications Magazine*, vol. 52, no. 3, pp. 104–110, 2014.
[69] A. Gurtov, T. Henderson, S. Floyd, and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm," RFC 6582, Apr. 2012. [Online]. Available: https://rfc-editor.org/rfc/rfc6582.txt
[70] S. Ha, I. Rhee, and L. Xu, "CUBIC: a new TCP-friendly high-speed TCP variant," *ACM SIGOPS operating systems review*, vol. 42, no. 5, pp. 64–74, 2008.
[71] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "Bbr: Congestion-based congestion control," *Queue*, vol. 14, no. 5, pp. 20–53, 2016.
[72] G. Carlucci, L. De Cicco, S. Holmer, and S. Mascolo, "Analysis and design of the google congestion control for web real-time communication (WebRTC)," in *Proceedings of the 7th International Conference on Multimedia Systems*, 2016, pp. 1–12.
[73] L. S. Brakmo and L. L. Peterson, "TCP Vegas: End to end congestion avoidance on a global internet," *IEEE Journal on selected Areas in communications*, vol. 13, no. 8, pp. 1465–1480, 1995.
[74] J. F. Box, "RA Fisher and the design of experiments, 1922–1926," *The American Statistician*, vol. 34, no. 1, pp. 1–7, 1980.
[75] 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*, vol. 113, pp. 26–31, 2012.
[76] J. P. Rula, J. Newman, F. E. Bustamante, A. M. Kakhki, and D. Choffnes, "Mile high WiFi: A first look at in-flight internet connectivity," in *Proceedings of the 2018 World Wide Web Conference*, 2018, pp. 1449–1458.
[77] A. Frömmgen, J. Heuschkel, and B. Koldehofe, "Multipath TCP scheduling for thin streams: Active probing and one-way delay-awareness," in *2018 IEEE International Conference on Communications (ICC)*. IEEE, 2018, pp. 1–7.
[78] C. Huitema, "Quic Timestamps For Measuring One-Way Delays," Internet Engineering Task Force, Internet-Draft draft-huitema-quic-ts-06, Sep. 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-huitema-quic-ts-06
[79] "Tixeo, secure video conferencing," https://www.tixeo.com/, 2022.