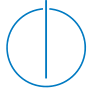


Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

BACHELOR'S THESIS IN INFORMATICS

Analysis of the Pacing Mechanism in Current QUIC Protocol Stacks

Lennart Keller

Technical University of Munich

Department of Informatics

Bachelor’s Thesis in Informatics

Analysis of the Pacing Mechanism in Current QUIC Protocol Stacks

Analyse des Pacing Mechanismus in aktuellen QUIC Protocol Stacks

Author:	Lennart Keller
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Benedikt Jaeger
	Johannes Zirngibl
Date:	February 15, 2021

I confirm that this Bachelor's Thesis is my own work and I have documented all sources and material used.

Garching, February 15, 2021

Location, Date

Signature

Abstract

QUIC is a new transport protocol combining features of the Transmission Control Protocol (TCP), Transport Layer Security (TLS) and Hypertext Transfer Protocol version 2 (HTTP/2). Consequently, QUIC provides reliable and secure transport of data. Two main goals of QUIC are the elimination head-of-line blocking and reduction of initial latency by combining the handshakes of the TCP+TLS stack. QUIC uses the User Datagram Protocol (UDP) as a foundation to allow for fast update cycles. In contrast to TCP, QUIC implements all of its features in user space, including congestion and flow control.

Packet pacing reduces traffic burstiness, which results in less packet loss. Since Linux kernel version 4.13, packet pacing is implemented within the TCP stack. QUIC cannot use the kernel implementation of TCP pacing, as it uses UDP, but has to implement its own pacing mechanism in user space.

This thesis investigates the pacing mechanism of various QUIC implementations and compares it to TCP. We conduct measurements on a testbed using multiple machines and passive fiber TAPs to capture packet timestamps with nanosecond precision.

At 10 Mbit/s, most of the examined QUIC implementations rely on ACK-clocking for packet pacing and show similar behavior to TCP. A QUIC receiver usually sends one acknowledgment after two packets have arrived, which complies with the acknowledgment frequency of TCP. Due to ACK-clocking, many QUIC senders send a burst of two packets every 2.1 ms. The variation around this value is up to five times larger than in the case of TCP.

In contrast to the other examined QUIC implementations and TCP, we found that the QUIC implementation picoquic does not rely on ACK-clocking. A picoquic sender using CUBIC does not send bursts of two packets but spreads 91 % of the packets evenly over time. Picoquic also implements the congestion control algorithm BBR, which integrates packet pacing. Our findings indicate that BBR in TCP can time the sending of a packet two times more precisely than picoquic.

Contents

1 Introduction **1**

1.1 Motivation 3

1.2 Research Questions 3

1.3 Outline 4

2 Background **5**

2.1 Kernel Space vs. User Space 5

2.2 Transmission Control Protocol 7

2.2.1 ACK-clocking 8

2.2.2 Packet Pacing 9

2.3 Congestion Control 10

2.3.1 CUBIC 11

2.3.2 BBR 13

2.4 User Datagram Protocol 14

2.5 QUIC 14

2.5.1 Acknowledgment Mechanism 16

2.5.2 Pacing Mechanism 16

2.5.3 Proposed Congestion Controller 17

2.5.4 Implementation in User Space 17

3 Related Work **19**

4 Analysis **21**

4.1 Sender and Receiver 21

4.2 Measurement Setup 22

4.3 Measurement Techniques 22

4.4 ACK-clocking 24

4.5 Pacing Mechanism 25

4.5.1 BBR 26

4.5.2 Loss-based Congestion Controllers 26

5 Methodology **29**

5.1 Selection of QUIC Implementations 29

5.2 Comparison with TCP 30

5.3 Measurement Setup 31

5.4 Traffic Shaping 32

5.5 Measurement Process 33

6 Evaluation **35**

6.1 Link Utilization 35

6.2 Acknowledgment Mechanism 37

6.2.1 Acknowledgment Frequency of TCP 38

6.2.2 Acknowledgment Frequency of QUIC Implementations 39

6.3 Pacing Mechanism in QUIC 41

6.3.1 ACK-clocking 42

6.3.2 BBR 46

6.3.3 Pacing Rate Calculation 49

7 Conclusion **53**

7.1 Major Contributions 53

7.2 Future Work 55

A List of acronyms **57**

Bibliography **59**

List of Figures

1.1 The TCP+TLS+HTTP/2 protocol stack compared to the QUIC+HTTP/3 protocol stack. 2

2.1 Linux architecture. 6
2.2 TCP 3-way handshake. 8
2.3 Adaption of the sending rate due to ACK-clocking. 9
2.4 Comparison between a traffic burst and evenly spread packets. 10
2.5 Congestion collapse can happen without congestion control. 11
2.6 Comparison of handshakes performed by different protocol stacks between a client and server. 15

5.1 Pacing mechanism measurement setup. 31

6.1 Simplified link utilization measurement setup. 36
6.2 Link utilization at different bandwidths. 37
6.3 Acknowledgment frequency of TCP at different bandwidths. 38
6.4 Acknowledgment frequency over time of TCP senders at 10 Mbit/s. . . . 39
6.5 Acknowledgment frequency of multiple QUIC implementations. 40
6.6 Acknowledgment frequency over time of aioquic at 100 Mbit/s. 40
6.7 Acknowledgment mechanism of a picoquic receiver if BBR is configured. 41
6.8 Acknowledgment mechanism of a picoquic receiver at 10 Mbit/s if CUBIC is configured. 41
6.9 Pacing mechanism of aioquic, ngtcp2 and paced TCP at 10 Mbit/s. . . . 43
6.10 Pacing mechanism of ngtcp2 and non-paced TCP at 100 Mbit/s. 44
6.11 Pacing mechanism of quic-go and non-paced TCP at 10 Mbit/s. 45
6.12 Pacing mechanism of quic-go and paced TCP at 100 Mbit/s. 46
6.13 Distribution of intervals between packets sent by a picoquic sender. . . . 47
6.14 Pacing mechanism of TCP at 10 Mbit/s using BBR. 49
6.15 Pacing mechanism of picoquic at 10 Mbit/s using CUBIC. 50

List of Tables

5.1 Overview of selected QUIC implementations. 30

6.1 Ethernet frame size and according intervals between sent packets. 42
6.2 Theoretical intervals between packets sent by picoquic using BBR. . . . 47
6.3 Cluster sizes of intervals between packets sent by picoquic using BBR. . 48
6.4 Cluster sizes of intervals between packets sent by TCP using BBR. . . . 49

Chapter 1

Introduction

On the Internet, the Transmission Control Protocol (TCP) is used to transfer data reliably from one endpoint to another. It was designed in 1981, and since then, it got many updates. TCP is usually implemented in the operating system kernel. It implements various features in kernel space, like congestion and flow control, loss detection, and retransmission. Additionally, a variety of optimizations for TCP (e.g., packet pacing or ACK-clocking) are implemented in the kernel. Packet pacing avoids sending packets in bursts. Instead, the sender sends packets evenly spread (e.g., over one round trip time (RTT)), which decreases short term congestion and packet loss [1]. If multiple connections share the same network link, pacing can increase the fairness at which the available bandwidth is split between connections [2].

However, updates and upgrades to TCP usually require operating system kernel updates, which slows down the update cycle of TCP.

With the worldwide Internet growth, security features became more and more important. The Transport Layer Security (TLS) protocol operates on top of TCP and is used to authenticate communication partners and encrypt the transmitted data. The application layer protocol Hypertext Transfer Protocol (HTTP) is commonly used in web browsers to retrieve content from websites. The client requests data from a server, which answers with the requested file. With HTTP version 2 (HTTP/2), these request and response pairs can be multiplexed to a single TCP connection to speed up the content delivery [3]. To deploy secure websites on the world wide web, HTTP is used on top of the TCP+TLS protocol stack. This creates the Hypertext Transfer Protocol Secure (HTTPS) protocol stack.

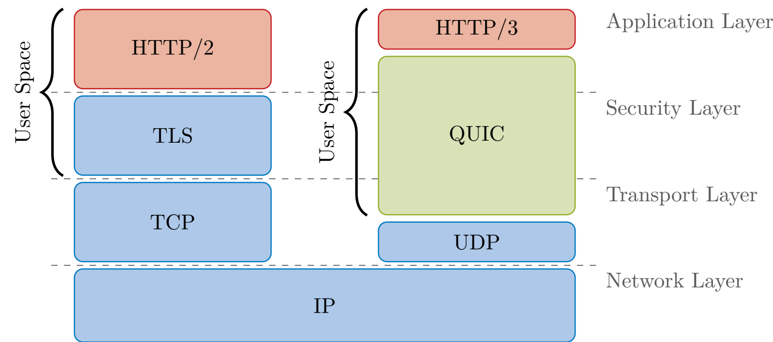


Figure 1.1: The TCP+TLS+HTTP/2 protocol stack in comparison to the QUIC+HTTP/3 protocol stack. Adapted from [7].

The layering of different protocols follows the design of the ISO/OSI standard but also introduces problems.

TCP and TLS perform separate handshakes. TCP uses the transport handshake to set up a reliable connection. Next, TLS performs the cryptographic handshake to secure the connection with authentication and encryption. These handshakes introduce a delay before application data can be transmitted.

HTTP/2 allows stream multiplexing of request and response pairs, but TCP abstracts these streams into a single byte stream. If a packet gets lost, all HTTP/2 streams have to wait for this packet to get retransmitted, even if it did not contain data for a specific stream. This delays other streams and is called head-of-line blocking.

To address these problems of TCP, TLS, and HTTP/2, Google developed the Quick UDP Internet Connections (QUIC) protocol around 2012 [4]. Since 2016, the Internet Engineering Task Force (IETF) is standardizing this new transport protocol [5]. QUIC combines selected features of TCP, TLS, and HTTP/2 into a single protocol. QUIC offers congestion control, flow control, loss detection, and retransmission. Additionally, it has security features, as it incorporates Transport Layer Security protocol version 1.3 (TLS 1.3), and also offers stream multiplexing to address the problem of head-of-line blocking. At the same time, HTTP version 3 (HTTP/3) is standardized, which is supposed to use QUIC as its transport protocol [6].

A comparison between the protocol stack using TCP, TLS, and HTTP/2 and the QUIC+HTTP/3 protocol stack can be seen in Figure 1.1.

Besides well-known features from TCP, TLS and HTTP/2, QUIC also implements new features like connection migration [5]. To be less dependent on middlebox vendors adding support for a new transport protocol, QUIC operates on top of the User Datagram Protocol (UDP).

1.1 Motivation

Unlike TCP, QUIC is implemented in user space instead of kernel space. This means all transport layer features are implemented in user space as well. On the one hand, this design decision allows speeding up the update cycle of QUIC compared to TCP. On the other hand, kernel optimizations of TCP (e.g., packet pacing) have to be implemented in user space. TCP kernel optimizations are meant to improve the performance. Therefore, these optimizations are also valuable for QUIC. The current QUIC draft recommends implementing packet pacing [1].

However, it is not clear how well the TCP kernel optimizations can be implemented in user space for QUIC. User space has limited access to hardware, limited processing time, and generally no privileged functionality. Since packet pacing needs precise timing, it is an open research field to investigate how it works, if it is implemented in user space, and how it affects the performance of QUIC. Past research has shown that QUIC is performing better than alternative protocol stacks in most scenarios [7–12]. For QUIC, it has not been investigated how efficient kernel optimizations can be implemented in user space. In this thesis, we have a closer look at the optimization of packet transmission with packet pacing in QUIC and compare it to TCP.

1.2 Research Questions

We want to examine if QUIC implementations can implement packet pacing in user space effectively. For our analysis of the pacing mechanism in current QUIC protocol stacks, we select a variety of QUIC implementations. These are analyzed and compared to TCP with the help of the following research questions:

To which extend can the pacing mechanism of current QUIC implementations be analyzed?

Different teams develop QUIC implementations in different programming languages. This causes wide performance differences between implementations. For the analysis of optimizations, like packet pacing, the QUIC sender and receiver must not be in overload. Therefore, we have to find a suitable measurement setup, allowing for a sound analysis of the peacing mechanism.

How does the acknowledgment frequency differ between QUIC implementations and TCP?

The QUIC draft does not propose a fixed acknowledgment strategy for the receiver. It primarily elaborates tradeoffs between a high and low acknowledgment frequency.

We examine how the acknowledgment mechanism differs between different QUIC implementations and TCP. The higher the acknowledgment frequency, the more impactful ACK-clocking can be for packet pacing.

How important is ACK-clocking for packet pacing in QUIC?

Packet pacing is related to ACK-clocking. If a sender is ACK-clocked, it spreads outgoing packets based on the rate of incoming acknowledgments. In general, ACK-clocking does not eliminate packet bursts but reduces the burst size. We investigate how much the QUIC implementations rely on ACK-clocking for packet pacing.

How does the pacing mechanism differ between QUIC implementations and TCP?

Due to the variety of QUIC implementations, we expect to see different pacing behavior and different implementation strategies. A perfectly paced sender would send all packets evenly spread [1]. Other pacing strategies may just limit burst sizes. We examine how different QUIC implementations pace packets in our measurements and compare the results. Additionally, we investigate how the pacing mechanism in QUIC sets apart from packet pacing in TCP.

1.3 Outline

This thesis is structured as follows.

Chapter 2 provides background information about TCP, congestion control algorithms, ACK-clocking, and packet pacing. We present details about the QUIC protocol and elaborate differences to TCP.

In Chapter 3, related work about the pacing mechanism in QUIC and the diversity between different QUIC implementations is presented.

Chapter 4 analyzes different measurement techniques to capture packet timestamps. Additionally, it elaborates on which phases during the data transmission we want to focus on for our analysis.

Chapter 5 explains how we deal with the QUIC implementation diversity. It provides a detailed insight into the measurement setup that is used to measure the pacing performance of a QUIC implementation.

In Chapter 6, the measurement results are evaluated. The pacing mechanism of QUIC implementations is compared to pacing in TCP.

Chapter 7 concludes this thesis and suggests future work.

