If you cite this paper, please use the ACM CoNEXT reference: Marcel Kempf, Simon Tietz, Benedikt Jaeger, Johannes Späth, Georg Carle, and Johannes Zirngibl. 2025. QUIC Steps: Evaluating Pacing Strategies in QUIC Implementations. Proc. ACM Netw. 3, CoNEXT2, Article 13 (June 2025), 14 pages. https://doi.org/10.1145/3730985

QUIC Steps: Evaluating Pacing Strategies in QUIC Implementations

MARCEL KEMPF, Technical University of Munich, Germany
SIMON TIETZ, Technical University of Munich, Germany
BENEDIKT JAEGER, Technical University of Munich, Germany
JOHANNES SPÄTH, Technical University of Munich, Germany
GEORG CARLE, Technical University of Munich, Germany
JOHANNES ZIRNGIBL, Max Planck Institute for Informatics, Germany

Pacing is a key mechanism in modern transport protocols, used to regulate packet transmission timing to minimize traffic burstiness, lower latency, and reduce packet loss. Standardized in 2021, QUIC is a UDP-based protocol designed to improve upon the TCP / TLS stack. While the QUIC protocol recommends pacing, and congestion control algorithms like BBR rely on it, the user-space nature of QUIC introduces unique challenges. These challenges include coarse-grained timers, system call overhead, and OS scheduling delays, all of which complicate precise packet pacing.

This paper investigates how pacing is implemented differently across QUIC stacks, including quiche, picoquic, and ngtcp2, and evaluates the impact of system-level features like GSO and Linux qdiscs on pacing. Using a custom measurement framework and a passive optical fiber tap, we establish a baseline with default settings and systematically explore the effects of qdiscs, hardware offloading using the ETF qdisc, and GSO on pacing precision and network performance. We also extend and evaluate a kernel patch to enable pacing of individual packets within GSO buffers, combining batching efficiency with precise pacing. Kernel-assisted and purely user-space pacing approaches are compared. We show that pacing with only user-space timers can work well, as demonstrated by picoquic with BBR.

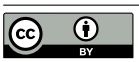
With quiche, we identify FQ as a qdisc well-suited for pacing QUIC traffic, as it is relatively easy to use and offers precise pacing based on packet timestamps. We uncovered that internal mechanisms, such as a library's spurious loss detection logic or algorithms such as HyStart++, can interfere with pacing and cause issues like unstable congestion windows and increased packet loss. Our findings provide new insights into the trade-offs involved in implementing pacing in QUIC and highlight potential optimizations for real-world applications like video streaming and video calls.

CCS Concepts: • **Networks** → **Transport protocols**; **Network measurement**.

Additional Key Words and Phrases: QUIC, Pacing, Congestion Control, GSO, Queueing Disciplines, Measurement Framework

ACM Reference Format:
Marcel Kempf, Simon Tietz, Benedikt Jaeger, Johannes Späth, Georg Carle, and Johannes Zirngibl. 2025. QUIC Steps: Evaluating Pacing Strategies in QUIC Implementations. *Proc. ACM Netw.* 3, CoNEXT2, Article 13 (June 2025), 14 pages. https://doi.org/10.1145/3730985

Authors' Contact Information: Marcel Kempf, kempfm@net.in.tum.de, Technical University of Munich, Munich, Germany; Simon Tietz, tietz@net.in.tum.de, Technical University of Munich, Munich, Germany; Benedikt Jaeger, jaeger@net.in.tum.de, Technical University of Munich, Munich, Germany; Johannes Späth, spaethj@net.in.tum.de, Technical University of Munich, Munich, Germany; Georg Carle, carle@net.in.tum.de, Technical University of Munich, Munich, Germany; Johannes Zirngibl, jzirngib@mpi-inf.mpg.de, Max Planck Institute for Informatics, Saarbrücken, Germany.



ACM 2834-5509/2025/6-ART13
https://doi.org/10.1145/3730985

1 Introduction

QUIC has rapidly gained adoption and now handles a significant share of global network traffic [8, 19, 47, 48]. It is used for applications like web browsing, video streaming, and video calls. As QUIC continues to grow further in importance, improving its performance is an active area of research.

Pacing, the controlled scheduling of packet transmissions, is a well-known mechanism for improving efficiency by trying to avoid bursty packet transmissions. It is known to reduce packet loss, lower queuing delays, and improve overall network performance by spacing out packets more evenly over time [10, 46]. The QUIC standard issues the normative statement that a sender should pace outgoing traffic with, for example, a leaky bucket algorithm [16]. As Congestion Control Algorithms (CCAs) like Bottleneck Bandwidth and RTT (BBR) depend on pacing to work, most QUIC implementations contain at least some form of pacing.

However, implementing pacing in QUIC presents specific challenges. Unlike TCP, which is paced in the kernel, QUIC runs in user-space. Therefore, QUIC stacks must implement pacing themselves, which may not be as effective as kernel-based pacing due to latency and jitter added by system calls and Operating System (OS) scheduling. Some implementations rely on kernel features, especially Linux queueing disciplines (qdiscs), to help with pacing. Additionally, techniques like Generic Segmentation Offload (GSO), which are designed to improve performance, might interfere with pacing by releasing bursts of packets together, counteracting the smoothing effect of pacing.

Since QUIC as a general purpose transport protocol is used for video streaming and video calls, understanding how to implement effective pacing is important for these real-world applications. This paper explores how pacing is implemented in different QUIC stacks, examines the impact of GSO, and investigates whether qdiscs can improve QUIC pacing.

This work covers the following contributions:

- *(i)* We evaluate the pacing behavior of QUIC libraries on the wire and find different behavior between libraries and configurations. While picoquic shows the largest bursts with loss-based CCAs, its pacing behavior using BBR outperforms other implementations without using kernel functionalities.
- *(ii)* We evaluate different mechanisms with and without kernel support, *e.g.*, Fair Queue (FQ) or GSO and their impact on the pacing behavior. Existing mechanisms offer QUIC flexibility, but an informed optimization of QUIC implementations and used mechanisms is important to reach optimal behavior for individual applications and the network in general.
- *(iii)* We extend and evaluate a kernel patch that enables pacing within GSO buffers. This allows QUIC stacks to retain the performance benefits of batching without sacrificing pacing.
- *(iv)* We provide all shown patches, configurations, measurement results, evaluation scripts, and our measurement framework to allow others to evaluate QUIC pacing in the future and optimize their applications. All data is available in a public GitHub repository [22].

