rQUIC: Integrating FEC with QUIC for Robust Wireless Communications

Pablo Garrido*, Isabel Sánchez†, Simone Ferlin‡, Ramón Agüero§ and Özgü Alay¶
*Cybersecurity IoT IK-Ikerlan, pgarrido@ikerlan.es
†Specure GmbH, isabel.sanchez@martes-specure.com
‡Ericsson Research, simone.ferlin@ericsson.com
§Dept. of Communications Engineering, University of Cantabria, ramon@tlmat.unican.es
¶Mobile Systems and Analytics, SIMULA Metropolitan, ozgu@simula.no

***Abstract*—QUIC, fostered by Google and under standardization in the IETF, integrates some of HTTP/s, TLS, and TCP functionalities over UDP. One of its main goals is to facilitate transport protocol design, with fast evolution and innovation. However, congestion control in QUIC is still severely jeopardized by packet losses, despite implemented loss recovery mechanisms, whose behavior strongly depends on the Round Trip Time. In this paper, we design and implement rQUIC, a framework that enables FEC within QUIC protocol to improve its performance over wireless networks. The main idea behind rQUIC is to reduce QUIC's loss recovery time by making it robust to erasures over wireless networks, as compared to traditional transport protocol loss detection and recovery mechanisms. We evaluate the performance of our solution by means of extensive simulations over different type of wireless networks and for different applications. For LTE and Wifi networks, our results illustrate significant gains of up to 60% and 25% savings in the completion time for bulk transfer and web browsing, respectively.**

I. Introduction

Quick UDP Internet Connections (QUIC) [1] is an experimental transport protocol designed to primarily reduce connection establishment and transport latencies, as well as to improve security standards with default end-to-end encryption in HTTP-based applications. More specifically, QUIC [2] emerged from the urgent need of innovation in the transport layer, mainly due to difficulties extending TCP [3] and deploying new protocols. Thus, QUIC's development had to take a different approach to avoid imminent *ossification*, i.e. inability to deploy updates and fixes. These considerations already resulted in QUIC's rapid uptake. Since early large-scale experiments in 2016 [4], [5], QUIC's traffic share already reached over 7% of a European Tier1-ISP [5] and it constitutes more than 30% of Google's egress traffic [4].

Dealing with the lossy characteristics of wireless networks has proven to be very challenging for transport protocols. Forward Error Correction (FEC) is a well-known technique to improve reliability in networks that do not guarantee packet delivery [6], and it has been widely applied in the lower layers of the network stack. Due to its proven success in the lower layers, there has been interest to experiment with FEC in the transport layer. FEC has a long history of being used over unreliable transport protocols such as UDP, to protect flows [7] or to deliver different types of traffic, from bulk to time-sensitive [8]. To keep the trade-off between reliability and overhead, adaptive FEC has shown to keep latency constraints and maintain reliability as network conditions change. For TCP, FEC is mostly used to avoid retransmissions and timeouts by introducing redundancy. Different schemes in this direction have adapted simple duplicate of packets such as TCP-Tail Loss Probe (TLP) [9], non-adaptive XOR based FEC transmission such as TCP-Instant Recovery (IR) [10] and its adaptive extension [11].

Adapting most of its loss detection and recovery features from TCP, QUIC faces similar challenges in wireless networks. Therefore, one of QUIC's key goals, defined at the IETF QUIC's working group,[1] is to provide FEC support. Given the importance and the continuity of this topic, in this paper, we present rQUIC framework that integrates FEC within QUIC. Our key contributions can be summarized as:

- We introduce rQUIC, which integrates FEC in QUIC for robust wireless communications. We use an adaptive XOR-based FEC algorithm, and our design aims to be as transparent to QUIC as possible, to allow for easy integration of other FEC algorithms with minimal effort.
- We carried out an extensive simulation campaign, where we examine the benefits of rQUIC compared to default QUIC, under different network configurations, changing bandwidth, end-to-end delay, and random loss parameters. The evaluations are performed for bulk transfer and web browsing.
- We setup a wireless testbed to complement our emulations with real-world experiments and run performance analysis of rQUIC in WiFi (IEEE802.11) and LTE networks.
- To promote research reproducibility and further improvements of the algorithm, the rQUIC source code, based on a QUIC implementation in `go` [21], is made publicly available.[2]

The paper is structured as follows: Section II discusses the related work. Section III depicts rQUIC, explaining our design and implementation. Section IV presents the experimental setup and performance evaluation. Finally, Section V concludes the paper and outlooks our future work.