Chapter 2

Background

This chapter presents information about the TCP and QUIC protocols and highlights differences between them. We provide further details about the concept of ACK-clocking, packet pacing, and congestion control.

2.1 Kernel Space vs. User Space

The Linux operating system supports two execution modes for processes: kernel mode and user mode. These offer a different level of authority to protect the operating system against attacks or programming errors by the user. Accordingly, the memory is divided into two distinct areas: kernel space and user space. Each space stores code and data for the respective processes.

The kernel space is reserved for the operating system kernel. It allows all machine code instructions to be executed and has direct access to hardware. Additionally, most device drivers operate in kernel mode.

The user space is meant for user applications to run in. Processes running in user mode are not allowed to run privileged instructions that enable full control over I/O-devices. This includes network interface devices. The user space has no direct access to hardware but can only access it indirectly through the functionality offered by the operating system. The user space process has to use a system call, which hands control over to the operating system.

For networking, user space applications can create a socket, as shown in Figure 2.1. It allows specifying the desired network protocol, either IPv4 or IPv6, and the type

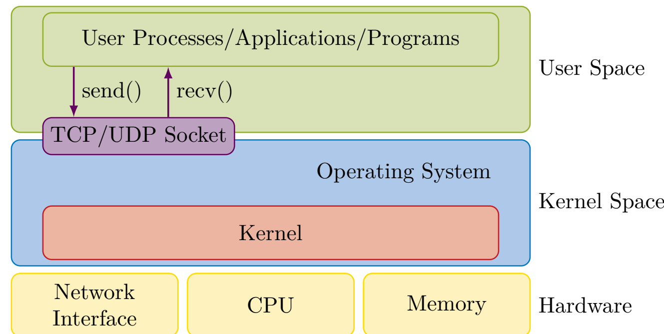


Figure 2.1: Linux architecture.

of the socket (e.g., stream-oriented or datagram-oriented, using TCP and UDP respectively) [13].

The application can write data to the socket with the `send` system call. The kernel is responsible for adding the transport and network protocol headers, like UDP and IPv4, and finding the correct route for the packets. Afterwards, packets are handed to the network device driver, which transmits the packets on the physical layer.

The application can read data from the socket with the `recv` system call. If incoming packets arrive at a host, the kernel identifies the receiving application with the help of the network and protocol device drivers. If the receive buffer is not full, the kernel buffers the data until the application reads data from the socket.

In the case of a datagram-oriented socket, the application reads the data of one packet from the socket with each system call. Analogously, the kernel sends one packet for each `send` call on the sender side.

In the case of a stream-oriented socket, message boundaries are not kept. The kernel decides in which form the data written to the socket is split into packets. The receiver can read no data, partial data from one packet, or data from multiple packets with one `recv` call.

In contrast to datagram-oriented sockets, a stream-oriented socket guarantees that the receiving application will receive all of the data sent by the sending application. The data is ordered and no parts are duplicate.

Since the user mode relies on the system call interface to access hardware, its possibilities to adjust and optimize the behavior of hardware is limited. Additionally, the socket API for networking includes expensive system calls. The user data has to be copied between user space and kernel space whenever the user process reads from or writes to the socket. This can be a bottleneck if high performance or concurrency is needed [14].

With every system call, the user space application execution is interrupted by the kernel. The application's execution state is stored and the kernel takes control. This is known as a context switch, which can include copying of data between user and kernel space [15].

2.2 Transmission Control Protocol

The design of the Transmission Control Protocol (TCP) was finished almost 40 years ago [16]. It is a connection-oriented protocol that adds reliable data transmission on top of the network protocol stack. TCP is implemented in kernel space. A user process only has to create a socket. The kernel is responsible for reliable data transfer.

TCP allows for reliable communication between two processes running on different hosts on the Internet. During transmission, packets can be lost, reordered, or duplicated. Due to reliability, a TCP receiver will receive the complete and unduplicated application data in the correct order a TCP sender transmits. Reliability is achieved with sequence numbers and acknowledgment numbers, which enable loss detection and retransmission of lost packets [16].

TCP abstracts the application data it transports into a single stream of bytes. Each sent packet contains a so-called TCP segment, which consists of the TCP header and application data.

At the beginning of a connection, the two communication parties perform the TCP 3-way handshake [16], as shown in Figure 2.2. One host initiates the connection with a TCP segment in which the *SYN flag* is set inside the TCP header [16]. The other party answers with a similar segment. Finally, this answer is acknowledged by the initial party with another segment with the *ACK flag* set. Within this handshake, the two parties can decide on various options that are used for the connection [16].

The TCP header contains a sequence number. This sequence number increases by one for each byte of application data the sender sends. In the case the network delivered TCP segments in incorrect order, this sequence number allows the receiver to reorder them correctly. The TCP header also includes an acknowledgment number. The receiver of TCP segments uses this header field to inform the sender about the next sequence number it is expecting. TCP uses cumulative acknowledgments, which means an acknowledgment number acknowledges the arrival of all lower sequence numbers. In general, a receiver sends back an acknowledgment every second received TCP segment [17]. The TCP sender is allowed to send a limited amount of data until it has to wait for an acknowledgment.

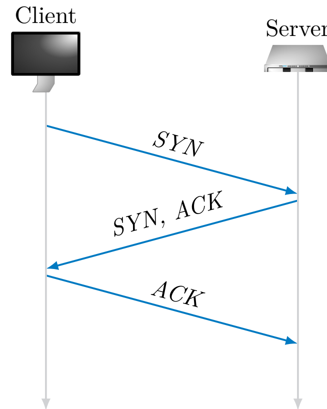


Figure 2.2: TCP 3-way handshake.

The amount of data a sender is allowed to send is restricted by two factors:

1. Congestion control, which avoids overloading the network with packets. Congestion control uses a congestion controller, which determines the size of a congestion window throughout the connection lifetime. This congestion window represents the maximum amount of unacknowledged bytes a sender can keep in transit.
2. Flow control. The receiver sets a receive window size within the TCP header of the acknowledgments it sends back to the sender.

The minimum of the receive window and the congestion window is the actual window the TCP sender uses to limit the number of bytes it sends before waiting for an acknowledgment. Dividing the window size by the RTT yields the average sending rate of a sender [2].

$$\text{average sending rate} = \frac{\text{window}}{\text{RTT}}$$

If the window is limited by the receive window, the sender is called flow control limited.

The data in flight is the amount of data, which was sent by the sender but is still unacknowledged by the receiver. If the receiver sends an acknowledgment, the sending window slides forward based on the number of acknowledged bytes. The amount of unacknowledged inflight data is now less than the window size. Consequently, the sender can send new data until the amount of inflight data matches the sending window size.

2.2.1 ACK-clocking

If the sender sends packets in a large burst or at a higher rate than the bottleneck rate of the network, the bottleneck spreads those packets at the bottleneck rate. The

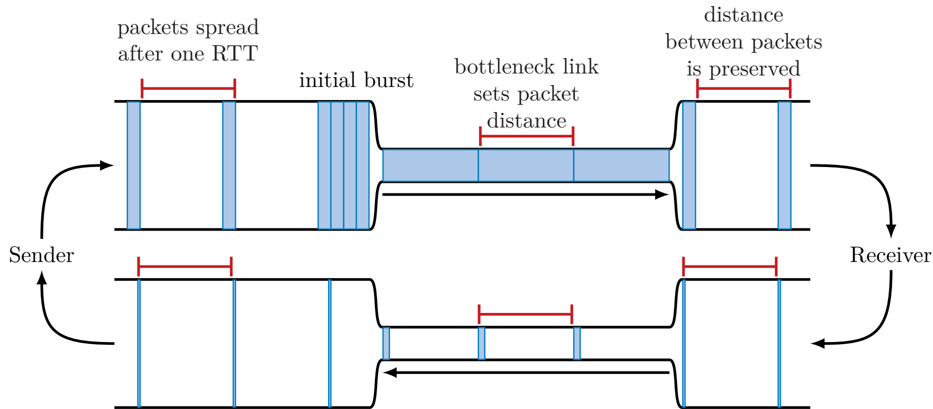


Figure 2.3: Adaption of the sending rate due to ACK-clocking. Adapted from [18, 19].

acknowledgments the receiver sends as a response keep the distance. Once the data in flight matches the sending window size, the TCP sender can only send new data if an acknowledgment arrives. As the incoming acknowledgments are spread according to the bottleneck rate, new data enters the network at the same rate. This mechanism, visualized in Figure 2.3, is called ACK-clocking [2].

In many scenarios, this mechanism is not sufficient for spacing packets effectively.

On a high bandwidth link, which is shared by many different TCP flows, packets likely stay clustered or might generate even larger bursts [2].

The retransmission of lost data can cause the sender to send a large burst of new data. Due to cumulative acknowledgments, the receiver sends duplicate acknowledgments if at least one segment got lost, but multiple new segments arrive. The receiver buffers these segments. If the sender successfully retransmits the lost packet, the receiver can acknowledge all the buffered segments. This allows the sender to send more data than usual.

2.2.2 Packet Pacing

Congestion controllers often send packets in bursts after the arrival of an acknowledgment [2]. The theory of queuing discipline has shown that traffic bursts increase queueing delays, which increases the RTT of a connection. Additionally, traffic bursts lead to more packet loss and, therefore, lower throughput [2].

In front of a bottleneck, the packets arrive at a higher rate than they can be transmitted across the bottleneck link. The arriving packets have to be buffered until they can be transmitted. This introduces queueing delays. If the packets arrive in a burst, the router has to buffer most of the packets at the same time. Therefore, the limit of the

(a) Packets sent in a burst.

(b) Packets sent evenly spread.

Figure 2.4: Qualitative comparison between a traffic burst and evenly spread packets. Adapted from [20].

buffer is likely to be exceeded. Some packets cannot be buffered, which leads to packet loss.

One solution to this problem is to prevent bursts by sending packets evenly spread over one RTT. This mechanism is referred to as packet pacing [2]. Pacing effectively decreases the size of queues. A qualitative comparison between a sender sending packets in bursts and a sender pacing packets can be seen in Figure 2.4.

Pacing can be achieved in two ways. It can be implemented on the sender or receiver side, but sender pacing is more effective [2].

If a sender implements pacing, it is not sending all new packets it is allowed to send after an incoming acknowledgment in one burst. Instead, it calculates a pacing rate by dividing the current window size by the estimated RTT. Following this rate, the packets are sent evenly spread over the next RTT [2].

If pacing is done by the receiver, it spreads acknowledgments over one RTT, which causes the sender to space packets as well due to ACK-clocking. This method is less effective for a couple of reasons [2]. One reason is ACK-compression. Acknowledgments might be buffered behind other acknowledgments on a busy network link. These acknowledgments arrive in a burst at the sender, causing the sender to send a burst of packets as well.

With Linux kernel version 4.13, packet pacing got implemented in the TCP module [21]. Until then, it was only handled by the Fair Queue (FQ) packet scheduler [22], which operates on top of the network device drivers in the Linux kernel but not within the TCP stack.

2.3 Congestion Control

A network should not be under-utilized because this decreases the throughput of a connection. Throughput is a measurement for the total amount of bytes a network can transmit within a specific time period.

A network should also not be overloaded with packets, as this leads to packet loss, which decreases the goodput of a connection. Goodput represents the amount of application

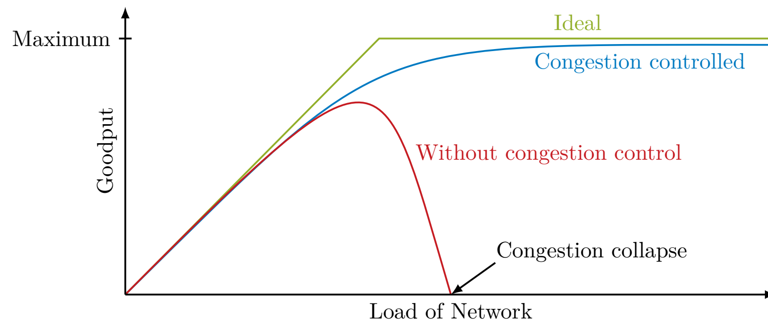


Figure 2.5: Congestion collapse can happen without congestion control.

data that arrives at the receiver within one unit of time. In contrast to throughput, goodput excludes the protocol overhead. Additionally, goodput is negatively impacted by packet loss because packets have to be retransmitted as they did not reach their destination.

One goal of congestion control is to prevent congestion collapse as indicated by Figure 2.5. If too many senders send more data than the network can deliver to the receiver in a particular time, packets are dropped by buffers. Due to the reliability feature of TCP, lost packets are retransmitted. If the amount of data the sender keeps in flight remains higher than the network capacity, retransmitted packets might get lost, causing more retransmissions. Due to an increasing amount of retransmissions, the goodput of the connection decreases.

Congestion controllers use various algorithms to control the size of the congestion window, to find an equilibrium that is neither under-utilizing nor overloading the network [17].

In 1999 the congestion control algorithm TCP NewReno was invented, which introduces improvements to the loss recovery and retransmission algorithms in comparison to older algorithms [23]. It got updated in 2012 [24].

2.3.1 CUBIC

CUBIC is the default congestion controller on Linux since kernel version 2.6.19 [25, 26]. It interprets packet loss as congestion and adapts the congestion window size if packet loss has occurred because it is a lost-based congestion controller. CUBIC is implemented by the sender and only needs feedback in the form of acknowledgments by the receiver. The network itself is not giving any feedback about congestion directly.

The congestion control algorithm is split into multiple phases.

At the beginning of the connection, the slow start phase increases the congestion window exponentially. Per RTT, the sender keeps one congestion window size of data in flight. Every time the sender receives an acknowledgment, it adds one maximum segment size (MSS) to the current congestion window [17]. This effectively doubles the congestion window size each round trip time.

This phase tries to quickly find the maximum network capacity, which utilizes the network to a certain degree without overloading it. The slow start phase ends if either loss is detected or an internal threshold *ssthresh* for the congestion window within the slow start phase is hit.