We introduce important background in Section 2, followed by a description of our measurement framework, used implementations and features in Section 3. Section 4 evaluates the behavior of different QUIC libraries in combination with system functionalities. Finally, we cover related work in Section 5 and conclude our findings in Section 6.

2 Background

This section provides relevant background for QUIC and pacing, as well as the Linux kernel features that are used in this work.

Pacing denotes the even distribution of packets over a Round-Trip Time (RTT) [1]. The goal of pacing is to reduce queuing and hereby packet loss by avoiding bursty packet transmissions.

Pacing is integrated in the Linux kernel for more than a decade [12]. However, TCP traffic is only paced when either the selected congestion control algorithm performs pacing or a qdisc is used that supports pacing. For example, Debian Bookworm uses TCP CUBIC and Fair Queuing Controlled Delay (FQ_CoDel) as defaults [11], thus TCP traffic is not paced. Regarding QUIC, RFC 9002 [16] states that packet bursts have to be prevented and suggests pacing in combination with the used CCA. During the current discussions to define a variable acknowledgment (ACK) frequency [17], the suggestion to pace is further highlighted. While a smaller ACK frequency reduces the overhead for data receivers, it reduces the effectiveness of ACK-clocking and could lead to bursts if pacing is not implemented.

QUIC traffic can be paced by a qdisc, a mechanism connecting the Linux kernel's networking stack and a Network Interface Card (NIC). It queues and schedules packets for transmission. qdiscs can prioritize, delay, or drop packets allowing features like rate limiting, traffic shaping, or pacing. QUIC traffic profits from a qdisc that supports the scheduling of packets based on a timestamp, which can be passed to the kernel for every packet. Pacing approaches with other qdiscs involving rate limiting, *e.g.*, Token Bucket Filter (TBF), do not offer an interface for adjusting the pacing rate dynamically from user-space. They are therefore less interesting for the usage with QUIC, as the desired pacing rate can change continuously during a connection. We look at the FQ and Earliest TxTime First (ETF) qdiscs in Section 4, as both of them support the previously mentioned timestamp-based scheduling. By setting the `SO_TXTIME` socket option, a timestamp can be passed with each `sendmsg` call using the `SCM_TXTIME` control message header. The main difference between FQ and ETF is that ETF drops packets if their timestamp is in the past, while FQ does not. To ensure that packets are enqueued before their scheduled transmission time and thus not dropped, ETF has a *delta* parameter that specifies a time offset at which the qdisc becomes active in advance [31]. The optimal value for this parameter depends on the processing overhead and varies from system to system. However, ETF supports hardware offloading for NICs that support the so-called *LaunchTime* feature. In this case, the NIC holds back outgoing packets until the timestamp is reached, sending them out at the specified timestamp. This could further increase the precision of pacing (see Section 4.4).

To reduce the overhead of context switches between QUIC in user-space and the kernel, segmentation offloading can be used. This technique allows passing multiple packets between the kernel network stack and the user space as one data unit [35]. GSO and Generic Receive Offload can be used with QUIC and can significantly improve the throughput [21]. However, GSO prevents pacing within each batch of packets. We focus on the sending behavior and thus GSO (see Section 4.3).

3 Approach

This section describes our measurement setup and the developed framework for reproducible measurements. We also introduce the network emulation techniques to achieve realistic conditions and a controlled bottleneck buffer. Finally, the tested QUIC implementations and their pacing behavior are shortly presented.

3.1 Measurement Framework

To perform reproducible measurements, we use the framework introduced by Jaeger et al. [18]. The framework is designed to set up and configure measurements on bare-metal servers, offering a consistent environment for QUIC experiments with configurable and extensible logging options. It automates two hosts as client and server to evaluate properties of single QUIC connections in different scenarios. The server is set up to be a QUIC server hosting a file of flexible size. The client

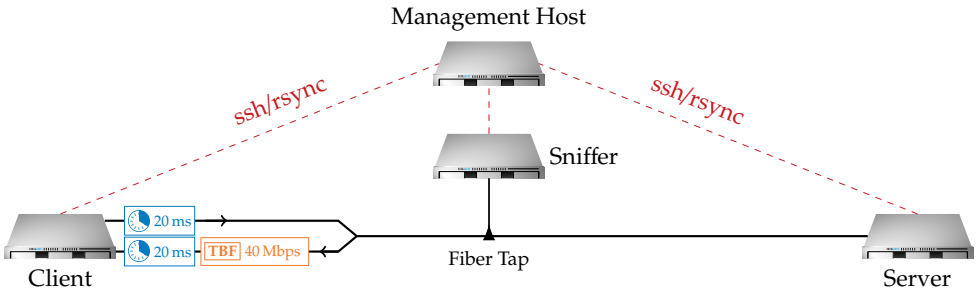


Fig. 1. Topology of the measurement setup with an optical fiber tap. Dashed links only carry management traffic, solid links only carry measurement traffic. Delay and rate limiting are denoted in blue and orange.

is instructed to download this file. No information for session resumption is stored on the client, so that each repetition has the same conditions. We focus on the server's pacing behavior during file transmission, as this is the direction of continuous user data flow.

To evaluate the pacing behavior and inter-packet gap, precise timestamps are required. When capturing timestamps at the server, the capturing itself might influence the timing of the packets. On the other side, capturing timestamps on the client only provides accurate values if no network emulation is applied, as network emulation usually re-shapes traffic. To avoid these issues, we extended the framework to support a third measurement host, the sniffer, which is responsible for capturing packets on the wire between client and server. We rely on a passive optical fiber tap, which forwards a copy of all packets to the sniffer. The resulting topology is shown in Figure 1. The optical fiber tap and the sniffer allow us to capture precise timing information without influencing the connection or involved hosts. *MoonGen* [13] is used to capture packets on the sniffer as it offers high precision with timestamp resolutions below 2 ns.