[1]https://quicwg.org
[2]https://github.com/pgOrtiz90/quic-go-fec/tree/quic-fec

II. Background and Related Work

Implemented in user space, QUIC [2] runs encapsulated inside UDP and it is inspired by best practices of several protocols and extensions such as TCP, TLS 1.3 and HTTP/2. QUIC aims to reduce connection latency with the possibility to send data directly at the connection setup. Internally, QUIC packets have an unencrypted public header and an encrypted payload, which can include one or more frames. Frames in the same packet may carry control or data information, and might belong to the same or different streams, i.e. stream multiplexing. In case of packet loss, only streams with frames in the lost packets are blocked, leaving others unaffected. This is one of the main benefits of QUIC, as it avoids HoL-blocking with multiple streams. However, if the lost packet contains frames from multiple streams, it is evident that a mechanism such as the one we propose can help reducing latency associated to retransmissions. Packet numbers in QUIC are monotonically increasing. Every transmitted packet, including retransmissions, carries a different number, avoiding TCP's retransmission ambiguity and so simplifying loss detection. Another fundamental difference lies in the Acknowledgment (ACK) frames, which carry multiple ACK blocks and information to yield a more precise RTT estimation. Even though QUIC runs encapsulated inside UDP, all data is reliably transmitted due to the stream and connection flow control [12], and congestion control [13]. The congestion control relies on Cubic,[3] with pacing applied at the sender.

Due to its rising popularity, there is interest to assess QUIC's performance and traffic share beyond Google's perspective. One of the initial works [14] compares QUIC, HTTP and SPDY without finding a clear winner, but stating that network conditions actually determine the protocol performance. This variability motivates an adaptive algorithm to increase robustness, as the one we propose herewith. Recently, the authors of [15] focus on a more controlled understanding of QUIC's performance following its rapid updates, mainly driven by gQUIC Chrome implementation. Rüth *et al.* [5] focus on finding QUIC's traffic share and which infrastructures, e.g. ISPs, Alexa Top 1M, already support QUIC in the Internet. Motivated by the boost of adaptive streaming, Bhat *et al.* [16] analyse the benefits QUIC can bring to adaptive video streaming (DASH), and evaluate its impact on the QoE.[4]

Google advocated for the need of loss recovery with FEC in QUIC [17] very early. However, after discouraging results with a closed source implementation, they started with a new approach [18], and FEC is now an IETF agenda item. Our analysis and evaluations indicate that rQUIC can improve latency and, due to its adaptability, not hinder QUIC's performance. Other proposals in the Network Coding Research Group (NWCRG) [19] list very high-level requirements for network and network-level packet erasure coding in QUIC. We adopt some of their recommendations alongside with [11] for rQUIC. Recently, Michel *et al.* have integrated FEC functionality into QUIC [20], but they did not consider an adaptation scheme as the one promoted by rQUIC. The obtained results show that for large size files, the performance might not be as good as expected. Last, this work is an extended version of our previous short paper [**?**].

[3]Latest drafts suggest a change to NewReno [13].

[4]Google serves YouTube's content over QUIC since 2016. [4]

III. rQUIC: Design and Implementation

In the following, we introduce rQUIC's framework design and integration into QUIC in Section III-A. Then, we describe rQUIC's implementation in detail in Section III-B, discussing the limitations in Section III-C.

A. rQUIC Design

We design rQUIC as a framework to enable FEC within QUIC. Figure 1 shows rQUIC's building blocks, focusing on the modifications introduced by rQUIC and its interaction with QUIC. A QUIC session represents a unique end-to-end connection. If both congestion and flow control allow the transmission of stream data (bundled in `STREAM` frames), the QUIC session will then generate QUIC packets, which are encrypted and authenticated.

The encrypted QUIC packet is then input to rQUIC module, which builds the rQUIC packet by *(i)* adding the `FEC Header` field to the QUIC header (see Figure 1), and *(ii)* applying the corresponding FEC algorithm depending on the packet type.

rQUIC header: As we can choose to enable or disable rQUIC in the QUIC session, the presence of the `FEC Header` field can be indicated through flags in the QUIC header or negotiated in the connection establishment. A `FEC header` is 4-Byte long including the following fields:

- Type (1B): identifies whether the packet is protected by FEC or not, or if it is a FEC packet itself. This field can take different values: 0x00 represents packets that are not protected, such as `ACK` or `STOP_WAITING` frames, which are transmitted periodically, 0x80 represents a protected packet, 0xC0 represents a FEC packet.
- Block ID (1B): identifies packets protected by the same FEC packet, i.e. packets that belong to the same block.
- FEC Ratio (1B): ratio used by the FEC algorithm.
- Count (1B): If protected, this field identifies the order of the packet in the FEC block.

rQUIC packets (protected, unprotected or FEC) are then sent to the receiver respecting flow and congestion control. At the receiver side, FEC packets will be used to recover erasures while QUIC packets are forwarded to the QUIC session.