The congestion controller then switches to the congestion avoidance phase. This phase adapts to network changes and tries to keep the network fully utilized.

The congestion window increases by the pattern of a cubic function [25]. This pattern leads to better utilization of networks with high bandwidth and high RTT than a linear increase of the congestion window [25]. This increase continues until packet loss is recognized. Packet loss can be caused by either random packet loss or by congestion. The TCP sender recognizes packet loss in two ways.

- With each segment sent by the sender, a timer is set. Within this time, the sender expects to receive the belonging acknowledgment. If a timeout occurs, the sender assumes the network is congested and all packets have been dropped by routers. Therefore, the sender retransmits the packets.
- If the sender receives acknowledgments with the same acknowledgment number at least three times, this indicates loss as well [17]. If a segment gets lost, but packets with a higher sequence number arrive at the receiver, it cannot increase the acknowledgment number and sends duplicate acknowledgments. The TCP sender recognizes these and retransmits the lost packet [17].

The sender interprets packet loss as congestion and decreases the congestion window size. If a packet loss is detected by a timeout, the congestion controller resets to the slow start phase. If a loss is detected by duplicate acknowledgments, the congestion controller reduces the congestion window by 30 % [25]. It adjusts parameters for the cubic function and then restarts again to increase the reduced congestion window over time. The TCP sender first retransmits lost data and then sends new data.

The approach of adding a certain amount of bytes to the congestion window every RTT but decreasing the congestion window by a specific percentage if packet loss has occurred is referred to as additive increase and multiplicative decrease [17].

Loss-based congestion controllers likely cause packet loss during the connection lifetime as they continuously increase the congestion window. Over time, this leads to more data in flight per RTT than the bottleneck router can buffer. The router has to drop packets.

2.3.2 BBR

In 2016, the rate-based congestion control algorithm BBR was presented by Google [27]. In contrast to loss-based congestion control algorithms, BBR tries to keep the buffer of the router in front of the bottleneck empty.

BBR estimates the round-trip propagation time (RTprop) and bottleneck bandwidth (BtlBw) of a network path. RTprop is the minimum RTT of the connection, which is constrained by the physical properties in terms of the propagation delay of the network. BtlBw is the maximum bandwidth of the connection, which is limited by the bottleneck of the network. With these two values, the bandwidth-delay product (BDP) can be calculated.

$$\text{BDP} = \text{RTprop} \cdot \text{BtlBw}$$

BBR uses this value as its congestion window. This is the optimal value for the congestion window size since the network cannot deliver more data more quickly [27]. Sending more data yields in queueing delays, which increases the RTT. Therefore, the effective delivery rate is not increased.

In contrast to loss-based congestion controllers, the pacing mechanism is a fundamental part of BBR [27]. BBR avoids building queues, which means it has to adjust its sending rate to the bottleneck rate. Therefore, packets have to be paced.

At the beginning of the connection, BBR doubles the sending rate every RTT in the startup phase, until the measured delivery rate is not increasing any more. Afterwards, the drain phase decreases the sending rate to remove potentially built queues at the bottleneck router until the amount of inflight data matches the estimated BDP.

BBR calculates the sending rate by multiplying the measured bottleneck bandwidth with a pacing gain value [27].

$$\text{sending rate} = \text{BtlBw} \cdot \text{pacing gain}$$

After the drain phase, pacing gain = 1 holds for most of the time during the probing-for-bandwidth phase. To probe for more bandwidth, BBR sets the pacing gain value to 1.25 for one RTprop. If more bandwidth is available, the RTT does not increase.

Afterwards, BBR sets pacing gain = 0.75 for one RTprop to remove potentially built queues if no additional bandwidth is available.

To probe for a lower RTprop value, the sender reduces the sending rate drastically every ten seconds. The amount of data in flight is reduced to four packets for at least one RTprop, trying to empty the bottleneck queue. Based on the received acknowledgments, BBR can detect if RTprop has decreased.

2.4 User Datagram Protocol

The User Datagram Protocol (UDP) is the lightweight counterpart to TCP and a datagram-oriented protocol [28]. It does not perform a handshake to initiate a connection or handles a connection teardown. Additionally, the transmitted application data is not abstracted into a stream of bytes. Instead, each time the application writes data to a UDP socket, this data is sent as one message, even if the amount of data is only a few bytes.

UDP only offers basic functionality to fulfill the role of a transport protocol: Allowing to address a source and destination process on a sending and receiving host. This is achieved by the source and destination port specified in the UDP header.

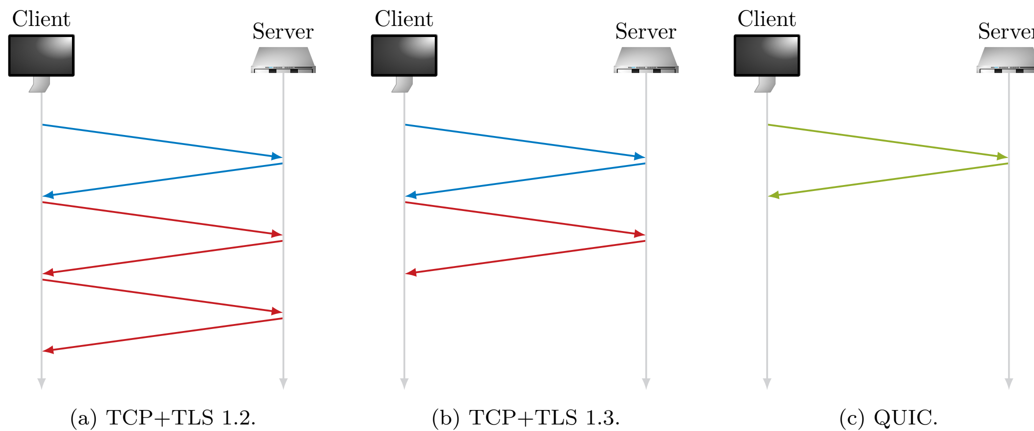
UDP is an unreliable protocol, meaning it does not implement any of the advanced features of TCP for loss detection and retransmission. It also does not support congestion or flow control.

UDP does not implement any of the optimizations for TCP, including packet pacing. Pacing needs feedback from the receiver to estimate a round trip time, but a UDP receiver does not send back acknowledgments. Additionally, the sender has to keep track of a sending window to calculate a pacing rate, but a UDP sender does not support this feature.

2.5 QUIC

QUIC is a new transport protocol implemented in user space, which combines features of TCP, TLS, and HTTP/2 [29]. In 2013, Google started to deploy and experiment with this new transport protocol on their servers [4].

Since 2016, QUIC is standardized by the IETF [5]. The standardization process holds on to the initial concepts of Google's version of QUIC but transforms it into a more modular and extensible general-purpose transport protocol [4, 5]. The most recent draft



(a) TCP+TLS 1.2. (b) TCP+TLS 1.3. (c) QUIC.

FIGURE 2.6: Comparison of handshakes performed by different protocol stacks between a client and server.

version is 34, which is the proposed standard. It is currently submitted to the IESG for publication [30].

Since QUIC includes the stream multiplexing feature of HTTP/2, the QUIC working group at the IETF standardizes HTTP/3 as well [6]. QUIC is the proposed transport protocol for HTTP/3 [6].

QUIC is a reliable transport protocol, like TCP. It offers congestion control and a connection and stream-level flow control mechanism. Additionally, it allows for connection migration if the client changes its IP address or port. The connection is authenticated and encrypted as QUIC incorporates TLS 1.3 [5].

As the QUIC handshake incorporates the TLS 1.3 handshake, it reduces the handshake latency in comparison to the TCP+TLS stack as shown in Figure 2.6.

QUIC uses UDP to transport QUIC packets. One UDP datagram can encapsulate multiple QUIC packets.

Each QUIC packet starts with a header, which contains a packet number [5]. This number increases with every QUIC packet a sender sends. Retransmitted application data is sent within a packet having a new packet number since the packet number is used as a cryptographic nonce for the decryption of the packet. In contrast, the TCP segment number is repeated if a segment is retransmitted.

The payload of a QUIC packet following the header consists of one or more QUIC frames. Each frame has a specific type [5]. The *STREAM frame* carries application data belonging to the specified stream. Various control frame types are used for flow control or the acknowledgment mechanism, for example.

2.5.1 Acknowledgment Mechanism

Like TCP, the default behavior of QUIC is to send back one acknowledgment every second incoming QUIC packet [5]. The acknowledgment mechanism works differently in QUIC than in TCP.

In QUIC, packet headers do not contain an acknowledgment number. Instead, acknowledgments are carried in so-called *ACK frames* within the payload of the packet [5].

An *ACK frame* contains one or more *ACK ranges*, which specify the packet numbers a QUIC receiver has received. Multiple specified acknowledgment ranges indicate a lost packet, as there is a gap between the numbers. Therefore, the sender knows which data it must retransmit.

In general, the QUIC receiver sends back an acknowledgment frame after it has received two QUIC packets [5]. The receiver does not only include one *ACK range*, which acknowledges the last two packets. It also incorporates *ACK ranges* which acknowledge older packets. The sender also sends acknowledgment frames. These tell the receiver which acknowledgments the sender has received. It can adapt the *ACK ranges* accordingly and does not send acknowledgments for older packets anymore.

A QUIC receiver might deviate from the strategy of sending an *ACK frame* every second incoming QUIC packet [5]. This will also be shown in Section 6.2.2.

2.5.2 Pacing Mechanism

To avoid short-term congestion and packet loss, a QUIC sender has to use packet pacing or limit traffic bursts [1]. A perfectly paced sender would send all packets evenly spread. Bursts should be limited to the initial congestion window size, which is a function of the maximum datagram size (MDS). The MDS is the counterpart to the MSS for TCP.

$$\text{initial congestion window} = \min(10 \cdot \text{MDS}, \max(14\,720\,\text{bytes}, 2 \cdot \text{MDS}))$$

If a QUIC packet only contains acknowledgment frames and no application data, it should not be paced [1]. This ensures timely delivery, which is important for loss recovery.

An implemented pacer works closely together with a congestion controller. To calculate the pacing rate, a window-based congestion controller can use the following formula [1].

$$\text{pacing rate} = N \cdot \frac{\text{congestion window}}{\text{smoothed RTT}} \tag{2.1}$$

N is a factor of at least 1. This avoids under-utilization of the congestion window by the congestion controller. Smoothed RTT is an exponentially-weighted moving average of the round trip time estimations throughout the connection lifetime.

As an alternative, the inter-packet interval can be calculated to pace packets [1].

$$\text{interval} = \text{smoothed RTT} \cdot \frac{\text{packet size}}{\text{congestion window}} \cdot \frac{1}{N}$$

This interval is the time a sender has to wait until it sends the next packet, instead of sending all packets at once.

The QUIC draft proposes a leaky bucket algorithm for the implementation of a pacer [1]. A bucket fills according to the pacing rate in Equation 2.1. The capacity of the bucket is limited to the maximum burst size.

2.5.3 Proposed Congestion Controller

The draft for QUIC specifies a loss-based congestion controller, which is very similar to TCP NewReno [1].

The main difference between the proposed congestion controller for QUIC and CUBIC is the algorithm for the increase of the congestion window during the congestion avoidance phase. The proposed congestion controller increases the congestion window linearly [17, 24]. This is achieved by limiting the increase of the congestion window to one maximum datagram size per RTT [1].

Another difference is that the congestion window reduced by 50 % when packet loss has occurred [1], instead of 30 % as in the case of CUBIC [25].

A QUIC implementation does not have to use the specified congestion control algorithm but can use other congestion controllers like CUBIC or BBR [1].

2.5.4 Implementation in User Space

In the case of networking, the kernel handles the processing of the network and transport layer headers of an incoming packet.

For TCP, this includes processing of the TCP header fields and updating TCP's internal state [16]. The kernel decides whether to send an acknowledgment or not. On the receiver side, the received packet contents are buffered according to the sequence number at the correct position in a ring buffer. If the application wants to read from the receive buffer, it can only read up until to the point where there is a gap in the sequence number space due to lost segments [16]. On the sender side, the received acknowledgments are

processed within kernel space. The data the sending application provides with the `send` system call is buffered until the kernel decides to send the data.

For QUIC, an incoming QUIC packet has to be copied between the kernel space and user space before the QUIC packet is processed [15]. This adds additional computational overhead in comparison to TCP for the processing of incoming acknowledgments. Additionally, the acknowledgments in QUIC are encrypted, which means they have to be decrypted by the QUIC application before they can be handled.

A context switch is also happing if the user process writes data to the socket. For TCP the kernel knows about the state of the TCP connection. Therefore, it has to perform many operations, like finding the correct route or applying firewall rules, only at the beginning of a connection [15]. For QUIC, the kernel cannot store any state, which means these operations have to be repeated for every sent packet.

Chapter 3

Related Work

In 2017, Yu et al. [12] were comparing the throughput and packet loss rates of Google's version of QUIC with pacing enabled and disabled. For their measurements, they deployed a testbed in Mininet [31]. Pacing reduces the number of packet losses by 50 %. It increases throughput by 50 % in networks with small buffers. Pacing decreases the throughput up to 20 % under better network conditions with buffer sizes of at least the BDP [12].

We analyze the pacing mechanism of multiple open-source QUIC implementations in further detail. We compare different approaches for packet pacing among different QUIC Implementations. Additionally, we compare the pacing mechanisms to TCP. Instead of using Mininet, we deploy a measurement setup using multiple machines.

Diversity between QUIC Implementations: Marx et al. [32] have compared 15 open-source QUIC implementations in May 2020. They discovered a wide diversity between the implementation of various features of the QUIC protocol. Their major methodology is the analysis of log files created by the QUIC implementations at runtime. We focus on timestamp measurements of transmitted packets.

The authors discovered different acknowledgment behaviors since the QUIC draft does not state a fixed acknowledgment frequency [5]. Some of the QUIC implementations show a narrow variation of an acknowledgment frequency of 2-4. Others show a wider range of up to 1-38. We further investigate the acknowledgment frequency in specific scenarios to evaluate the importance of ACK-clocking for the pacing mechanism in QUIC.