Client and server are both equipped with an *Intel I210 Gigabit Fiber* NIC. We chose this card, as it provides support for the *LaunchTime* feature, which is part of our measurements. The sniffer is equipped with an *Intel E810-XXV* NIC. All hosts run Debian Bookworm with client and server running on Linux kernel *6.1.112-rt30* and the sniffer running on *6.1.0-17-amd64*.

3.2 Network Emulation

In our setup, client and server are directly connected by a 1 Gbit/s link with a latency of less than 1 ms. As this ultra low RTT might cause unwanted side effects in the congestion control mechanisms, we emulate a bandwidth of 40 Mbit/s with a minimum RTT of 40 ms. While real-world networks may introduce additional dynamics such as cross traffic, queue sharing, competing flows, or load balancing, we intentionally avoid these complexities to ensure reproducibility and focus on stack behavior. We use *tc* to set up a TBF for bandwidth limitation and *netem* for the RTT emulation [23, 24]. While qdiscs are usually used for egress traffic, we need to shape the incoming traffic at the client to not influence the pacing done by the server sending data. Therefore, we use an intermediate functional block on the client to use a TBF on incoming traffic at the client. The bandwidth from client to server does not need to be limited as mainly ACKs are sent in this direction. To achieve the desired RTT, 20 ms are applied directly after the TBF as well as for outgoing traffic at the client. This split is necessary to avoid measuring the uncommon scenario of highly asymmetric latency. However, to avoid unwanted packet loss due to the usage of *netem* after TBF, we increase the buffer size to fit two Bandwidth-Delay Products of packets. Due to the independent monitoring with the sniffer, we are able to capture packets sent by the server before any traffic shaping is done.

3.3 Implementations

For our measurements, we use the example client / server implementations of the QUIC libraries quiche [6], picoquic [29], and ngtcp2 [36]. We chose these implementations as they follow different approaches for pacing. While the pacing rate calculation works the same for all three implementations, the actual pacing is done differently. Cloudflare's quiche calculates an optimal sending time for each packet and uses the `SO_TXTIME` socket option as described in Section 2 to pass this timestamp to the kernel. A qdisc like FQ or ETF can then schedule the packets based on this timestamp. In contrast, ngtcp2 does not depend on system clocks. An application is responsible to send packets at the correct time as calculated by the library. For both implementations, the timestamp of the current packet is based on the previous packet's timestamp and the pacing rate. picoquic depends on the application to respect timestamps and wait until it is allowed to send, but it uses a leaky bucket algorithm for pacing as proposed in RFC 9002 [16]. This credit-based approach allows small bursts after inactivity in contrast to the interval-based approach of quiche and ngtcp2. For the comparison with TCP / TLS, we use *nginx* and *wget* explicitly enabling TLS to ensure a fair comparison with QUIC, which includes both transport and encryption [20].

3.4 Limitations

The results presented in Section 4 are based on a single, specific network configuration (40 Mbit/s bandwidth, 40 ms minimum RTT). We chose these conditions to study pacing in a controlled environment. This setup allows for reproducible measurements and focuses on stack behavior. The exact findings are specific to these fixed parameters and may not be directly generalizable to all network conditions However, general trends and differences in behavior are visible and explainable with the implementations. Real networks have different and complex properties, like varying burstiness, cross traffic, shared queues, competing connections, or load balancing. The impact of pacing and burstiness can be highly context-dependent, and it can be expected that the results vary under different network configurations. We leave the evaluation of pacing in further network scenarios to future work.

3.5 Ethics

This work relies solely on a controlled test environment utilizing synthetic data. Therefore, it does not involve any real-world user traffic or personal information, raising no privacy concerns or other ethical issues.

4 Evaluation

We evaluate the pacing behavior of QUIC implementations using the measurement framework described in Section 3. Our goal is to analyze how different configurations and system-level features influence pacing. We establish a baseline using QUIC implementations and the OS in their default configurations. We compare results to the TCP / TLS stack. This provides insights into the general behavior of libraries and allows a comparison with more advanced configurations. Next, we investigate the impact of different Linux qdiscs on pacing, focusing on their ability to regulate packet transmission. Subsequently, we analyze how GSO affects pacing in QUIC, also considering an approach that paces packets from a single GSO buffer inside the kernel. Finally, we evaluate hardware offloading capabilities by enabling the ETF qdisc with support for *LaunchTime*.

During each measurement, the server transferred a 100 MiB file to the client over HTTP (v3 for QUIC and v2 for TCP/TLS). Each configuration was repeated 20 times and the values of all repetitions are combined for the evaluation. Before combining measurements, we verified the stability of results and found that the presented inter-packet gap and packet train length metrics

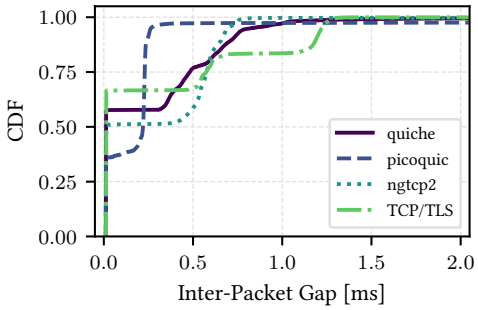


Fig. 2. Distribution of gaps between packets sent by the server during the download for the baseline measurements. All implementations use CUBIC.

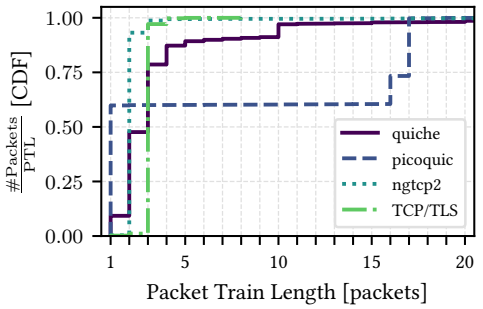


Fig. 3. Distribution of packets across packet trains with specific lengths (x-axis) for the baseline. Packet trains are sequences of packets with at most 0.1 ms between two packets.