FEC Algorithm: One of the challenges introducing *sophisticated* FEC algorithms to QUIC is the potential latency for encoding and decoding [19], which goes against QUIC's latency improvements and may conflict with power-constrained devices. Hence, a lightweight and efficient algorithm could help with some initial guidelines for future research. Further, the design of a FEC algorithm must adapt to the link characteristics, providing a trade-off with bandwidth consumption due to the introduced overhead. For these reasons, for this

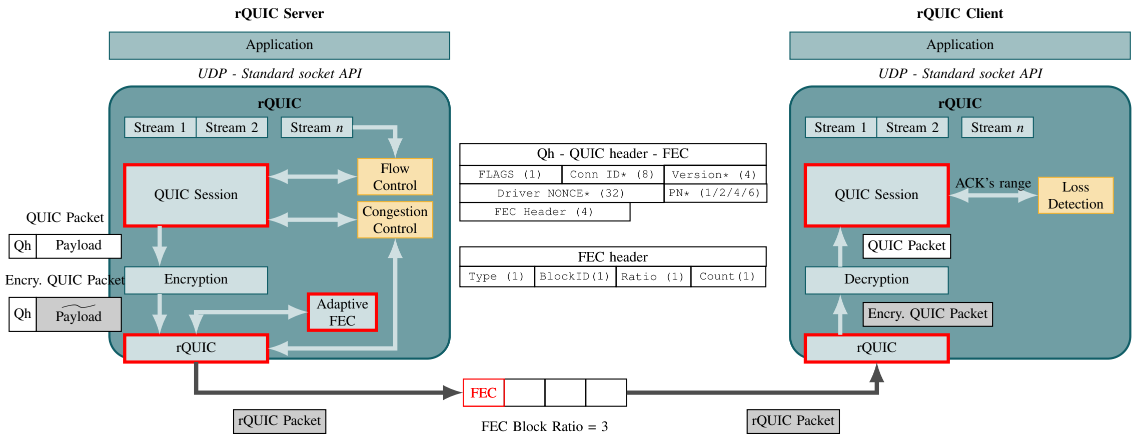


Fig. 1. rQUIC's framework: data flow from application to network layer.

initial implementation of FEC, we choose an adaptive FEC algorithm based on XOR, which we explain next.

1) XOR-based FEC: XOR-based FEC only introduces $n+1$ redundancy, i.e. for every n data packets there is one extra packet to protect the data block, which is the XOR of the n data packets. XOR is known to have low computational complexity but, on the other hand, such a simple approach only allows to recover one packet per block. In the case of multiple losses, the XOR FEC will not prevent the classic loss recovery mechanism with retransmissions,[5] but would consume resources. Consequently, the size of the FEC block is a key optimization parameter that can significantly impair the overall performance. That is why we adopt an adaptive algorithm that varies the FEC ratio according to changes in the retransmissions that are needed.

2) Adaptive FEC: Losses in wireless channels are difficult to predict and they may vary over time. Therefore, the FEC block size should take this into account, providing gains over transmissions without FEC. For this reason, we follow an *adaptive* FEC approach that reduces the overhead in the absence of losses and increases the redundancy otherwise. The proposed algorithm is based on steering *residual losses*, which are packets that need to be retransmitted due to FEC failing to recover.

Hence, the adaptive algorithm counts the total transmitted and retransmitted packets over a period, i, of length T and computes the residual loss as:

$$\epsilon_i = \frac{\text{retransmissions}}{\text{transmissions} - \text{retransmissions}} \tag{1}$$

Measurements of residual losses are then averaged over N periods:

$$\overline{\epsilon}_i = \frac{\sum_{i=1}^{N} \epsilon_i}{N} \tag{2}$$

The FEC ratio is then updated as follows: if the average residual loss is higher than a target value γ, the FEC ratio is increased by δ, and decreased by δ otherwise. Both δ and γ

[5]To date the default algorithms used in QUIC, NewReno and Cubic, take loss as congestion input signal.

Algorithm 1 Adaptive FEC in rQUIC

```
$r = r_{\text{init}}$
**if** $\overline{\epsilon} > \gamma$ **then**
    $r = r \times (1 - \delta)$
**else**
    $r = r \times (1 + \delta)$
**end if**
```

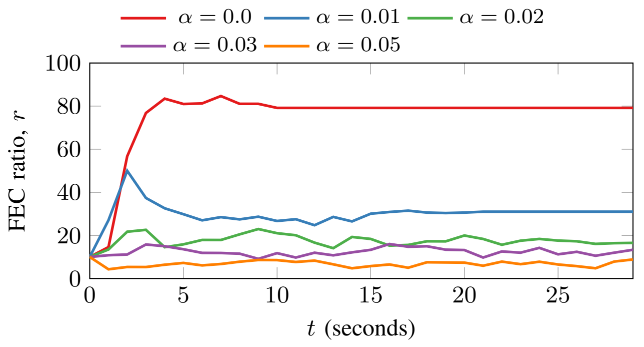


Fig. 2. Evolution of rQUIC's adaptive FEC ratio over time, for different link loss rates with e.g. no injected loss (only congestion) 0, 1, 2, 3 and 5%.

can be seen as the aggressiveness parameters of the algorithm. These are however configurable and determine the tolerance to FEC recovery failure. In preliminary experiments, we choose $T = 3 \cdot RTT$ as the measurement period for the residual value, with the assumption that if a loss occurs during one RTT, a retransmission will take place in the second and, possibly, be concluded in the third one. The proposed algorithm (see Algorithm 1) will increase quickly the FEC ratio on links with a small error rate, while on high-loss links the FEC ratio will oscillate between low values. To illustrate rQUIC's adaptivity, Figure 2 shows the FEC ratio evolution for different link loss rates α for a 30-second bulk transfer over a channel with 20Mbps and 25ms RTT and an initial ratio of $r_{\text{init}} = 10$. One can see that the FEC ratio quickly converges from the initial value to minimize residual losses.