Additionally, they also discovered that only 8 out of the 15 examined implementations had implemented packet pacing because of 'the complexity of the technique [...] and lacking support in the Linux kernel in combination with other optimization techniques' [32]. They did not investigate how well the implemented pacers work in practice but just reviewed the source code. We take measurements for multiple QUIC implementations and analyze how well the implemented pacing mechanisms work.

System Call Interface: User space applications rely on system calls to send and receive data from the network [13]. This system call interface does not allow various optimization techniques to be applied in parallel, like generic segmentation offload (GSO) and packet pacing [20].

Ghedini [20] examined how different system calls for sending data affect the throughput of the QUIC implementation quiche by Cloudfare [33]. The system call `sendmsg` allows to send one UDP packet and a different system call `sendmmsg` to send multiple UDP packets. The latter yielded a higher throughput because fewer system calls were needed, which caused less overhead due to context switches. GSO can be used to improve throughput even further. Without GSO, the application has to use a small buffer for each QUIC packet it wants to send. Each of these small buffers is transmitted as a UDP datagram by the kernel. GSO allows the application to pass one large buffer with data to the kernel, which is then responsible for splitting it into smaller UDP datagrams. This doubled the throughput for both of the investigated system calls.

One problem is that sending multiple UDP datagrams with one system call or using GSO conflicts packet pacing [20]. If the application implements pacing, it can only pass one QUIC packet at a time to the kernel.

Due to these observations, we investigate the throughput of different QUIC implementations on different bandwidths before analyzing the pacing mechanism. We focus on bandwidths the QUIC implementations can fully utilize to avoid any influence of performance limitations on the pacing mechanism.

Chapter 4

Analysis

Packet pacing is a packet transmission optimization that requires the possibility to precisely time packet sending. We want to examine whether packet pacing implemented in user space operates as efficiently as packet pacing implemented in kernel space.

To analyze the pacing mechanism of a QUIC implementation or TCP, we have to capture the traffic sent between a sender and receiver. In this chapter, we discuss different measurement setups and different capture techniques.

4.1 Sender and Receiver

A QUIC or TCP connection can be used to transport application data in either both directions or primarily just one direction. In the former case, both connection endpoints send approximately the same amount of application data to the other endpoint simultaneously. The latter case follows a client-server model, where the client initiates the connection and requests data, which is sent from the server towards the client. In this case, the client is only sending acknowledgments but no or only very little application data.

For the analysis of the pacing mechanism, we need a sender, which sends data packets and tries to pace these outgoing packets. How well these packets are paced can be evaluated by analyzing the time intervals between the packets. The better the pacing mechanism works, the less of a traffic burst is seen. Instead, packets are spread evenly over one round trip time.

On the other hand, we need a receiver that acknowledges the incoming data packets, allowing the sender to send more data towards the receiver.

Consequently, the measurement setup can follow the server-client model, where user data is mostly sent in one direction. We can evaluate the pacing mechanism of the QUIC implementation by measuring how well the server paces packets. The QUIC client only sends acknowledgments, which should not be paced [1].

4.2 Measurement Setup

To evaluate the pacing mechanism of various QUIC implementations, we deploy a measurement setup to collect data. The measurement setup can be realized on real hardware with multiple distinct hosts. Alternatively, a software-based setup with a network emulator like *Mininet* can be used, which creates a virtual network on a host [31].

The advantage of a network emulator is that it can create complex network architectures very easily and quickly because only one machine is needed. On the other hand, it is less suitable for precise measurements than real hardware. The network is just emulated on one host, which means the measurements of timings are deferred if the host is under load.

A hardware-based setup is less flexible but offers the potential for more accurate results, as the load is split on different machines. The network has to be realized with real hardware components.

We can use a simple network to benchmark the implemented pacing mechanism of a QUIC implementation because packet pacing implemented on the sender side does not depend on the network architecture itself. The sender only uses more abstract values, like the RTT and the congestion window, to calculate a pacing rate, as specified in Section 2.5.2. Therefore, a hardware-based setup is more suitable than a software-based setup, as it allows for more accurate results.

With the Linux traffic control functionality, various network conditions can be emulated on a hardware-based setup [34]. It allows specifying queueing disciplines for egress traffic, which can emulate latency introduced by a long network link or a lower bandwidth caused by a bottleneck link. The protocol device drivers enqueue outgoing network traffic into the queueing disciplines, which are responsible for scheduling the outgoing packets. When the outgoing packets leave the queue, they are handed to the network device drivers, which send the packet on the physical layer [35].

4.3 Measurement Techniques

On a hardware-based setup, the measurements can be retrieved in three different ways.

Tcpdump: First of all, the command-line tool *tcpdump* can be used to capture the in- and outgoing packets on each host [36].

On the sender side, the packet capture of *tcpdump* is influenced by offloading features like TCP segmentation offload (TSO) or GSO. In both cases, neither the kernel nor the user space application is responsible for segmenting data into smaller chunks. With offloading enabled, this task is performed by the network interface card (NIC) itself. For transmission, packets can only have a size of the maximum transmission unit (MTU). Otherwise, a NIC or network link cannot transmit the packet.

Consequently, *tcpdump* does not capture each packet sent, but only larger packets with the size of multiple thousands of bytes, instead of just up to 1500 bytes, the default MTU for Ethernet networks [37].

To capture packet timestamps with *tcpdump* on the sender side, these offloading features have to be disabled. Disabling offloading features might interfere with either a QUIC implementation or our measurements for TCP if the sender uses offloading features in its implementation.

For received packets, the *tcpdump* capture is influenced by GRO (generic receive offload) or LRO (large receive offload). Received packets are aggregated together by the NIC, which reduces the number of packets that have to be processed by the network protocol drivers. Again, *tcpdump* only captures these larger packets, but not the individual packets that have been transmitted through the network.

Disabling offloading features on other machines than the sender does not interfere with packet pacing done by the sender. Therefore, it is more suitable to capture the packets with *tcpdump* on a different device than on the sender.

Passive Fiber TAPs: The second method uses passive fiber test access points (TAPs). These split the signals sent via optical fiber between two network devices. A third machine, which is connected to the network TAPs, can capture and analyze the split signals. This measurement technique is the most significant advantage of a hardware-based setup. It allows collecting timestamps for the total traffic created by a QUIC or TCP sender and receiver, without causing overhead on the respective two hosts. Additionally, this measurement technique is not influenced by any offloading features because it recognizes actual Ethernet frames sent via the network link.

If the sending rate of the sender matches the bottleneck rate, no queue is built in the bottleneck router buffer. If the sending rate of a sender is higher than the bottleneck rate or the sender sends a burst of packets, the packets get buffered and delayed by the

bottleneck router. In both scenarios, the packets are transmitted via the bottleneck link according to the bottleneck rate.

Consequently, passive fiber TAPs must not be installed at the bottleneck link but on a faster link between the sender and the bottleneck router. Otherwise, passive fiber TAPs cannot measure if the sender sent a burst of packets or sent packets at the bottleneck rate.

Qlog: The last measurement technique is possible with QUIC implementations, but not TCP. Most QUIC implementations support qlog [38]. Qlog is a JSON-based logging format, which is specifically designed for QUIC. It logs the internal state of a QUIC implementation in the form of events in combination with a timestamp. Qlog was designed to make QUIC implementations easier to debug. Since the QUIC packet header is almost entirely encrypted [5], information from packet captures cannot be extracted without decrypting it. Additionally, each QUIC packet consists of one or more QUIC frames [5]. For these reasons, the complete packet captures have to be stored, including the transmitted application data. Qlog solves this issue, as it logs the frames contained in a sent or received QUIC packet without storing the application data. We can use the information from qlog files to analyze the acknowledgment behavior for each QUIC implementation.

Qlog alone is not sufficient for the analysis of the pacing mechanism. The timestamps are set by the application and therefore do not match the timestamps of the packets that are sent on the physical layer.

4.4 ACK-clocking

Received acknowledgments allow the sender to send more data. Depending on the receiver's acknowledgment frequency, ACK-clocking plays an essential role in packet pacing.

Even if the sender sends a large burst of packets, the effective delivery rate does not increase due to the bottleneck [27]. The delivery rate specifies the rate at which data reaches the receiver and is the amount of delivered data per time.

$$\text{delivery rate} = \frac{\Delta\text{delivered}}{\Delta\text{time}}$$

This delivery rate equals the rate at which a receiver acknowledges data. Due to ACK-clocking, the delivery rate limits the rate at which the sender sends more data, as explained in Section 2.2.1.

If each incoming acknowledgment on the sender side triggers new packets to be sent, the acknowledgment frequency determines the burst sizes of a sender. If the receiver sends an acknowledgment after every incoming packet, the sender can send one new packet after it received an acknowledgment. Even if no packet pacing is implemented on the sender side, the sent packets would be paced if the incoming acknowledgments are not arriving in a burst.

In general, a receiver sends an acknowledgment after two or more incoming packets [5], which would trigger higher bursts to be sent if the sender does not pace packets.

Therefore, to evaluate the pacing mechanism of a QUIC sender, the acknowledgment frequency of the QUIC receiver has to be measured.

4.5 Pacing Mechanism

The pacing algorithm in QUIC calculates the pacing rate with

$$\text{rate} = N \cdot \frac{\text{congestion window}}{\text{smoothed RTT}}, \tag{4.1}$$

as explained in Section 2.5.2.

Using this rate results in an interval of

$$\text{interval} = \text{smoothed RTT} \cdot \frac{\text{packet size}}{\text{congestion window}} \cdot \frac{1}{N} \tag{4.2}$$

between sent packets.

To analyze the pacing mechanism, we do not need a complex network. A simple hardware-based measurement setup can be configured to perform very reliably throughout the whole lifetime of a connection between sender and receiver. Therefore, it is not susceptible to frequent network changes or lossy links. The connection between sender and receiver does not have to share the network with different flows, leading to queueing delays on a router due to congestion. This removes many variables, leading to variating behavior between different congestion controllers. We do not have to consider fairness between multiple flows.

Therefore, if the implemented pacing mechanisms work well, the interval between packets specified by Equation 4.2 should be observed during the steady state of different congestion controllers.

In the following, we discuss which phases of each congestion controller we want to focus on for the analysis of the pacing mechanism. We are interested in phases lasting for the majority of the connection lifetime in which the sending rate of the sender is only fluctuating a little. Short phases with a rapidly changing sending rate do not allow for a consistent pacing rate.

4.5.1 BBR

We do not analyze the pacing within the startup and drain phase since the sending rate changes exponentially and both phases only last a few round trip times [27].

The probing for a higher BtlBw and probing for a lower RTprop are part of the steady state of BBR [27]. Both phases will not discover a new bandwidth or RTT in our static and isolated setup.

In the probing-for-bandwidth phase, BBR periodically increases the pacing rate by 25 % every eighth RTT. BBR decreases the sending rate by 25 % in the next RTT. This phase is relevant to the pacing mechanism analysis because it lasts for the majority of the connection [27].

The probing phase for RTprop only happens every ten seconds [27]. Due to the rare interval and low sending rate, as explained in Section 2.3.2, this phase is not significant for evaluating the pacing mechanism.

4.5.2 Loss-based Congestion Controllers

Window-based congestion controllers control the size of the congestion window to limit the amount of data in flight. Loss-based congestion controllers, like NewReno [24] or CUBIC [25], increase the size of the congestion window until loss occurs.

The pacing rate is calculated by dividing the congestion window through the smoothed estimation of the RTT as specified in Equation 4.1.

At the start of the connection, the congestion window increases exponentially within the slow start phase. Additionally, the sender has only received a few acknowledgments to estimate the RTT, which lead to a varying smoothed RTT value. Therefore, we do not focus on the slow start phase.

The slow start phase at the beginning of the connection is used to start the ACK-clock [17], which supports more accurate RTT estimations during the congestion avoid-

ance phase. Therefore, we focus on the congestion avoidance phase, which lasts for the majority of the connection.

During congestion avoidance, the congestion window increases with every incoming acknowledgment.

If the congestion window is smaller than the BDP, an increase in the congestion window yields a proportional increase of the pacing rate. The RTT does not increase, as no persistent queue at the bottleneck forms, but the congestion window increases. Accordingly, the interval between paced packets decreases proportionally.

Once the congestion window is larger than the BDP, a persistent queue builds in front of the bottleneck. This leads to an increase in the RTT due to the queueing delay. The RTT increases with the same rate as the congestion window. Consequently, the pacing rate is more stable if the congestion window has at least the size of the BDP.

Loss-based congestion controllers can cause packet loss due to the increasing congestion window. Once packet loss has happened, the sending rate decreases to reduce the amount of data in flight, as the congestion window is decreased. Exact numbers depend on the implemented congestion controller. Therefore, the buffer for the bottleneck in the measurement setup should be at least twice the BDP. A multiplicative decrease of the congestion window by 50 % after packet loss still yields a congestion window larger than the BDP.

Packet loss can also happen due to a random packet loss, which cannot be avoided on real hardware. On modern hardware, random packet loss is an order of magnitude lower than the number of losses caused by a full buffer. Therefore, we are not considering random packet loss as a significant factor for our analysis. Nevertheless, experiments and measurements should be performed multiple times to minimize the probability and impact of biased data.

Chapter 5

Methodology

There are more than 15 different open-source implementations [39], which implement features of the QUIC protocol differently [32]. Therefore, we examine multiple implementations representing the QUIC implementation diversity.

The measurement setup is realized with multiple distinct machines. The use of real hardware allows for more accurate results. Our design choices and the performed measurements are explained in this chapter.

5.1 Selection of QUIC Implementations

Due to the diversity between QUIC implementations, we expect to see different behavior in the pacing mechanism between different implementations. Because of this assumption, we conduct measurements with multiple QUIC implementations to allow for a sound analysis of the pacing mechanism.

To allow for a more in-depth analysis of the pacing mechanism of individual QUIC implementations, we select a subset of the available open-source implementations [39]. An overview of the selected QUIC implementations is shown in Table 5.1. This subset represents the QUIC implementation diversity because they are implemented in different programming languages by different teams.