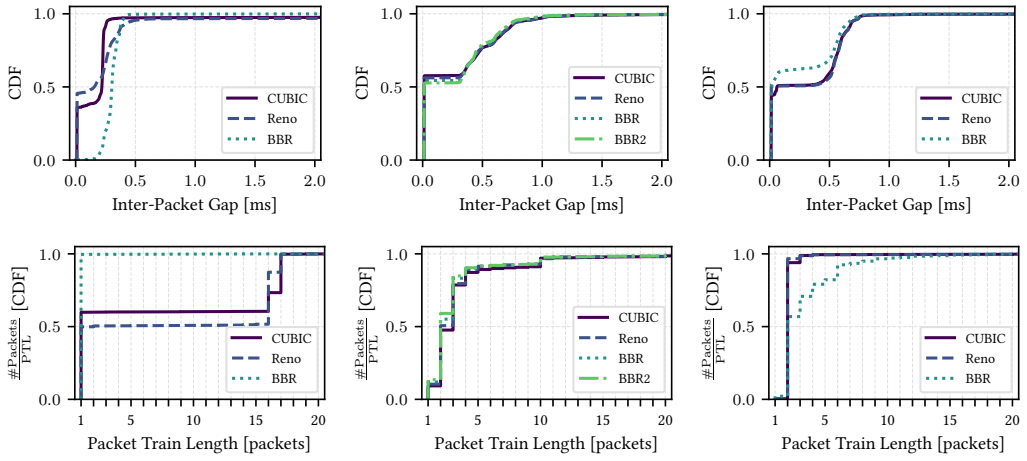
showed a small standard deviation. The UDP receive buffer size was increased to 50 MiB to prevent packet loss at the client [21].

4.1 Baseline Measurement

For the baseline measurements, we use the default settings for all QUIC implementations and set the CCA to CUBIC for comparability. Figure 2 shows the Cumulative Distribution Function (CDF) of the inter-packet gaps. We observe that even though the distributions differ, approximately 50 % of the packets are sent back-to-back without being paced. picoquic sends slightly fewer packets back-to-back, only reaching 40 %. Another commonality is that the majority of packets is sent with inter-packet gaps of less than 1.5 ms.

Looking only at these distributions does not provide a clear picture of the pacing behavior. To gain further insights, we analyze the distribution of the lengths of the packet trains in Figure 3. All consecutive packets with an inter-packet gap of <0.1 ms each are considered a packet train. Since the minimum theoretical inter-packet gap in our setup is approximately 0.012 ms, a 0.1 ms threshold is sufficiently larger to distinguish between fundamental serialization delays and actual bursts or intentionally paced packets. A packet train of size one is a single packet. As larger packet trains are bursts, we consider small packet trains and a rather constant inter-packet gap to be necessary for good pacing.

For TCP / TLS and ngtcp2, more than 99.9 % of the packets are in packet trains consisting of five packets or less, demonstrating consistent pacing behavior. In contrast, picoquic limits packet trains to five packets or fewer for only 60 % of the packets, and quiche achieves this for 89 %. After taking a closer look at the distributions, we see that picoquic has 3.6 % of packet trains, making up almost 40 % of all packets, consisting of 16 or 17 packets. We observed that those bursts are usually sent after a 5 ms idle period happening almost every 10 ms. This behavior is visible with CUBIC and NewReno as CCA but not present with BBR. Figure 4 compares the pacing behavior for different CCAs. The built-in optimization of BBR in picoquic to pace traffic can be fully utilized and traffic is close to perfectly spaced. Only loss-based CCAs result in bursty behavior in picoquic. This behavior is not visible with the other implementations. For quiche and ngtcp2, bursts are smaller when CUBIC or NewReno are used. However, both libraries do not reach the pacing behavior of picoquic if BBR is used, but the pacing behavior is similar to the baseline. The BBR implementation of ngtcp2 even leads to an increase of loss by an order of magnitude.



(a) picoquic (b) quiche (c) ngtcp2

Fig. 4. Comparison of the QUIC libraries picoquic, quiche and ngtcp2 configured with different CCAs. For each library, the top subfigure shows the inter-packet gaps while the bottom subfigure shows the packet train lengths (PTLs).

Table 1. Reached goodput and number of dropped packets during the baseline measurements. The bandwidth is limited to 40 Mbit/s. Packets are dropped due to a full buffer at the introduced bottleneck.

Implementation	Dropped packets	Goodput [Mbit/s]
quiche	687.15 ± 338.12	34.67 ± 0.64
picoquic	861.45 ± 99.53	37.09 ± 0.03
ngtcp2	503.45 ± 7.39	15.93 ± 0.00
TCP / TLS	16.50 ± 0.67	37.37 ± 0.02

The packet train length of quiche is more evenly distributed across values between 6 and 20 and does not show any significant outliers. Looking at Table 1, it is evident that the standard deviation of both metrics is highest for quiche. A patch for quiche presented in Section 4.2 unravels the origin of these observations, as they were not caused by the missing pacing. quiche itself does not pace packets but relies on a suitable qdisc to schedule packets based on timestamps.

4.2 Fair Queue

To evaluate kernel-supported pacing, we use the FQ qdisc, which integrates with quiche's pacing mechanism to schedule packets based on timestamps. With FQ, we surprisingly observe a performance decrease. While the goodput worsens to 33.64 ± 0.89 Mbit/s, the number of dropped packets increases to 1022.55 ± 324.33 packets. The overall stability of metrics decreased, resulting in an increased standard deviation for both metrics. The tail of the packet train length distribution shown in Figure 5 reveals that the number of packet trains with more than five packets increased. A detailed analysis of the source code and individual loss events shows that after a packet is lost, quiche periodically performs a rollback on the congestion window. This happens multiple times in a row, leading to more packet loss. We were able to observe periods of up to 5 s where the congestion window fluctuates between two values, causing a high number of packet loss. The only scenario in which CUBIC may roll back the congestion window is the detection of a spurious congestion

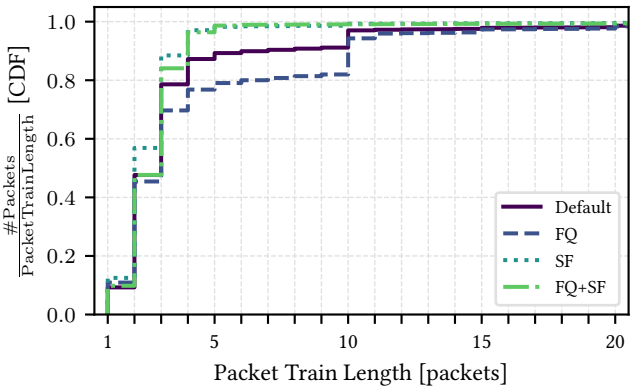


Fig. 5. The impact of the FQ qdisc on quiche pacing. SF denotes our quiche patch to change the behavior of the spurious loss detection. Without the patch, pacing can be drastically impacted in case of loss.