In order to avoid RTOs in tail losses of short flows, TLP is now also part of QUIC [13], [21]. However, we decided to exclude TLP of rQUIC, as we aim to recover losses with FEC and avoid retransmissions of packets in any position in a flow, and not only account for tail losses.

B. *rQUIC Implementation*

We implement rQUIC based on an open source implementation of QUIC in `go` [21] (v0.7.0), keeping its default settings, e.g. Cubic is used as congestion control. However, `quic-go`'s implementation follows the IETF QUIC specification, which is heavily under development. Thus, we had to opt for the latest stable release available (v0.7.0) that may not incorporate the most recent updates. Also, compatibility across `quic-go` versions cannot be guaranteed, as the codebase is constantly refactored.[6]

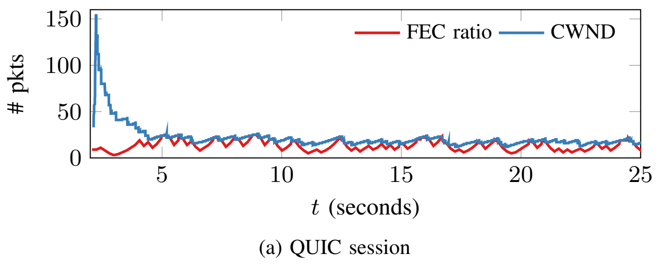
We keep rQUIC's implementation transparent to the other modules inside QUIC. We highlight in Figure 1 the main modifications introduced by rQUIC as red blocks. Since QUIC's loss recovery mechanism is done at the packet level, we opt to follow the same strategy. The rQUIC block takes as input the encrypted QUIC packets, adds the FEC Header field, and, if the packet contains `STREAM` frames, it applies the corresponding FEC algorithm to the encrypted payload. Since FEC packets count towards the CWND size, rQUIC also interacts with the congestion control, obeying the allowed sending rate. Complying with the congestion control impacts the implementation, as we explain in Sec. III-C.

On the receiver side, the FEC decoder parses all the received protected packets. If one is lost, the following received packets from the same Block ID will be held by the decoder until the following events: another packet is lost in the same FEC block, there are no further losses in the FEC block and FEC packet arrives or packet from a different FEC block arrives. Afterwards, all held packets are forwarded to the upper layer, including the recovered packet if possible.

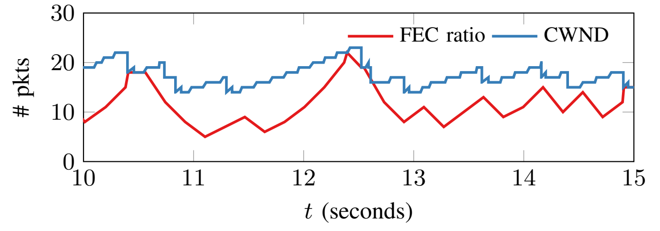
Recovering the original QUIC header is very critical, with special attention needed to the Packet Number (PN). As mentioned before, QUIC's payload is encrypted and its header authenticated. This also includes the PN of the recovered packet, which is unique for every packet and can be randomly skipped by the sender [22]. We protect encrypted QUIC payload with FEC, but we also need to be able to recover the header of the lost packet. To accurately recover the PN of an erasure, we set the last 6 bits of the FEC type field in each packet to the offset with respect to the PN in the previous protected packet.

The loss detection mechanism counts the different PN's received, and it would not see that a packet is missing if the lost packet is recovered by rQUIC. Such detection mechanism generates QUIC acknowledgments, notifying the transmitter the range of PN already received. A FEC packet is seen by the lost detection and congestion control mechanisms as a regular QUIC packet, increasing or reducing the CWND if it is acknowledged or lost. However, if a FEC packet is lost it will not be retransmitted, but it will generate a CWND reduction.

[6]`quic-go` v0.8.0 has recently been released introducing new features following the guidelines of the IETF QUIC working group [22].



(a) QUIC session



(b) Zoom $t \in \{10, 15\}$

Fig. 3. Adaptive FEC and CWND over time for a 3% error rate.

C. *Implementation Limitations and Corner Cases*

We discuss some limitations and corner cases to provide guidelines for other FEC algorithms in QUIC.