The selected QUIC implementations provide a fully functional example-client and -server implementation to generate network traffic. The client has to initiate a connection and request a test file. Then, the server sends this test file to the client. All implementations support qlog, which supports analyzing the acknowledgment mechanism.

Name	Language	Version	Git commit	Pacing	BBR	CUBIC	NewReno
aioquic	Python	draft-28	`04b28d8`	✓	×	×	✓
picoquic	C	draft-27	`bf84867`	✓	✓	✓	✓
quic-go	Go	draft-29	`eff36f3`	✓	×	✓	×
ngtcp2	C	draft-32	`20d04c3`	×	×	✓	×

Table 5.1: Overview of selected QUIC implementations.

The majority of selected QUIC implementations implement packet pacing. For comparison with a QUIC implementation without an implemented pacing mechanism, we added ngtcp2 [40] to the list. Ngtcp2 is programmed in C, as is picoquic [41], which allows for the comparison of the general performance of a paced and non-paced sender. Additionally, it uses the same congestion controller as quic-go, which allows for the comparison between an implementation with and without pacing.

The QUIC implementations accord with different QUIC draft versions during our measurements, but all of these versions recommend implementing packet pacing [1].

5.2 Comparison with TCP

The measurements for the pacing mechanism of QUIC implementations are compared to pacing in TCP. The command-line tool *nc* (netcat) allows to configure a very basic TCP connection [42]. This serves as a baseline for measuring the pacing performance of different QUIC implementations and TCP. A netcat server is started and waits for a netcat client to connect and request the test file. There are no other higher layer protocols involved.

Using QUIC, the connection is authenticated and encrypted due to TLS 1.3. Additionally, the server and the client application use the application layer protocol HTTP/3 [6] to communicate.

To create a more similar scenario using TCP, a secure Apache web server is used [43]. The server and client use HTTPS to communicate [44]. With the command-line tool *curl*, the client requests a test file over HTTPS [45]. The connection is authenticated and encrypted with TLS 1.3. The client sends an HTTP/2 [3] request to the server, which sends the test file in response. This covers the TCP+TLS+HTTP/2 stack and is a similar scenario to the QUIC implementations' server and client setup.

In the case of TCP, packet pacing can be enabled in three different ways. First of all, the FQ queueing discipline can be configured on the outgoing network interface of the sender [22]. FQ handles the pacing of egress traffic within the kernel, but outside of the

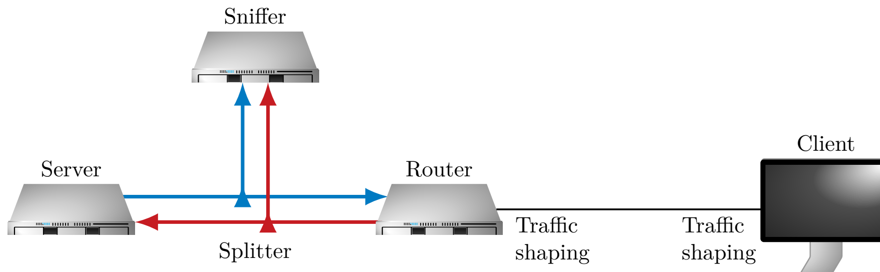


Figure 5.1: Pacing mechanism measurement setup.

TCP stack. Alternatively, the TCP kernel module can be instructed to use BBR. The third way is setting the `SO_MAX_PACING_RATE` socket option on the socket the sender uses to send data to a receiver.

In the case that FQ is not configured, the kernel uses the internal pacing implementation of the TCP stack if it is requested by the usage of BBR or the socket option.

We implement a custom TCP sender and receiver in C using the socket API. The sender sets the value of the socket option `SO_MAX_PACING_RATE` to the bottleneck bandwidth.

5.3 Measurement Setup

To retrieve accurate results, we set up a testing environment with four distinct machines. An overview can be seen in Figure 5.1.

The operating system on every host is Debian 10.3 with a Linux kernel version of 4.19.0-8. All machines are equipped with an Intel Xeon D-1518 CPU @ 2.20 GHz with four cores and 32GB RAM. The hosts are live systems, meaning that everything is stored within the RAM and there is no hard drive.

This setup with four devices allows measuring on real hardware, leading to more precise results than measurements done with network emulators or virtualization software. Additionally, this reduces the overhead on the server and client side as much as possible and serves as a baseline for further experimentation with higher CPU load.

The server and client applications under test are running on two distinct machines. Between the client and server, a third Linux machine is configured as a router and forwards packets between the server and client.

The server and the router are connected with two optical fibers that both support full-duplex mode and a bandwidth of up to 10 Gbit/s. We use two optical fibers because a passive fiber TAP is installed. This device splits signals which are transmitted through

the fiber [46], which only works in one direction. To split signals for both directions, we have to use two fibers.

The server uses the first fiber to transmit packets to the router. The other fiber is used by the router for the reverse direction to forward packets sent by the client to the server. This behavior is achieved by configuring the routing tables on each host. Packets, which are destined to the other machine, are sent via the network interface connected to the correct fiber.

The split signals can be received by the last machine, which operates as a sniffer and captures the total traffic exchanged between server and client on the high bandwidth link.

Due to the sniffer capturing the traffic, neither the server, nor the client, nor the router has to capture the traffic themselves with *tcpdump*, which further decreases the overhead on the machines.

The router and the client are connected by a slower link, with a maximum bandwidth of 1 Gbit/s. This link is the bottleneck of the connection.

This setup allows measuring how well the server implementation can pace packets on a link with a bandwidth of 10 Gbit/s if the bottleneck bandwidth is at or below 1 Gbit/s.

5.4 Traffic Shaping

The bottleneck link between the router and client operates on up to 1 Gbit/s. To emulate a lower bandwidth, we are using the Token-Bucket Filter (TBF) queueing discipline [47]. The command to configure TBF can be seen in the first and second line of Listing 5.1. A *tocken bucket* fills at the specified rate (e.g., 10 Mbit/s). A packet is dequeued if enough tokens are in the *token bucket*. We set the size of this bucket to approximately one packet size. Otherwise, packets can be sent at a higher rate than the specified bandwidth after idle times. This behavior interferes with BBR's estimation of the bottleneck bandwidth [48]. TBF requires setting a buffer size. This buffer determines how long packets can wait for enough tokens to become available. We set the buffer size equal to $4 \cdot \mathrm{BDP}$, which means packets wait for up to $4 \cdot \mathrm{RTT}$ within the buffer. This buffer size should yield more stable results than smaller buffers, as it does not get filled up as quickly and, therefore, packet loss happens less likely.

Listing 5.1: Configuring network interface $IF to limit the bandwidth and add a delay.

```
1 tc qdisc replace dev $IF root handle 1: \
2     tbf rate 10mbit burst 1600b latency 200ms
3 tc qdisc add dev $IF parent 1: handle 2: netem delay 25ms limit 1000
```

Since all the hosts are connected via short cables, the RTT with empty buffers is below 0.5 ms. If a server uses a loss-based congestion controller, the buffer specified for TBF increases the RTT, because the buffer fills as explained in Section 2.3.1. For BBR, the RTT stays low, as BBR avoids filling up buffers [27].

To increase RTprop when buffers are empty, we additionally configure the Network Emulator (NetEm) queueing discipline as shown in the third line of Listing 5.1. It allows specifying a delay to outgoing packets, which effectively emulates a network link with a higher signal propagation time. NetEm uses an internal queue to delay the packets [49]. To avoid that packets have to be dropped because of a small queue within NetEm, we set the limit of packets that can be stored to at least $2 \cdot \text{BDP}$. In theory, one BDP would be enough, as this would be the capacity of a real link.

After packets have passed both queueing disciplines, they are dequeued and then passed to the network interface card, which is responsible for sending the packets on the physical layer.

5.5 Measurement Process

One the server, either the QUIC server application, *nc* in listen mode, the Apache web server, or our custom TCP sender is started. Respectively, on the client machine, the QUIC client, *nc*, *curl*, or our custom TCP client is started afterwards. The client initiates the connection to the server and requests a test file. Next, the server sends the file to the client until it is fully transmitted and the client closes the connection. The test file is large enough to take at least two minutes to transmit. This procedure is done ten times for each implementation. Analysis can be done on averaged results to reduce the impact of unusual behavior caused by random packet loss, for example.

Each time an experiment is conducted, we capture the traffic with qlog and passive fiber TAPs. We do not use *tcpdump* to capture traffic.

To capture and save the split signals on the sniffing host, we use a tool called *MoonSniff* [50]. It allows measuring timestamps of sent packets with nanosecond precision, with an accuracy of ± 20 ns.

Only QUIC clients log their internal state and events with qlog. To reduce overhead on the QUIC server, we disable logging on it. The qlog files contain the acknowledgment frames the client sent. We use this information to determine the acknowledgment frequency. For TCP, the acknowledgment number in the TCP header captured by the sniffer allows determining the acknowledgment frequency.

Chapter 6

Evaluation

We performed measurements with a variety of QUIC implementations. This chapter presents our results and elaborates differences between the acknowledgment and pacing mechanisms of different QUIC implementations and TCP.

6.1 Link Utilization

Measurements for the pacing behavior should be performed on a bandwidth on which the QUIC implementations are not in overload. The sending of packets is application limited if the sender cannot send data as fast as the bandwidth would allow because the performance of the implementation is too low. This scenario does not allow to measure in which way the sender actively paces packets since the sender is overloaded. Therefore, a maximum bandwidth has to be found, on which the QUIC implementations' sending rate is limited by the bandwidth and not due to overload.

Therefore, we measured how well each implementation can utilize a link at various bandwidths with a single stream. For this performance measurement, we used a simplified setup, which is shown in Figure 6.1. It is similar to the full setup in Figure 5.1, but the router is missing. The server and client are connected on a link offering a bandwidth of up to 10 Gbit/s. The traffic can be captured with the sniffer.

This performance measurement was done on four different bandwidths. The 1 Gbit/s and 10 Gbit/s could be configured on the network interface cards themselves by using the command-line tool *ethtool*, as shown in Listing 6.1. For 10 Mbit/s and 100 Mbit/s, we used the TBF traffic shaper with a bucket size of 1600 B and a maximum latency of 200 ms.

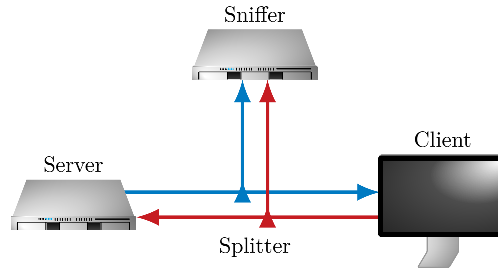


Figure 6.1: Simplified link utilization measurement setup.

Listing 6.1: Configure a bandwidth of 1 Gbit/s on interface $IF.

```
1 ethtool -s $IF speed 1000 duplex full autoneg off
```

We calculated the link utilization by dividing the averaged throughput over the connection lifetime by the configured bandwidth.

$$\text{link utilization} = \frac{\text{throughput}}{\text{bandwidth}}$$

The throughput is the quotient of the total amount of bytes sent and the transmission duration.

$$\text{throughput} = \frac{\text{sent bytes}}{\text{transmission duration}} \tag{6.1}$$

The transmission duration is the difference between the timestamp of the first packet and the last packet belonging to a connection. Each measurement was taking at least 40 seconds.

The amount of sent bytes is the sum of all Ethernet frames the server sent. Each frame contains the Ethernet frame header, the IP header, and a TCP segment or a QUIC packet wrapped by a UDP datagram.

On the client and server machine, only the respective applications were running. Logging and capturing was disabled on both hosts. The packets were captured by the sniffer, allowing to measure the transmission duration and the total amount of transmitted bytes.

The link utilization was measured ten times for each implementation. An overview of the averaged results is shown in Figure 6.2. Every implementation fully utilizes a 10 Mbit/s and 100 Mbit/s link. The only exception is aioquic, which shows a link utilization of 46.92 % at 100 Mbit/s. At 1 Gbit/s, all TCP applications and the QUIC implementations picoquic and ngtcp2 can utilize the link with at least 97 %. The QUIC implementations

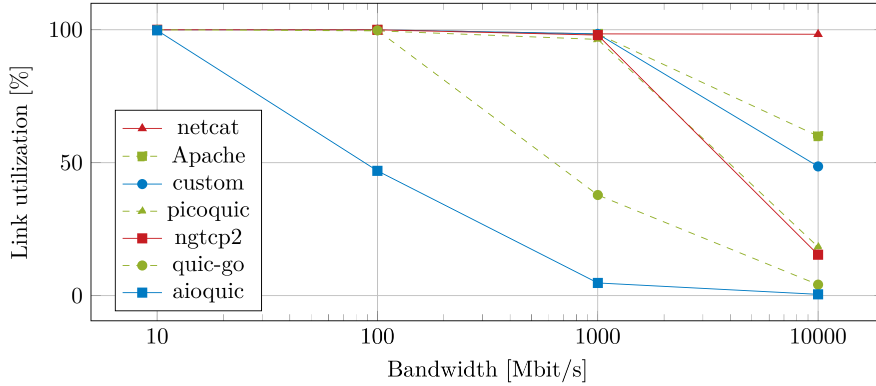


Figure 6.2: Link utilization at different bandwidths.

quic-go and aioquic lose performance. At 10 Gbit/s, the TCP applications are above 48 % link utilization and all QUIC implementations are below 19 %.

Based on these results, we decided to focus on a bandwidth of 10 Mbit/s and 100 Mbit/s for analyzing the pacing mechanism. Except for aioquic on 100 Mbit/s, logging can be enabled on the client side at those two bandwidths without overloading the client.