event [40]. While spurious loss is usually defined as receiving an ACK for a packet considered lost, quiche also defines any loss that involves a number of packets lower than a specific threshold as spurious loss [7]. When quiche detects packet loss after the start of the current congestion recovery period, it creates a congestion event and initiates a new recovery period, with the congestion controller state checkpointed beforehand. If the acknowledged packet was sent after the recovery period began, quiche's CUBIC algorithm checks whether the increase in lost packets since the checkpoint is below a threshold and restores the checkpoint state if so, creating a risk of perpetual congestion window rollbacks. An example of this behavior can be seen in Appendix Figure 7. As observed, pacing worsens this issue by reducing packet loss per cycle, which increases the chances of rollbacks, whereas quiche recovers more quickly in the baseline measurements because of larger bursts of loss. Although this mechanism was originally introduced to improve performance [44], we deactivate it for all further measurements, as it causes aggressive behavior and worse performance in our setup.

Comparing the pacing of the quiche baseline with the FQ configuration and the previously mentioned mechanism disabled, we observe similar pacing during large parts of the connection. However, with FQ, packet trains longer than five packets are rare while they make up over 10 % of the packets in the baseline. Due to the ACK-clocking effect, packets are paced during large parts of the connection with both configurations. Only after a congestion event, we observe that ACK-clocking does not lead to a stable pacing behavior without the FQ qdisc. This happens because the congestion window is decreased, but ACKs are still arriving at the old rate. This offset leads to delayed bursts, which are not present when using FQ.

4.3 GSO

GSO can significantly reduce CPU overhead for packet processing in QUIC. However, its batching nature leads to bursts as shown in Figure 6. Pacing single packets with qdiscs, as introduced in Section 4.2, remains possible with batching methods like `sendmmsg()`, but not with GSO. The GSO buffer size, controlled by the QUIC implementation, directly influences this burstiness. While GSO reduces CPU load, the resulting bursty traffic might cause a performance degradation.

To mitigate the burstiness introduced by GSO, two approaches can be considered. The easier approach is to send smaller GSO bursts and to pace the gaps between them. This can be done by adjusting the GSO buffer size calculation in the QUIC implementation. However, this approach does not fully utilize the advantages of GSO and requires a trade-off between CPU load and burstiness.

Table 2. Reached goodput and number of dropped packets during the GSO measurements. The bandwidth is limited to 40 Mbit/s. Packets are dropped due to a full buffer at the introduced bottleneck.

GSO	Dropped packets	Goodput [Mbit/s]
enabled	6.35 ± 1.01	31.06 ± 0.33
disabled	160.80 ± 39.17	31.71 ± 0.08
paced	166.20 ± 41.00	31.71 ± 0.07

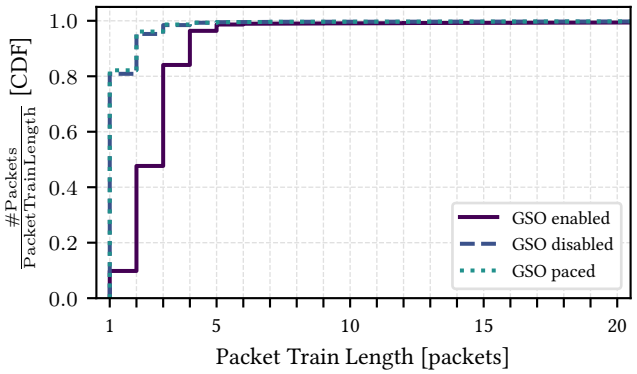


Fig. 6. The impact of GSO on the pacing behavior of quiche. Combining multiple QUIC packets into one buffer reduces system calls but increases the burstiness. GSO-paced is based on a kernel patch that allows senders to provide a pacing rate with the GSO buffer.

The second approach is to pace individual packets within the GSO buffer at the kernel level. Our evaluation focuses on the second approach, as it allows large buffers to be passed to the kernel, maximizing the benefits of GSO while enabling the kernel to pace packets individually. The trade-off here, however, is that this requires modifications to the kernel.

We implemented a kernel patch for GSO pacing based on a proposal by Willem de Bruijn [9]. The patch was adapted for easier integration with QUIC libraries, enabling the kernel to accept a pacing rate in bytes per second for each GSO buffer. Figure 6 shows the results. We compare three variants of quiche with paced traffic: GSO is disabled in the first measurement, GSO is enabled in the second, and GSO is enabled with our kernel patch, allowing the kernel to pace packets within each GSO batch, in the third. The results show that paced GSO achieves pacing behavior similar to GSO-disabled configurations. Over 80 % of packets are sent outside a packet train, indicating effective pacing. However, Table 2 shows that packet loss increases to nearly ten times that of unpaced GSO. This additional loss occurs only at the end of the slow start phase. We attribute this behavior to HyStart++, a modification to the slow start phase to exit early based on RTT increases [2]. With standard GSO, the bursty traffic fills the bottleneck buffer, causing a rapid RTT increase that causes an early exit from slow start. In contrast, paced GSO and GSO-disabled configurations exhibit smoother traffic patterns, leading to a slower RTT increase that does not trigger early slow start exits. Adjusting the parameters of HyStart++ based on the performed pacing could mitigate this issue.

4.4 ETF and Hardware Offloading

We now evaluate the ETF qdisc, which offers support for *LaunchTime* to offload pacing to the NIC. For that, we compare FQ, ETF, and ETF with *LaunchTime*, all with paced GSO. As described in

Section 2, ETF requires a *delta* parameter suitable for the used hardware. Bosk et al. [4] identified 175 µs to be most suitable on a system similar to ours, but they argue that a higher value could reduce packet drops. To be a bit more conservative, we chose a *delta* of 200 µs.