FEC Ratio: The FEC Ratio value must lie between 2 and 255. In the former case it implies the transmission of only coded packets, whereas in the latter it is limited by the FEC Ratio header field of 1 Byte. Note that when the CWND reaches such small values, this is not a limitation of rQUIC, but an unstable behavior caused by the connection oscillating between congestion avoidance and slow-start, which depends on the congestion control policy.[7] Further, the FEC ratio cannot exceed the CWND, preventing the congestion control from blocking the transmission in the middle of a FEC block.

Figure 3 shows both the FEC ratio and the CWND evolution for 25s and how FEC ratio adapts well to the changing conditions.

Slow-start: It is important to highlight two characteristics of the congestion control in `quic-go`: The Initial Window (IW) has 32 packets, and the `ssThreshold` is not set, i.e. the slow-start phase lasts until the first loss occurs. In our experiments, we observe a spike at the beginning of the QUIC connection, therefore observing poorer adaptation performance in the first 5s, which can limit the gains of rQUIC for web transfers with small objects. To address this issue, one modification to rQUIC is to disable the FEC in slow-start and activate it through the connection when needed. We leave this analysis and optimization as part of our future work.

Out-of-order: If a packet from the next FEC block arrives before all packets from a previous one, a decoding failure is assumed and all held packets are forwarded to the QUIC session. However, packets from the previous block could also be out-of-order. For future work, we plan on avoiding this decoding failures by working with a timer at the receiver.

[7]https://github.com/lucas-clemente/quic-go/blob/master/internal/congestion/cubic.go

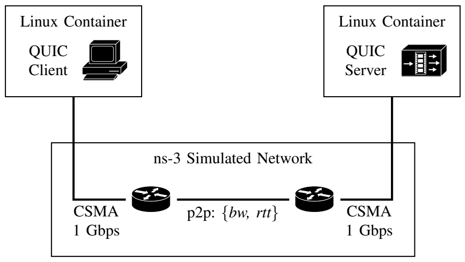


Fig. 4. Emulation scenario

TABLE I
NETWORK PARAMETER RANGES FOR DIFFERENT WIRELESS TECHNOLOGIES

	NetType1		**NetType2**	**NetType3**
	WLAN	**4G**	**2G/3G**	**Satellite**
Capacity [Mbps]	20-30	15-40	5-10	1-2
RTT [ms]	20-30	25-40	75-100	350-700
Loss [%]	0-5	0-1	0-1	5-10

IV. Evaluation and Results

A. Experimental Setup

For the simulations, we use the NS-3[8] discrete-event network simulator. With NS-3, we are able to connect real application traffic over a simulated network, in this case rQUIC client and server, as described in Section III. The emulated environment (see Figure 4) consists of client and server applications running in Linux containers hosted in the same machine and connected through a simulated network. The containers are seen by the NS-3 as *ghost* nodes,[9] each of which is connected through a CSMA network to a node, which is the router. Then, routers are connected with a point-to-point link, which can be configured with different combinations of bandwidth, delay and loss rates, mimicking various network technologies. Following Table I, we vary the parameters on the link, so that three Network Types (NetTypes) with different characteristics are covered. We then vary the loss rates between 0-5% in all configurations. Although there exists evidence that cellular networks maintain buffer sizes larger than the path's Bandwidth Delay Product (BDP), the so-called *bufferbloat* [23], we adjust all buffers to be one BDP.

We perform experiments with bulk and web traffic, using HTTP/2, and we compare the performance of default QUIC with our proposed rQUIC scheme.With bulk transfers, the client downloads a large file (20MB for *NetType1* and *NetType2* and 5MB for *NetType3*) from the server. In the case of the web transfer, we select www.flickr.com, which comprises 30 objects, with an overall size of $1,776$ KiB. We also considered other websites with different combination of objects and sizes, but due to lack of space and the lack of noteworthy differences, we just show the results observed for www.flickr.com. Since rQUIC builds upon the `quic-go` project, we cannot use a web browser to download the websites. Instead, we follow the Epload project,[10] which downloads the websites and creates a dependency graph. We implement an application in `go` (integrated in our rQUIC client) that reads these dependencies and generates the corresponding HTTP requests.

In order to measure the benefits of rQUIC, we consider the following metrics: Completion time ratio (ξ) and Overhead.

$$\xi = \frac{\overline{\text{Completion Time rQUIC}}}{\overline{\text{Completion Time QUIC}}} \tag{3}$$

where $\overline{\text{Completion Time}}$ is the average completion time, i.e the total time required to complete a download after 100 iterations. Hence, $\xi \leq 1$, represents the performance gain rQUIC yields over legacy QUIC. The overhead represents the redundant data (FEC packets) transmitted in a rQUIC session, as a percentage of the total amount of data transmitted in the session.

B. Performance Evaluation

In this section, we evaluate the performance of rQUIC in the network configurations shown in Table I. First, we carry out extensive simulations and analyze the performance of rQUIC under different network conditions for bulk transfer and web traffic. Before running the simulations, we conducted sensitivity analysis for both tolerance and correction ratio to understand their impact. These results are not presented due to lack of space. With respect to tolerance value, although the completion time ratio does not significantly vary, we observed higher download transfer times for higher tolerance values. With respect to the correction rate, the completion time ratio does not dramatically change, but we observe lower download transfer times as we increase the correction rate. In order to keep rQUIC generic and adaptive to an unpredictable amount of random losses, we choose $N = 3$, $\delta = 0.33$ and $\gamma = 1\%$. Unless otherwise stated, we use these values for the rest of our experimental evaluation.