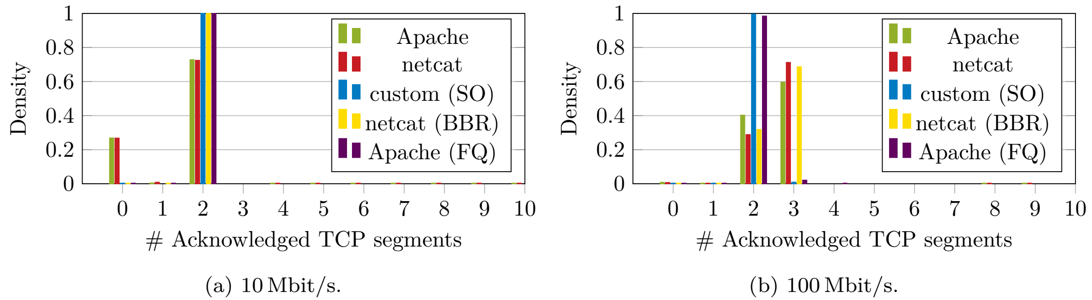
For each measurement, we remove the first five seconds from the analyzed data to circumvent the influence of the connection initialization, startup and drain phases, or slow start phase of various congestion controllers.

6.2 Acknowledgment Mechanism

The acknowledgment mechanism of a receiver is closely related to the pacing mechanism. The sender needs the acknowledgments sent back by the receiver to estimate the RTT of the connection. Packets can also be paced due to the ACK-clocking mechanism.

The more similar the acknowledgment behavior of different implementations, the easier it is to compare their pacing mechanism. With every arriving acknowledgment, which acknowledges new packets, the smoothed RTT value is updated. Ideally, newly sent packets are paced according to the calculated pacing rate. The ACK-clock is established if acknowledgments are sent at a steady rate, which is the bottleneck rate. The sender can use the ACK-clocking mechanism for packet pacing.

The network in our setup, shown in Figure 5.1, is not changing throughout the measurement. The sender and receiver do not share the link with competing parties. The only change is the RTT, which increases if the bottleneck router buffer fills. Loss-based congestion controllers repeatedly cause a slow increase of the RTT due to the additive



(a) 10 Mbit/s.

(b) 100 Mbit/s.

FIGURE 6.3: Acknowledgment frequency of TCP at different bandwidths.

increase of the congestion window during the congestion avoidance phase. A sudden decrease in the RTT happens regularly due to the multiplicative decrease of the congestion window after a packet loss. The rate-based congestion controller BBR keeps the RTT more constant and close to the with NetEm configured RTprop.

If the acknowledgments arrive at a steady rate, they can be used for a reliable estimation of the RTT.

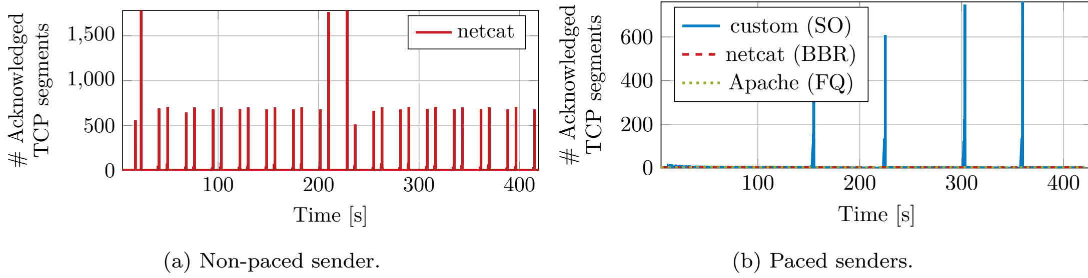
In our setup, the acknowledgments arrive at a steady rate, as they cannot get queued behind other packets in the reverse direction, as only the sender sends large packets containing data.

We examine the acknowledgment frequency of the client on different bandwidths. The acknowledgment frequency counts the number of packets a receiver acknowledges with one acknowledgment. Stable behavior supports the sender with packet pacing. Similar behavior between different implementations simplifies comparing the pacing mechanism between these.

6.2.1 ACKNOWLEDGMENT FREQUENCY OF TCP

Figure 6.3a shows the acknowledgment frequency of two TCP receivers without pacing enabled and three TCP receivers with pacing enabled by either of the options explained in Section 5.2.

CUBIC increases its congestion window and amount of data in flight during congestion avoidance, which causes the bottleneck router buffer to fill up. This leads to packet loss because the bottleneck router has to drop packets. TCP segments with a higher sequence number still arrive at the receiver and get buffered. The receiver sends duplicate acknowledgments due to the cumulative acknowledgment mechanism of TCP. On 10 Mbit/s, 26.52 % of the acknowledgments were duplicate, in the case of netcat and Apache using CUBIC without pacing enabled as shown in Figure 6.3a. If no loss



(a) Non-paced sender.

(b) Paced senders.

Figure 6.4: Acknowledgment frequency over time of different TCP senders at 10 Mbit/s.

occurs, an acknowledgment is sent after two TCP segments have arrived. In the case of TCP with pacing enabled, the amount of duplicate acknowledgments is close to zero, indicating lower packet loss rates.

After the sender retransmits the lost segment, the receiver can acknowledge many packets since it has buffered the arrived segments with a higher sequence number. Figure 6.4a shows their repeating occurrence throughout the connection lifetime.

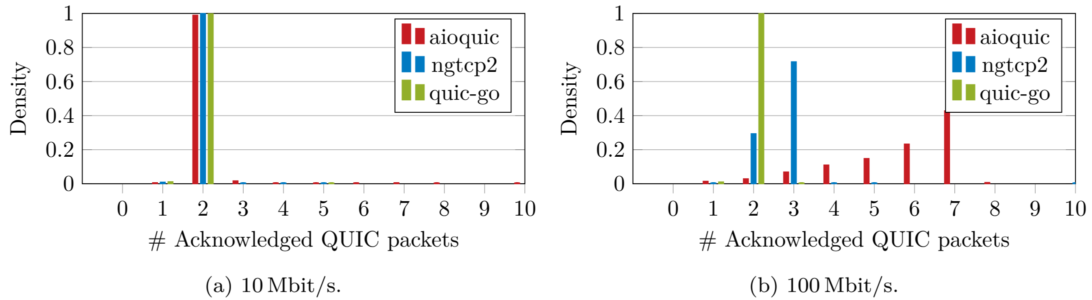
Figure 6.4b shows how those large acknowledgments are not occurring if packet pacing is enabled by using BBR or FQ. They did not completely disappear but yielded a lower frequency in the case of our custom TCP sender and client written in C.

At 100 Mbit/s, the majority of times, a TCP receiver sends an acknowledgment after three packets have arrived if pacing is not enabled or enabled by BBR, as shown in Figure 6.3a. It stabilizes at two if pacing is enabled by the socket option or FQ.

There are much fewer duplicate acknowledgments seen on 100 Mbit/s than on 10 Mbit/s in the case of non-paced senders. This is most likely caused by the larger buffer of the TBF queueing discipline configured on the router. TBF is configured to let packets wait for up to 200 ms in the queue for enough tokens to become available. At 100 Mbit/s, tokens become ten times faster available than on 10 Mbit/s, effectively increasing the queue size by a factor of 10. Packet loss still occurs, but about three times less frequent than on 10 Mbit/s. The receiver sends only a few duplicate acknowledgments in comparison to 10 Mbit/s. This could be caused by the higher bandwidth.

6.2.2 Acknowledgment Frequency of QUIC Implementations

Figure 6.5a shows that three out of the four examined QUIC receivers send an acknowledgment after two QUIC packets have arrived on 10 Mbit/s. This accords with the acknowledgment frequency of TCP at 10 Mbit/s for the most part. The duplicate acknowledgments are missing because QUIC does not use cumulative acknowledgments.



(a) 10 Mbit/s.

(b) 100 Mbit/s.

Figure 6.5: Acknowledgment frequency of multiple QUIC implementations.

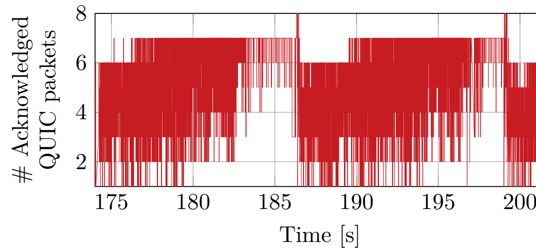
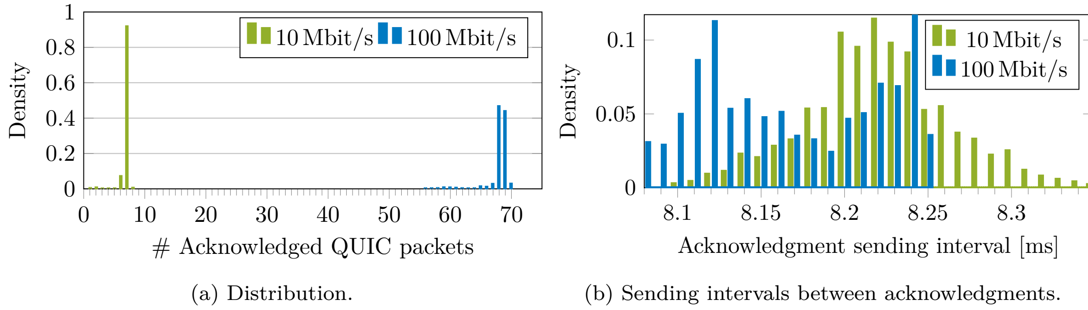


Figure 6.6: Acknowledgment frequency over time of an aioquic receiver at 100 Mbit/s.

At 100 Mbit/s, the acknowledgment frequency is more diverse between the QUIC implementations, as shown by Figure 6.5b. Quic-go still has an acknowledgment frequency of two. Ngtcp2 varies between sending an acknowledgment after two or three packets have arrived, similarly to TCP without pacing at 100 Mbit/s. Aioquic shows the highest variation by sending increasingly many acknowledgments that acknowledge up to seven packets. These do not seem to occur randomly. For most of the transmission, the acknowledgment frequency follows the pattern shown in Figure 6.6. The acknowledgment frequency fluctuates between 2-6 packets. This range narrows down to 5-7 within about ten seconds, before it shows a peak with eight packets and repeats afterwards. This pattern might be caused by aioquic being overloaded on 100 Mbit/s.

In the case of picoquic, the acknowledgment frequency depends on the activated congestion controller.

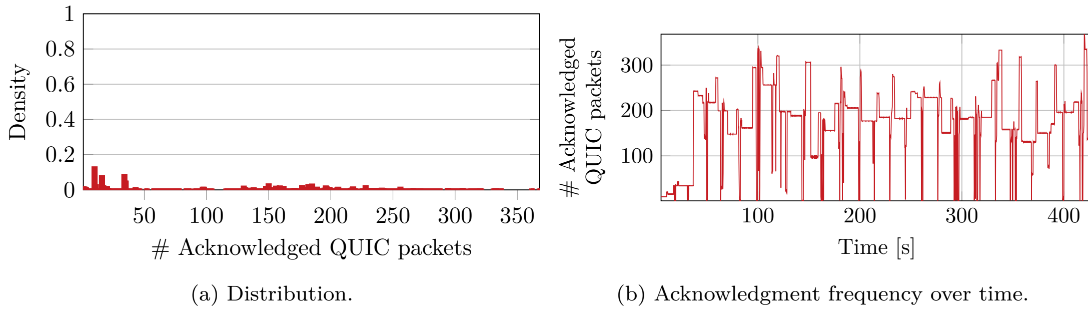
If picoquic is configured to use BBR, it seems to use a different approach for determining the acknowledgment frequency than the other QUIC implementations and TCP. Figure 6.7a shows an acknowledgment frequency of seven at 10 Mbit/s. At 100 Mbit/s, each *Ack frame* the picoquic receiver sends usually acknowledges 68 to 69 packets. Instead of sending an acknowledgment after a certain amount of arrived packets, picoquic sends an acknowledgment after a certain time interval. In Figure 6.7b one cluster forms at



(a) Distribution.

(b) Sending intervals between acknowledgments.

Figure 6.7: Acknowledgment mechanism of a picoquic receiver if BBR is configured.



(a) Distribution.

(b) Acknowledgment frequency over time.

Figure 6.8: Acknowledgment mechanism of a picoquic receiver at 10 Mbit/s if CUBIC is configured.

about 8.22 ms in the case of 10 Mbit/s. At 100 Mbit/s there are two clusters at 8.12 ms and 8.23 ms respectively.

We tested the acknowledgment frequency of picoquic at 10 Mbit/s if it is configured to use CUBIC. The client shows a very low acknowledgment frequency in comparison to other QUIC implementations and TCP. Often, the receiver sends only one *ACK frame* after 150 to 250 packets have arrived, as indicated by Figure 6.8a. Figure 6.8b shows how the acknowledgments are distributed over time. The acknowledgment frequency stabilizes at values around 200 for about 15 s and then drops to a value often below four for less than one second in most cases.

6.3 Pacing Mechanism in QUIC

The bottleneck bandwidth specifies the maximum delivery rate at which data can be delivered to the receiver [27]. Together with the Ethernet frame size, the interval between packets on a fully utilized link can be calculated.

$$\text{interval} = \frac{\text{frame size}}{\text{bandwidth}} \tag{6.2}$$

Implementation	Ethernet frame size [B]	Packet interval [ms] at 10 Mbit/s	100 Mbit/s
TCP	1514	1.211	0.121
picoquic	1482	1.186	0.119
aioquic	1332	1.058	0.106
quic-go	1294	1.035	0.104
ngtcp2	1294	1.035	0.104

Table 6.1: Median Ethernet frame size and corresponding intervals between sent packets according to Equation 6.2.

By substituting a higher bandwidth with the bottleneck bandwidth, this yields the interval between optimally paced packets on a faster link in front of the bottleneck.

In Table 6.1 the median size for Ethernet frames can be seen for every implementation we examined. For all the TCP applications, the Ethernet frame size is the same since all use the same kernel implementation. For QUIC, the Ethernet frame size varies between different implementations and is overall lower than in the case of TCP.