The distributions of the inter-packet gaps and the packet train lengths for the three measurements don't show any significant differences. As the implementations of both qdiscs do not differ much, this is expected.

To evaluate if the *LaunchTime* feature improves pacing precision, we compare expected and actual send timestamps for all packets. As GSO introduces extra complexity and lowers the amount of samples for this metric, the measurements are done without GSO. The expected timestamp together with the QUIC packet number is logged by the quiche server while the actual timestamp and the packet number are retrieved from the packet capture of the sniffer. While the actual difference is not a suitable metric here, the standard deviation, from now on referenced as precision, of the differences is. It is independent of the mean average, which is not meaningful as the clocks of server and sniffer are not synchronized in our setup.

The precision of ETF with and without hardware offloading is 0.27 ms and 0.28 ms respectively. As the behavior and precision do not differ from the ETF qdisc without hardware offloading, we do not consider the *LaunchTime* feature to be beneficial for pacing. Others have also experienced no increase in performance when using ETF with *LaunchTime* [30]. To our surprise, the precision of FQ is 0.12 ms, which is better than the ETF implementations. The baseline measurements without any qdisc show the worst precision with 0.94 ms. This is also expected, as the kernel does not process the timestamps of the packets in this case.

5 Related Work

Pacing for TCP was already investigated in 2000 by Aggarwal et al. [1]. They showed that pacing can have advantages in many scenarios but might also negatively impact the performance. More recent studies confirmed many of these findings [15] and focused on use cases, *e.g.*, integrated into CCA [5, 33, 45] or video streaming scenarios [34]. The ongoing importance of pacing in transport protocols is further highlighted by recent discussions within the IETF, where Welzl et al. [38] provide an overview of pacing mechanisms and their role in improving network performance. Its existence underscores that effective pacing, particularly for user-space protocols like QUIC, remains a relevant area of research.

Differences and the performance of QUIC compared to the existing stack have been evaluated by different works [3, 18, 21, 27, 32, 37, 39, 41, 42]. However, they mostly focus on the reachable goodput, page load times, or offloading features without considering pacing behavior of implementations.

In 2020, Marx et al. [26] compared different QUIC implementations based on their source code. They find that only 8 out of 15 libraries implement pacing. Other research argues that QUIC has advantages due to pacing [10, 43, 46]. However, the quality of the actual pacing is not evaluated. Fastly [28] and Cloudflare [14] have discussed the effects of optimizations on performance, *e.g.*, GSO, but also mentioned drawbacks regarding pacing and suggest further investigation. In contrast, Manzoor et al. [25] explicitly prevent pacing to improve QUIC performance in WiFi. While the increased burstiness improves their results, they did not evaluate inter-packet gaps and the actual pacing behavior in more detail.

We show that even though pacing might be implemented, the interplay of QUIC as user-space implementation with the kernel and offloading features can have a drastic impact. These effects have to be considered in the future and configurations should be adapted based on the use case.

6 Conclusion

In this work, we evaluated the pacing behavior of three QUIC libraries, Cloudflare quiche, ngtcp2 and picoquic. We shed light on the impact of different CCAs and functionalities offered by the kernel. Our results show that pacing behavior differs widely between libraries. While picoquic offers a BBR implementation which evenly spaces packets, other libraries show advantages using CUBIC. quiche can make use of the FQ qdisc for pacing but also GSO to reduce context switches between the library in user-space and the kernel. We show that both functionalities reach the targeted goal, but while GSO reduces the sending overhead, it drastically impacts packet spacing. Our adapted kernel patch for paced GSO, optimized for easy integration into QUIC implementations, combines the efficiency of batching to reduce the overhead of system calls while keeping the pacing behavior. Hardware offloading does not show relevant improvements and reduces pacing precision as intended by the library.

Finally, our work shows that pacing in QUIC is possible and can be achieved using different approaches. Accurate pacing can be entirely done from user-space as shown by picoquic with BBR or with the help of kernel functionality such as the FQ qdisc or when using GSO. However, the behavior differs between implementations and approaches. Depending on the application use case, *e.g.*, video streaming, real-time communications, or web access, different pacing strategies or even no pacing at all might be beneficial. Our evaluation offers important insights for a better understanding of pacing in QUIC, contributing to the ongoing development and successful usage in different scenarios.

Acknowledgment

We thank the anonymous reviewers and our shepherd for their valuable feedback. This work was supported by the EU's Horizon 2020 programme as part of the projects SLICES-PP (10107977) and GreenDIGIT (101131207), by the German Federal Ministry of Education and Research (BMBF) under the projects 6G-life (16KISK002) and 6G-ANNA (16KISK107), and by the German Research Foundation (HyperNIC, CA595/13-1).

References

[1] Amit Aggarwal, Stefan Savage, and Thomas E. Anderson. 2000. Understanding the Performance of TCP Pacing. In *Proceedings IEEE INFOCOM Conference on Computer Communications*. doi:10.1109/INFCOM.2000.832483

[2] Praveen Balasubramanian, Yi Huang, and Matt Olson. 2023. HyStart++: Modified Slow Start for TCP. RFC 9406. doi:10.17487/RFC9406

[3] Simon Bauer, Patrick Sattler, Johannes Zirngibl, Christoph Schwarzenberg, and Georg Carle. 2023. Evaluating the Benefits: Quantifying the Effects of TCP Options, QUIC, and CDNs on Throughput. In *Proceedings of the Applied Networking Research Workshop*. doi:10.1145/3606464.3606474

[4] Marcin Bosk, Filip Rezabek, Kilian Holzinger, Angela Gonzalez Marino, Abdoul Aziz Kane, Francesc Fons, Jörg Ott, and Georg Carle. 2022. Methodology and Infrastructure for TSN-Based Reproducible Network Experiments. *IEEE Access* (2022). doi:10.1109/ACCESS.2022.3211969

[5] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2017. BBR: congestion-based congestion control. *Commun. ACM* (2017). doi:10.1145/3009824

[6] Cloudflare. 2024. *quiche*. Retrieved 2024-12-03 from https://github.com/cloudflare/quiche

[7] Cloudflare. 2024. *Spurious Loss Detection*. Retrieved 2024-12-04 from https://github.com/cloudflare/quiche/blob/115d6ee15465677fc760958019d92776fd28d3e9/quiche/src/recovery/congestion/cubic.rs#L207-L228

[8] Cloudflare Radar. 2024. *Adoption & Usage*. Retrieved 2025-04-26 from https://radar.cloudflare.com/adoption-and-usage

[9] Willem de Bruijn. 2020. *multi release pacing for UDP GSO*. Retrieved 2024-12-03 from https://lore.kernel.org/all/20200609140934.110785-1-willemdebruijn.kernel@gmail.com/#r

[10] Willem De Bruijn and Eric Dumazet. 2018. Optimizing UDP for content delivery: GSO, pacing and zerocopy. In *Linux Plumbers Conference*.