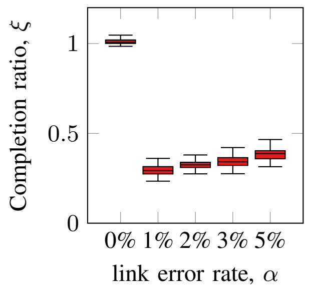
First, we focus on the performance of rQUIC for bulk transfer over different emulated channels, shown in Table I. In Fig. 5 we illustrate the completion time ratio, as well as the overhead caused by FEC packets (Fig. 5d). We observe that regardless the RTTs, dynamically adjusting FEC to the link characteristics, yields a clear benefit, up to 60% over *NetType1* and 50% over *NetType3*. Moreover, the overhead gets higher as we increase the link erasure ratio, being particularly relevant with satellite communications. It is worth highlighting that with very long delay, 400ms, our adaptive algorithm shows a different pattern compared to the other two cases, which implies that our adaptive algorithm does not react quickly enough under these circumstances. With 0% injected loss[11] rQUIC does not bring any benefit, as it was expected, but the performance is neither jeopardized. On the other hand, the FEC overhead increases as the link error rate gets higher, as expected from the adaptiveness of the FEC algorithm. Note that with 0% injected losses, the overhead is below 5%, regardless the RTT.

[8]https://www.nsnam.org

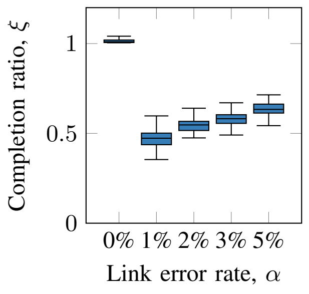
[9]In NS-3, only CSMA and WiFi modules can be connected to *ghost* nodes.

[10]http://wprof.cs.washington.edu/spdy/tool

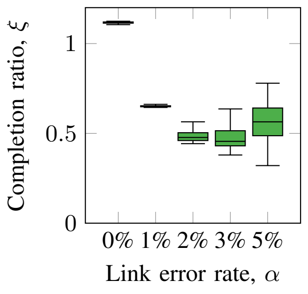
[11]Under such circumstance there exist losses due to network congestion.



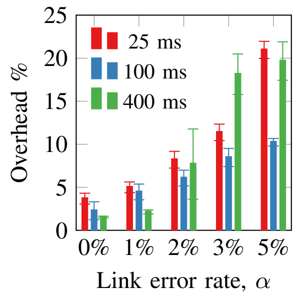
(a) NetType1 (25ms,20Mbps)



(b) NetType2 (100ms,10Mbps)

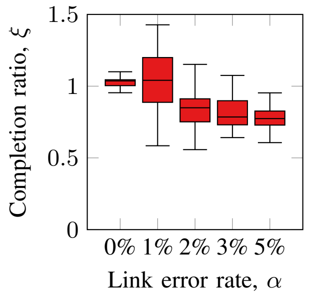


(c) NetType3 (400ms,1.5Mbps)

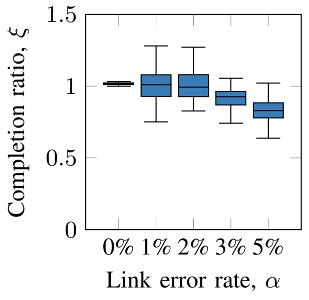


(d) Overhead

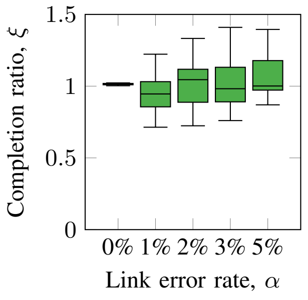
Fig. 5. Completion time ratio for the three scenarios under different amount of random losses for the bulk transfer experiments.



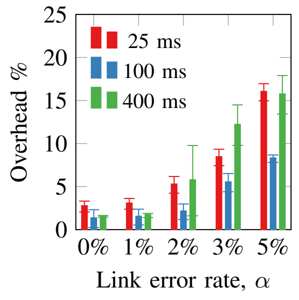
(a) NetType1 (25ms,20Mbps)



(b) NetType2 (100ms,10Mbps)



(c) NetType3 (400ms,1.5Mbps)



(d) Overhead

Fig. 6. Completion time ratio for the 3 scenarios under different amount of random losses for the HTTP/2 transfer (flickr.com).