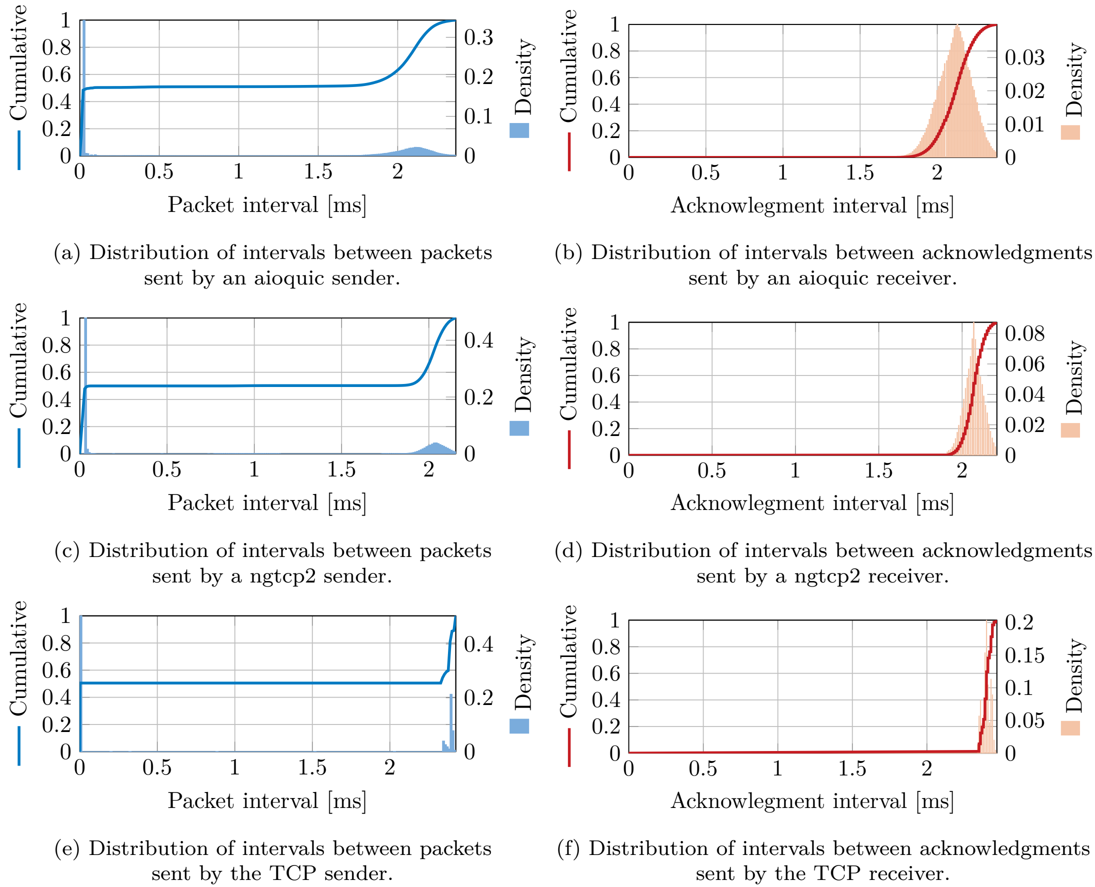
Table 6.1 also shows an overview of the optimal packet intervals according to Equation 6.2 on a faster link, if the bottleneck is 10 Mbit/s or 100 Mbit/s. A perfectly paced sender would send packets evenly spread according to these intervals [1].

In the following, we present our measurement results by analyzing aspects of the pacing mechanism in QUIC and TCP.

6.3.1 ACK-clocking

The ACK-clocking mechanism can be responsible for packet pacing, as explained in Section 2.2.1. The QUIC implementations aioquic, ngtcp2, and quic-go use a loss-based congestion controller and rely on ACK-clocking for packet pacing at 10 Mbit/s.

Figure 6.9a and Figure 6.9c show that half of the packets are sent with no gap in between them in the case of aioquic and ngtcp2. The other half of the packets are sent with a delay in an area around 2.1 ms. This behavior is caused by ACK-clocking as the sender sends a burst of two packets after an acknowledgment has arrived. Figure 6.9b and Figure 6.9d show the intervals between acknowledgments sent by the aioquic and ngtcp2 receiver, respectively. The acknowledgment intervals match the intervals between the bursts of two packets sent by the aioquic or ngtcp2 sender very closely. The difference between the two QUIC implementations is that for ngtcp2 the range of intervals between acknowledgments has only 45.45 % of the width as in the case of aioquic.



(a) Distribution of intervals between packets sent by an aioquic sender.

(b) Distribution of intervals between acknowledgments sent by an aioquic receiver.

(c) Distribution of intervals between packets sent by a ngtcp2 sender.

(d) Distribution of intervals between acknowledgments sent by a ngtcp2 receiver.

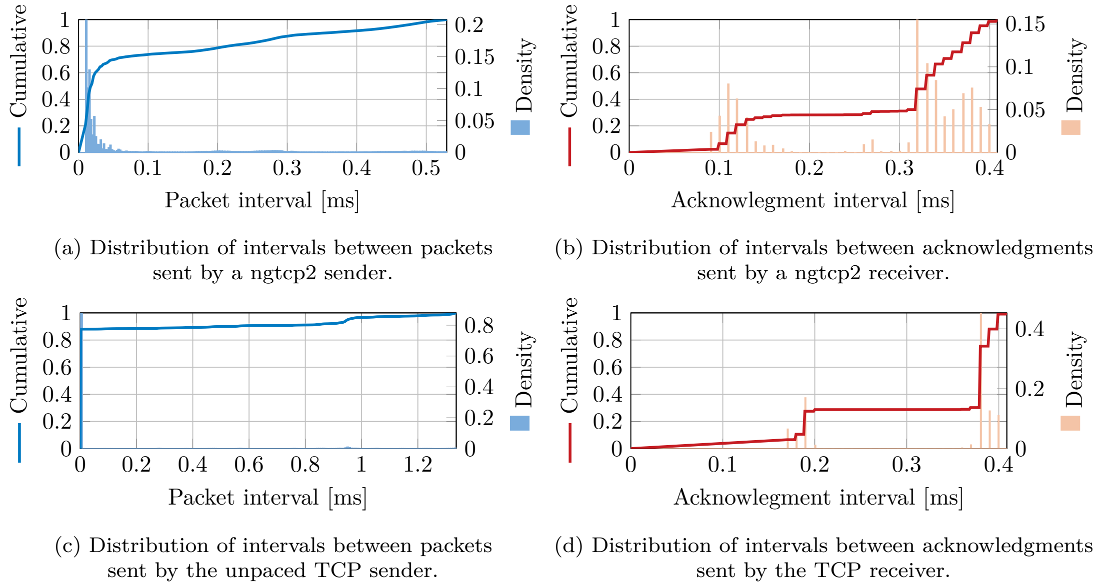
(e) Distribution of intervals between packets sent by the TCP sender.

(f) Distribution of intervals between acknowledgments sent by the TCP receiver.

Figure 6.9: Pacing mechanism of aioquic, ngtcp2 and paced TCP at 10 Mbit/s. The TCP sender sets the value of the `SO_MAX_PACING_RATE` socket option to the bottleneck bandwidth.

The same pacing mechanism as for aioquic and ngtcp2 at 10 Mbit/s is observable in the case of TCP with our custom server and client implementation using the socket option to set a pacing rate. This can be seen in Figure 6.9. Due to larger packet sizes in the case of TCP, the cluster of intervals between acknowledgments is located at 2.4 ms in Figure 6.9f. The range of intervals around that value has 43.33 % of the width as in the case of ngtcp2 and consequently only 19.7 % as in the case of aioquic.

Aioquic is in overload at 100 Mbit/s. In the case of ngtcp2, the relationship between ACK-clocking and packet pacing is not clear on 100 Mbit/s, as indicated by Figure 6.10a and Figure 6.10b. The acknowledgments arrive at a less reliable rate, similarly to the acknowledgments sent by a TCP receiver if the TCP sender does not pace packets, as shown in Figure 6.10d. Figure 6.10b shows clusters of intervals between acknowledgments at around 0.11 ms and between 0.3 ms and 0.4 ms. On 100 Mbit/s, the ngtcp2 sender sends bursts of up to five packets. The intervals between bursts are relatively



(a) Distribution of intervals between packets sent by a ngtcp2 sender.

(b) Distribution of intervals between acknowledgments sent by a ngtcp2 receiver.

(c) Distribution of intervals between packets sent by the unpaced TCP sender.

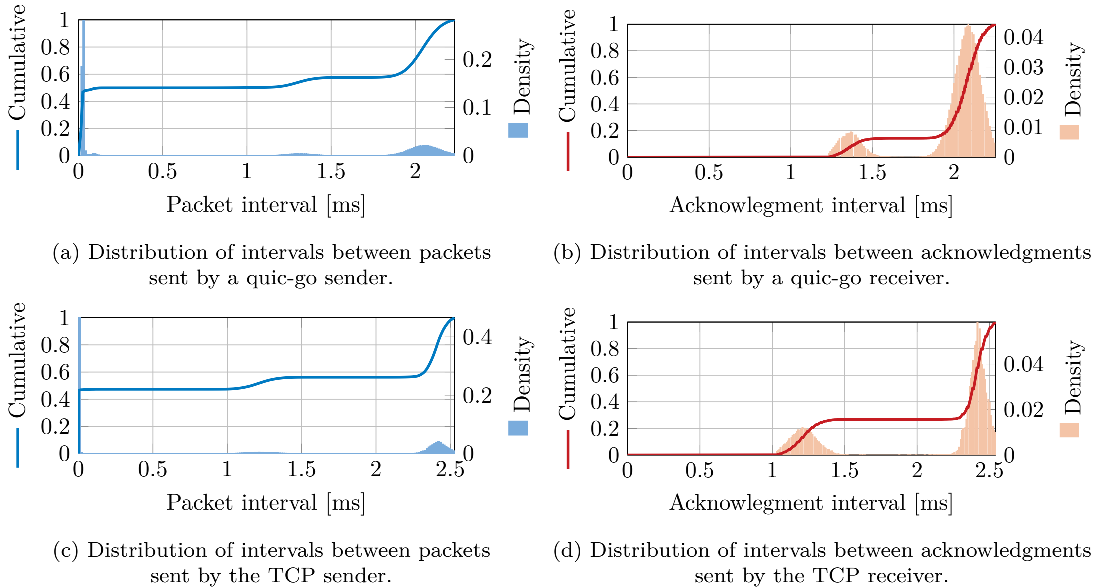
(d) Distribution of intervals between acknowledgments sent by the TCP receiver.

Figure 6.10: Pacing mechanism of ngtcp2 and non-paced TCP at 100 Mbit/s.

evenly distributed between 0.05 ms and 0.5 ms, as indicated by Figure 6.10a. Ngtcp2 does not implement an explicit pacer. Further analysis is necessary for more detailed statements.

Figure 6.10a shows that bursts of packets are not as dense as the 10 Gbit/s link the ngtcp2 sender is connected to would allow. Figure 6.10c shows that the interval between packets in a burst is around 1.2 µs in the case of TCP due to the 10 Gbit/s link the sender is connected to. In theory, the interval between bursted packets on a 10 Gbit/s link would be around 1 µs in the case of ngtcp2. Our measurements show an interval of at least 8 µs. Ngtcp2 uses the `sendmsg` system call to send QUIC packets [51]. Therefore, there is one context switch per QUIC packet [20]. This overhead seems to limit the sending rate of the ngtcp2 sender to about 1 Gbit/s. None of the other observed QUIC implementations seems to be optimizing the number of system calls, which could be one reason for the low performance of all the QUIC implementations at 10 Gbit/s in Figure 6.2. The overhead for each packet varies between programming languages and other implementation details of a QUIC implementation, which is why the observed link utilization varies between different QUIC implementations.

Quic-go seems to rely on ACK-clocking for packet pacing in our measurements at 10 Mbit/s. This is indicated by Figure 6.11a and Figure 6.11b because the intervals between packets sent by the sender match the intervals between arriving acknowledgments. A very similar pattern can be seen for TCP in Figure 6.11 if TCP pacing is not enabled.



(a) Distribution of intervals between packets sent by a quic-go sender.

(b) Distribution of intervals between acknowledgments sent by a quic-go receiver.

(c) Distribution of intervals between packets sent by the TCP sender.

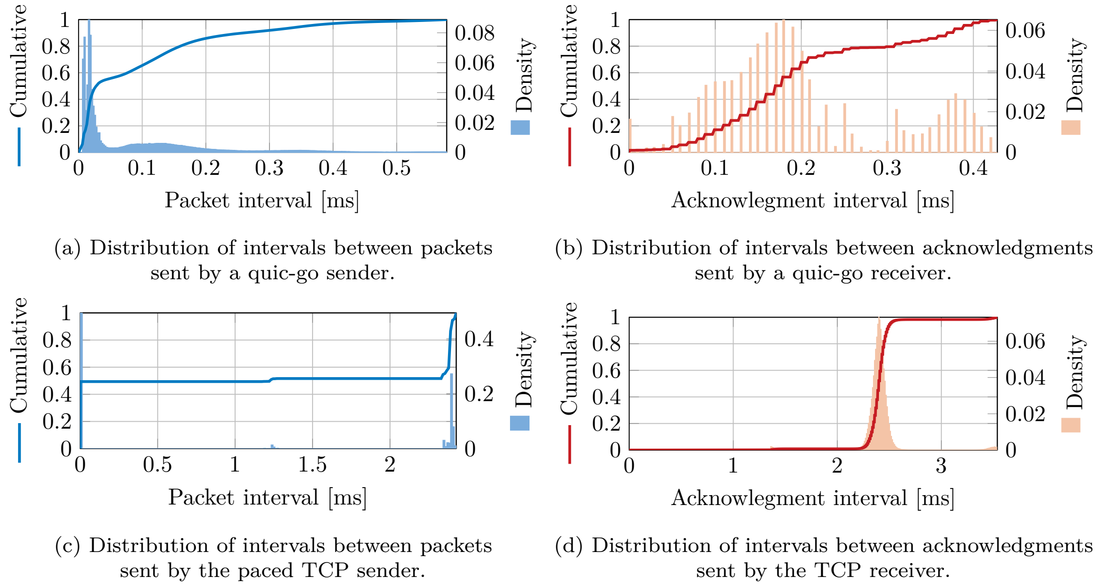
(d) Distribution of intervals between acknowledgments sent by the TCP receiver.