[11] Debian. 2024. *Bookworm Defaults*. Retrieved 2024-12-04 from https://salsa.debian.org/kernel-team/linux/-/blob/21f66065f7e57d90ff70669d29d5cb8ee09e842c/debian/config/config

[12] Eric Dumazet. 2013. *pkt_sched: fq: Fair Queue packet scheduler.* Retrieved 2024-12-03 from https://git.kernel.org/pub/scm/network/iproute2/iproute2.git/commit/?id=bc113e4

[13] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference 2015 (IMC'15)*.

[14] Alessandro Ghedini. 2020. *Accelerating UDP packet transmission for QUIC.* Retrieved 2024-12-03 from https://blog.cloudflare.com/accelerating-udp-packet-transmission-for-quic/

[15] Carlo Augusto Grazia, Martin Klapez, and Maurizio Casoni. 2021. The New TCP Modules on the Block: A Performance Evaluation of TCP Pacing and TCP Small Queues. *IEEE Access* (2021). doi:10.1109/ACCESS.2021.3113891

[16] Jana Iyengar and Ian Swett. 2021. QUIC Loss Detection and Congestion Control. RFC 9002. doi:10.17487/RFC9002

[17] Jana Iyengar, Ian Swett, and Mirja Kühlewind. 2025. *QUIC Acknowledgment Frequency.* Internet-Draft draft-ietf-quic-ack-frequency-11. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-quic-ack-frequency/11/ Work in Progress.

[18] Benedikt Jaeger, Johannes Zirngibl, Marcel Kempf, Kevin Ploch, and Georg Carle. 2023. QUIC on the Highway: Evaluating Performance on High-Rate Links. In *IFIP Networking Conference*. doi:10.23919/IFIPNetworking57963.2023.10186365

[19] Matt Joras and Yang Chi. 2020. *How Facebook is bringing QUIC to billions.* Retrieved 2024-12-03 from https://engineering.fb.com/2020/10/21/networking-traffic/how-facebook-is-bringing-quic-to-billions/

[20] Marcel Kempf, Nikolas Gauder, Benedikt Jaeger, Johannes Zirngibl, and Georg Carle. 2024. A Quantum of QUIC: Dissecting Cryptography with Post-Quantum Insights. In *IFIP Networking Conference*. doi:10.23919/IFIPNetworking62109.2024.10619916

[21] Marcel Kempf, Benedikt Jaeger, Johannes Zirngibl, Kevin Ploch, and Georg Carle. 2024. QUIC on the Fast Lane: Extending Performance Evaluations on High-rate Links. *Computer Communications* (2024). doi:10.1016/j.comcom.2024.04.038

[22] Marcel Kempf, Simon Tietz, Benedikt Jaeger, Johannes Späth, Georg Carle, and Johannes Zirngibl. 2025. *Artifacts for the Paper "QUIC Steps: Evaluating Pacing Strategies in QUIC Implementations".* https://doi.org/10.5281/zenodo.15311561

[23] Alexey N. Kuznetsov. 2001. *tc-tbf - Linux manual page.* Retrieved 2024-12-03 from https://man7.org/linux/man-pages/man8/tc-tbf.8.html

[24] Fabio Ludovici and Hagen Paul Pfeifer. 2011. *tc-netem - Linux manual page.* Retrieved 2024-12-03 from https://man7.org/linux/man-pages/man8/tc-netem.8.html

[25] Jawad Manzoor, Llorenç Cerdà-Alabern, Ramin Sadre, and Idilio Drago. 2019. Improving Performance of QUIC in WiFi. In *2019 IEEE Wireless Communications and Networking Conference (WCNC)*. doi:10.1109/WCNC.2019.8886329

[26] Robin Marx, Joris Herbots, Wim Lamotte, and Peter Quax. 2020. Same Standards, Different Decisions: A Study of QUIC and HTTP/3 Implementation Diversity. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*.

[27] Péter Megyesi, Zsolt Krämer, and Sándor Molnár. 2016. How quick is QUIC?. In *Proc. IEEE ICC*. doi:10.1109/ICC.2016.7510788

[28] Kazuho Oku and Jana Iyengar. 2020. *QUIC matches TCP's efficiency, says our research.* Retrieved 2024-12-03 from https://www.fastly.com/blog/measuring-quic-vs-tcp-computational-efficiency

[29] Private Octopus. 2024. *picoquic.* Retrieved 2024-12-03 from https://github.com/private-octopus/picoquic

[30] Filip Rezabek, Marcin Bosk, Georg Carle, and Jörg Ott. 2023. TSN Experiments Using COTS Hardware and Open-Source Solutions: Lessons Learned. In *Proc. IEEE Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*. doi:10.1109/PerComWorkshops56833.2023.10150312

[31] Jesus Sanchez-Palencia and Vinicius Costa Gomes. 2018. *tc-etf — Linux manual page.* Retrieved 2024-12-03 from https://man7.org/linux/man-pages/man8/tc-etf.8.html

[32] Tanya Shreedhar, Rohit Panda, Sergey Podanev, and Vaibhav Bajpai. 2022. Evaluating QUIC Performance Over Web, Cloud Storage, and Video Workloads. *IEEE Transactions on Network and Service Management* (2022).