In Figure 6, we present the results obtained through simulation with web traffic, which is characterized by the transmission of several objects of small size. We observe that rQUIC yields better performance over lossy networks, specially over *NetType1* and *NetType2*. With long delay networks, the benefits brought by FEC are not clear, mainly due to the poor response during the slow start phase of the congestion control. In our future work, we plan to disable FEC operation during slow start, where the dynamic algorithm is not able to correctly adapt to the channel losses, because the congestion window increases quickly until the first loss is seen by the transmitter. On the other hand, the overhead produced by rQUIC follows the same pattern as the one seen with bulk traffic, but with lower values. This is due to the fact that most part of the web transmission is carried out during the slow start phase, where the FEC ratio quickly increases, not seeing any loss.

We have run further experiments with other websites including www.google.com and www.huffingtonpost.com, with 6 and 110 objects respectively. In both cases we saw that rQUIC yielded a slight performance improvement. We note that the gains decrease for websites with small objects since transmission of these objects mostly occurs during slow start. Results are not discussed due to space constraints.

To complement our simulations with real-world experiments we also analyzed the performance of rQUIC over real WiFi (IEEE802.11g) and LTE networks. We used the testbed shown in Figure 7. Figure 8a shows the completion time ratio results of the bulk over the WLAN and LTE connections in our testbed. For WLAN, we observe an average gain of 20%. Similar to bulk transfer, the results for web traffic, illustrated in Figure 8b, do not show significant gains for LTE. However, rQUIC yields a 10% gain on average over WLAN. Note that these gains are lower than the simulation results, since the average losses we observed during the experiments were lower than those considered during the simulations (median loss was observed to be less than 0.09%). Note that in order to establish a baseline, we have tested an optimistic scenario in this case, as there was just one client connected to the Access Point. We expect that the error rate would be higher in a WLAN if several devices were connected, due to higher level of interference and collisions. Under these circumstances rQUIC is expected to yield stronger benefits, as was shown in the simulation. On the other hand, with LTE, the average number of losses is close to 0 and, therefore, the adaptive FEC algorithm can not bring strong gains. However, LTE networks are known to be lossy under mobility scenarios where rQUIC benefits will be much higher. We plan to extend our experimental study in different WLAN settings and a more extensive measurement campaign over LTE networks, specially under mobility.

V. Conclusions

The performance of transport protocols over wireless networks is known to be suboptimal, with one of the main limiting factors being the loss recovery time. In this paper, we introduced rQUIC framework, which enables FEC in QUIC for robust wireless communications. To illustrate the benefits of rQUIC, we presented the design and open source implementation of an adaptive XOR-based FEC algorithm,

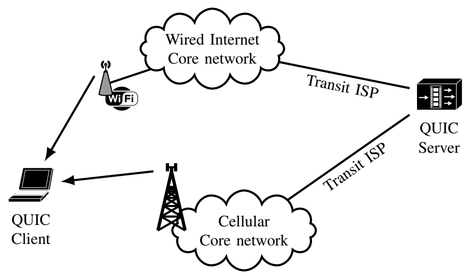


Fig. 7. Wireless testbed scenario.

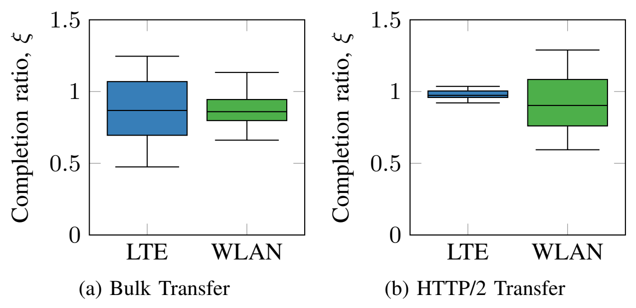


Fig. 8. Bulk and web transfer experiment over wireless networks

and we integrated it with QUIC. However, our design is transparent to QUIC, thus allowing easy integration of other FEC algorithms with minimal effort. We then evaluate the performance of rQUIC compared to legacy QUIC under different network scenarios for bulk and web transfers, by means of an extensive simulation-based analysis including simulation and real network experiments. Our simulation results show that for typical wireless networks (WiFi and LTE), rQUIC yields to significant gains of up to 60% and 25% savings in the completion time for bulk transfer and web browsing, respectively (Figs 5a and 6a). Experiments carried out over real testbeds showed lower gains, since the loss rate was lower, but the proposed scheme still outperforms legacy QUIC.

In our future work, we plan to focus on addressing the limitations and corner cases discussed previously. Furthermore, we plan to extend our work by implementing other FEC mechanisms in rQUIC and evaluating the complexity and gain trade-offs for these mechanisms. Finally, considering the most dominant traffic in Internet is video traffic, we plan to analyze rQUIC performance for video streaming applications.

Acknowledgment

Özgü Alay was partially supported the Norwegian Research Council project No. 250679 (MEMBRANE). Ramón Agüero was partially supported by the Spanish Government (MINECO, MCIU, AEI, FEDER) by means of the projects ADVICE: Dynamic provisioning of connectivity in high density 5G wireless scenarios (TEC2015-71329-C2-1-R) and FIERCE: Future Internet Enabled Resilient Cities (RTI2018-093475-A-100).