[33] Yeong-Jun Song, Geon-Hwan Kim, Imtiaz Mahmud, Won-Kyeong Seo, and You-Ze Cho. 2021. Understanding of BBRv2: Evaluation and Comparison With BBRv1 Congestion Control Algorithm. *IEEE Access* (2021). doi:10.1109/ACCESS.2021.3061696

[34] Bruce Spang, Shravya Kunamalla, Renata Teixeira, Te-Yuan Huang, Grenville Armitage, Ramesh Johari, and Nick McKeown. 2023. Sammy: smoothing video traffic to be a friendly Internet neighbor. In *Proc. ACM SIGCOMM*. doi:10.1145/3603269.3604839

[35] The kernel development community. 2024. *Segmentation Offloads.* Retrieved 2024-12-03 from https://docs.kernel.org/networking/segmentation-offloads.html

[36] Tatsuhiro Tsujikawa. 2024. *ngtcp2.* Retrieved 2024-12-03 from https://github.com/ngtcp2/ngtcp2

[37] Nikita Tyunyayev, Maxime Piraux, Olivier Bonaventure, and Tom Barbette. 2022. A High-Speed QUIC Implementation. In *Proceedings of the 3rd International CoNEXT Student Workshop*.

[38] Michael Welzl, Wesley Eddy, Vidhi Goel, and Michael Tüxen. 2025. *Pacing in Transport Protocols.* Internet-Draft draft-welzl-iccrg-pacing-02. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-welzl-iccrg-pacing/02/ Work in Progress.
[39] Konrad Wolsing, Jan Rüth, Klaus Wehrle, and Oliver Hohlfeld. 2019. A Performance Perspective on Web Optimized Protocol Stacks: TCP+TLS+HTTP/2 vs. QUIC. In *Proceedings of the Applied Networking Research Workshop.*
[40] Lisong Xu, Sangtae Ha, Injong Rhee, Vidhi Goel, and Lars Eggert. 2023. CUBIC for Fast and Long-Distance Networks. RFC 9438. doi:10.17487/RFC9438
[41] Xiangrui Yang, Lars Eggert, Jörg Ott, Steve Uhlig, Zhigang Sun, and Gianni Antichi. 2020. Making QUIC Quicker With NIC Offload. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC.* https://doi.org/10.1145/3405796.3405827
[42] Alexander Yu and Theophilus A. Benson. 2021. Dissecting Performance of Production QUIC. In *Proceedings of the Web Conference 2021.* doi:10.1145/3442381.3450103
[43] Yajun Yu, Mingwei Xu, and Yuan Yang. 2017. When QUIC meets TCP: An experimental study. In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC).* doi:10.1109/PCCC.2017.8280429
[44] Gina Yuan. 2023. *Weird sawtooth cwnd in response to spurious congestion events.* Retrieved 2024-12-03 from https://github.com/cloudflare/quiche/issues/1411
[45] Danesh Zeynali, Emilia N. Weyulu, Seifeddine Fathalli, Balakrishnan Chandrasekaran, and Anja Feldmann. 2024. Promises and Potential of BBRv3. In *Proc. Passive and Active Measurement (PAM).* doi:10.1007/978-3-031-56252-5_12
[46] Xumiao Zhang, Shuowei Jin, Yi He, Ahmad Hassan, Z. Morley Mao, Feng Qian, and Zhi-Li Zhang. 2024. QUIC is not Quick Enough over Fast Internet. In *Proceedings of the ACM Web Conference 2024.* doi:10.1145/3589334.3645323
[47] Johannes Zirngibl, Philippe Buschmann, Patrick Sattler, Benedikt Jaeger, Juliane Aulbach, and Georg Carle. 2021. It's over 9000: Analyzing early QUIC Deployments with the Standardization on the Horizon. In *Proc. ACM Int. Measurement Conference (IMC).*
[48] Johannes Zirngibl, Florian Gebauer, Patrick Sattler, Markus Sosnowski, and Georg Carle. 2024. QUIC Hunter: Finding QUIC Deployments and Identifying Server Libraries Across the Internet. In *Proc. Passive and Active Measurement (PAM).*

A Spurious Loss Behavior: quiche

Figure 7 shows the behavior of the original quiche implementation in case of spurious loss. The behavior is explained in more detail in Section 4.2. It is also discussed in a GitHub issue [44]. It can drastically impact pacing behavior and performance. Therefore, we deactivate the behavior by patching quiche.

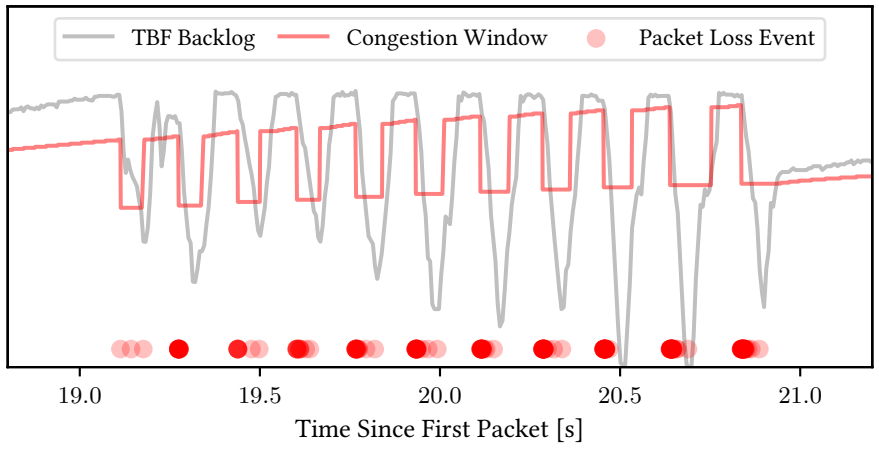


Fig. 7. Example of the quiche behavior in case of spurious loss. The current implementation can lead to perpetual congestion window rollbacks.

B Artifacts

The source code of our measurement framework introduced in Section 3 is published on GitHub [22]. As explained, it is an extension to the framework published by Jaeger et al. [18]. Furthermore, this repository contains the configurations for all measurements and all used patches for quiche and paced GSO. The framework and configurations can be used to reproduce results and study further configurations or libraries.

Besides these artifacts to allow measurements, the collected data from our measurements presented in this work is also published in the same repository. The data contains detailed logs, packet captures and additional information from the involved systems. By using the included evaluation scripts, the data can be analyzed and visualized.

Received December 2024; revised April 2025; accepted April 2025