References

[1] J. Roskind, “QUIC (Quick UDP Internet Connections): Multiplexed Stream Transport Over UDP,” *Technical report, Google*, 2013.

[2] Y. Cui, T. Li, C. Liu, X. Wang, and M. Kühlewind, “Innovating Transport with QUIC: Design Approaches and Research Challenges,” *IEEE Internet Computing*, vol. 21, no. 2, pp. 72–76, Mar 2017.

[3] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda, “Is It Still Possible to Extend TCP?” in *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement*, ser. IMC ’11. ACM, 2011, pp. 181–194.

[4] Langley, A. et al., “The QUIC Transport Protocol: Design and Internet-Scale Deployment,” in *Proceedings of the ACM SIGCOMM*. ACM, 2017, pp. 183–196.

[5] J. Rüth, I. Poese, C. Dietzel, and O. Hohlfeld, “A First Look at QUIC in the Wild,” in *International Conference on Passive and Active Network Measurement*. Springer, 2018, pp. 255–268.

[6] M. Watson, M. Luby, and L. Vicisano, “Forward error correction (fec) building block,” Internet Requests for Comments, RFC Editor, RFC 5052, August 2007.

[7] M. Watson, A. Begen, and V. Roca, “Forward Error Correction (FEC) Framework,” RFC 6363 (Proposed Standard), Internet Engineering Task Force, Oct. 2011.

[8] C. Zhang, C. Huang, P. A. Chou, J. Li, S. Mehrotra, K. W. Ross, H. Chen, F. Livni, and J. Thaler, “Pangolin: Speeding up concurrent messaging for cloud-based social gaming,” in *Proceedings of CoNEXT ’11*. ACM, 2011, pp. 23:1–23:12.

[9] N. Dukkipati, N. Cardwell, Y. Cheng, and M. Mathis, “Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses,” IETF, Internet-Draft draft-dukkipati-tcpm-tcp-loss-probe-01, Oct. 2015, work in Progress.

[10] T. Flach, N. Dukkipati, Y. Cheng, and B. Raghavan, “TCP Instant Recovery: Incorporating Forward Error Correction in TCP,” Internet Engineering Task Force, Internet-Draft draft-flach-tcpm-fec-00, work in Progress.

[11] S. Ferlin and Ö. Alay, “TCP with dynamic FEC for high delay and lossy networks,” in *ACM CoNEXT Student Workshop*, 2016.

[12] J. Iyengar and M. Thomson, “QUIC: A UDP-based multiplexed and secure transport,” *draft-ietf-quic-transport-01 (work in progress)*, 2017.

[13] J. Iyengar and I. Swett, “QUIC Loss Detection and Congestion Control,” Working Draft, IETF Secretariat, Internet-Draft draft-ietf-quic-recovery-00, November 2016.

[14] P. Megyesi, Z. Krämer, and S. Molnár, “How quick is QUIC?” in *2016 IEEE International Conference on Communications (ICC)*, May 2016, pp. 1–6.

[15] 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 IMC*. ACM, 2017, pp. 290–303.

[16] D. Bhat, A. Rizk, and M. Zink, “Not so QUIC: A performance study of DASH over QUIC,” in *Proceedings of the 27th Workshop on Network and Operating Systems Support for Digital Audio and Video*. ACM, 2017, pp. 13–18.

[17] J. Roskind, “Quick UDP internet connections: Multiplexed stream transport over UDP,” 2013. [Online]. Available: https://www.ietf.org/proceedings/88/slides/slides-88-tsvarea-10.pdf

[18] I. Swett, “QUIC FEC v1,” https://docs.google.com/document/d/1Hg1SaLEl6T4rEU9j-isovCo8VEjjnuCPTcLNJewj7Nk/edit, 2016.

[19] I. Swett, M.-J. Montpetit, and V. Roca, “Coding for QUIC,” Working Draft, IETF Secretariat, Internet-Draft draft-swett-nwcrg-coding-for-quic-00, March 2018, http://www.ietf.org/internet-drafts/draft-swett-nwcrg-coding-for-quic-00.txt.

[20] F. Michel, Q. De-Coninck, and O. Bonaventure, “QUIC-FEC: Bringing the benefits of Forward Erasure Correction to QUIC,” in *IFIP Networking 2019 Conference*, May 2019.

[21] M. S. Lucas Clemente, “A QUIC implementation in pure go,” https://github.com/lucas-clemente, 2013.

[22] J. Iyengar and M. Thomson, “QUIC: A UDP-Based Multiplexed and Secure Transport,” Standards Track, IETF, Internet-Draft draft-ietf-quic-transport-13, June 2018.

[23] H. Jiang, Y. Wang, K. Lee, and I. Rhee, “Tackling bufferbloat in 3G/4G networks,” in *Proceedings of the 2012 ACM conference on Internet measurement conference*, ser. IMC ’12. ACM, 2012, pp. 329–342.