Figure 6.11: Pacing mechanism of quic-go and non-paced TCP at 10 Mbit/s.

In Figure 6.11d, 26.71 % of the acknowledgments are sent with an interval of around 1.2 ms by the TCP receiver. This percentage matches the percentage of duplicate acknowledgments in Figure 6.3a. The TCP receiver sends duplicate acknowledgments immediately if a segment introducing a gap in the sequence number space arrives. It does not wait for a second segment, which is the recommended behavior for TCP receivers [17]. The interval between duplicate acknowledgments is 1.2 ms because the bottleneck delivers packets sent by the server at this interval due to the TCP segment size.

Quic-go seems to follow the recommendation for TCP since the distribution of intervals looks similar to TCP in Figure 6.11. The receiver seems to send an acknowledgment without delay if a packet introducing a gap in the packet number space arrives. Due to the smaller size of quic-go packets compared to TCP, the interval for these immediately sent acknowledgments should be around 1.04 ms. Instead, the cluster is located at 1.37 ms. One reason might be additional overhead for processing a QUIC packet.

The strict acknowledgment frequency of two shown in Figure 6.5 seems to contradict the hypothesis that quic-go sends acknowledgments without delay after loss. We would have expected the quic-go receiver to acknowledge just one packet in a few cases, similar to duplicate acknowledgments in TCP. Further research has to be done to find an exact answer.



(a) Distribution of intervals between packets sent by a quic-go sender.

(b) Distribution of intervals between acknowledgments sent by a quic-go receiver.

(c) Distribution of intervals between packets sent by the paced TCP sender.

(d) Distribution of intervals between acknowledgments sent by the TCP receiver.

Figure 6.12: Pacing mechanism of quic-go and paced TCP at 100 Mbit/s.

On 100 Mbit/s, ACK-clocking could still be important for the pacing mechanism in quic-go. This is indicated by Figure 6.12a and Figure 6.12b due to a similar distribution of intervals between arriving acknowledgments and the bursts of two packets the sender sends.

In the case of our custom TCP sender, which paces packets, ACK-clocking seems to be the primary factor for pacing at 100 Mbit/s. This is indicated by Figure 6.12c and Figure 6.12d. However, Figure 6.12 shows that the intervals between acknowledgments the TCP receiver sends are variating much less compared to quic-go. Further research is required to find the reason for quic-go's behavior.

6.3.2 BBR

Pacing can be achieved by using BBR. Picoquic is the only QUIC implementation we examined which uses BBR.

Because of BBR, the picoquic sender estimates the BtlBw and RTprop value [27]. For most of the time, the sending rate is set equal to the estimated bottleneck bandwidth. During the probing-for-bandwidth phase, every eighth RTprop the sending rate is increased by 25 % and afterwards decreased for one RTprop as explained in Section 2.3.2 and shown in Listing 6.2.

Pacing gain	1.25	1.0	0.75
Interval [ms] with BtlBw = 10 Mbit/s	0.949	1.186	1.581
Interval [ms] with BtlBw = 100 Mbit/s	0.095	0.119	0.158

Table 6.2: Theoretical intervals between packets sent by a picoquic sender using BBR during the probing-for-bandwidth phase.

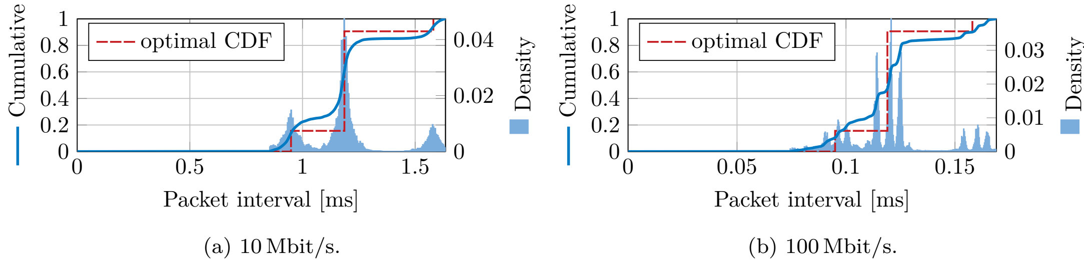


Figure 6.13: Distribution of intervals between packets sent by a picoquic sender.

Listing 6.2: The pacing gain cycle used in the picoquic implementation.

```
1 static const double bbr_pacing_gain_cycle[BBR_GAIN_CYCLE_LEN] = \
2     { 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.25, 0.75};
```

Consequently, the intervals between evenly spread packets on a faster link in front of the bottleneck vary due to BBR since the intervals depend on the sending rate.

$$\text{interval} = \frac{\text{frame size}}{\text{sending rate}} = \frac{\text{frame size}}{\text{BtlBw} \cdot \text{pacing gain}}$$

For picoquic, the theoretical optimal packet intervals on a faster link in front of the bottleneck are shown in Table 6.2.

The distribution of intervals between packets the picoquic sender sent on a 10 Mbit/s connection are shown in Figure 6.13a. There are three clusters visible around the optimal interval values from Table 6.2 indicating that picoquic is able to estimate the bottleneck bandwidth correctly.

If only the phase for probing for more bandwidth is taken into consideration, 75 % of the packets should be paced according to the bottleneck bandwidth in a pacing gain cycle length of eight. About 15.6 % of the packets should be paced with a 25 % higher rate. 9.4 % of the packets should be paced with a 25 % lower rate.

The measured amount of packet intervals in an area of ± 0.1 ms around the interval values according to the different pacing gains are shown in Table 6.3. There are fewer packets in the center cluster and more packets in the two outer clusters than by theory.

Interval [ms]	0.949	1.186	1.581
Theoretical	15.60 %	75.00 %	9.40 %
Measured at 10 Mbit/s	24.78 %	58.84 %	14.52 %

Table 6.3: Percentage of intervals between packets sent by picoquic using BBR in an area of ± 0.1 ms around the given interval value.

Consequently, the cumulative distribution function (CDF) of the observed intervals does not perfectly match the optimal CDF, as shown in Figure 6.13a.

At 100 Mbit/s, the results look similar but show an interesting pattern, which is shown in Figure 6.13b. Again, there are three major clusters visible, but each is divided into three minor clusters. The three major clusters are formed because of the BBR algorithm. The minor clusters are probably caused by variations in the measurements of the bottleneck bandwidth throughout the connection lifetime. The three minor clusters of the center major cluster are located at 114 µs, 120.5 µs and 125 µs. The other minor clusters distribute accordingly.

This pattern does not change if a three times smaller test file is transmitted to reduce the transmission duration. Therefore, the over and under-estimation of the bottleneck bandwidths seems to be occurring in much shorter cycles. At 10 Mbit/s, we did not observe this pattern, even if a file of the same size as on 100 Mbit/s is transmitted.

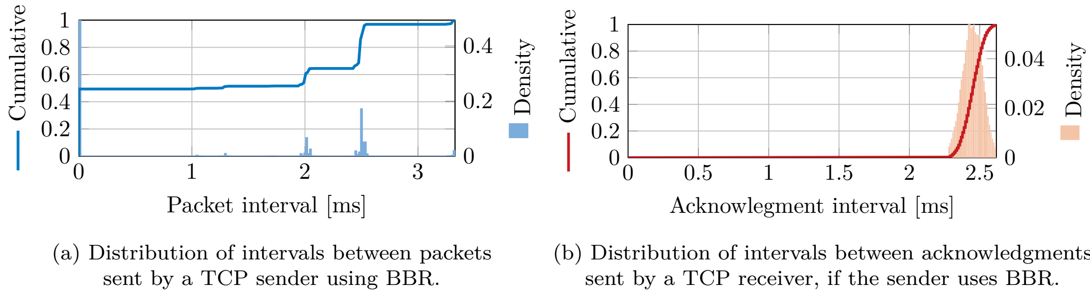
One reason for this observation in the behavior of picoquic's BBR implementation at 100 Mbit/s could be inaccuracies within TBF at a rate of 100 Mbit/s.

The lower acknowledgment frequency we observed at 100 Mbit/s in comparison to 10 Mbit/s in Figure 6.7a is most likely not the reason. In both cases, the picoquic sender receives an acknowledgment approximately every 8.2 ms.

For TCP, the Linux kernel allows using BBR instead of CUBIC as the congestion controller [52]. Figure 6.14 compares the intervals between and packets sent by a TCP sender using BBR and incoming acknowledgments on 10 Mbit/s.

Figure 6.14b shows that the receiver sends an acknowledgment acknowledging two packets every 2.4 ms.

In general, the sender sends bursts of two packets, as indicated by Figure 6.14a due to an interval of just 1.2 µs between 50 % of the packets. In contrast to picoquic, the three clusters in the pattern of intervals are not shown for individual packets. Instead, the pattern applies to the distribution of intervals between bursts of two packets.



(a) Distribution of intervals between packets sent by a TCP sender using BBR.

(b) Distribution of intervals between acknowledgments sent by a TCP receiver, if the sender uses BBR.

Figure 6.14: Pacing mechanism of TCP at 10 Mbit/s using BBR.

Interval [ms]	2.007	2.492	3.310
Theoretical	7.80 %	37.50 %	4.70 %
Measured	12.86 %	32.45 %	3.07 %

Table 6.4: Percentage of intervals in an area of ± 0.1 ms around the given interval value during the probing-for-bandwidth phase of a TCP sender using BBR.

This explains the three clusters around 2.01 ms, 2.49 ms, and 3.31 ms on 10 Mbit/s. These values are more than twice as large as in the case of picoquic because of implementation details of BBR in TCP and the larger size of TCP segments in comparison to picoquic packets [52].

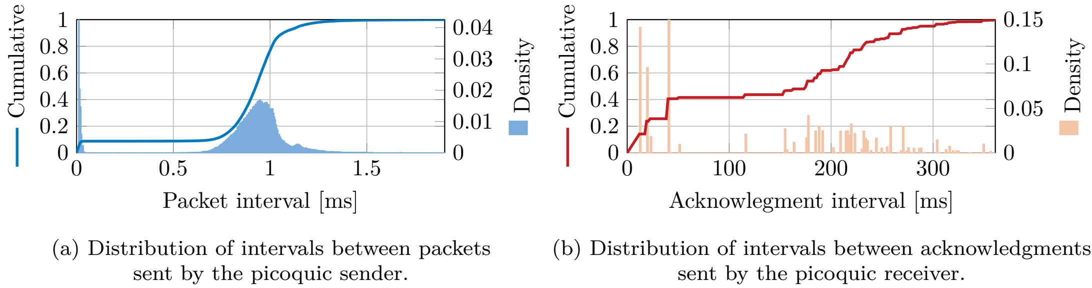
The measured percentage of packets in an area of ± 0.1 ms around 2.01 ms, 2.49 ms, and 3.31 ms are shown in Table 6.3. Similarly to picoquic, there are more packets in the lower cluster and fewer packets in the center cluster than expected by theory. In contrast to picoquic, packets are missing in the upper cluster.

In the case of TCP, the density of intervals falls below a threshold of 0.0002 at a distance of ± 50 µs to the center of the three clusters. In the case of picoquic, this threshold is usually only undercut at a distance of at least ± 120 µs. This could be hinting that the TCP kernel implementation could be able to time the sending of packets two times more precisely than a QUIC user space implementation.

6.3.3 Pacing Rate Calculation

For loss-based congestion controllers, aioquic, guic-qo, and picoquic calculate the pacing rate by dividing the current congestion window through the smoothed RTT value [53–55].

$$\text{pacing rate} = N \cdot \frac{\text{congestion window}}{\text{smoothed RTT}} \tag{6.3}$$



(a) Distribution of intervals between packets sent by the picoquic sender.

(b) Distribution of intervals between acknowledgments sent by the picoquic receiver.

Figure 6.15: Pacing mechanism of picoquic at 10 Mbit/s using CUBIC.

Based on the received acknowledgments, the QUIC sender calculates a smoothed RTT value as an estimation for the RTT of the connection [1]. How much the congestion window and the RTT change over time depends on the used congestion controller of an application. There is no other influence in our isolated setup.

Aioquic uses $N = 1$ in Equation 6.3 and leaky bucket algorithm as its pacer implementation [53]. The rate the bucket fills is constant. A packet needs a different amount of tokens depending on the calculated pacing rate. The maximum capacity of the bucket is limited to 16 maximum datagram sizes. Consequently, the pacer of aioquic allows bursts of up to 16 packets. In theory, the remaining packets of a larger burst will be delayed. In our measurements, those burst sizes were not achieved during the congestion avoidance phase, which is why our results solely show ACK-clocking being responsible for packet pacing.

Quic-go does not implement a leaky bucket algorithm for the pacer but follows a different approach [54]. If the calculated pacing rate results in an interval of below 1 ms between packets, the sender does not delay each packet accordingly but sends packets in a burst instead. Then, the sender waits for the remaining time until 1 ms has passed. Consequently, the pacing implementation in quic-go allows higher bursts on higher bandwidths.

In contrast to aioquic, quic-go uses $N = 1.25$ in Equation 6.3 [54]. With a bottleneck bandwidth of 10 Mbit/s, quic-go might calculate an interval of below the 1 ms threshold for the pacer. This could be the reason why quic-go shows ACK-clocking at 10 Mbit/s instead of evenly spread packets in our measurements. Further research is required for a definite answer.

Picoquic can also be configured to use CUBIC instead of BBR. In contrast to other QUIC implementations and TCP, picoquic does not use ACK-clocking for packet pacing but sends almost 91 % of the packets evenly spread as indicated by Figure 6.15a.

Figure 6.15b shows that the client often only sends one *ACK frame* every 150 ms to 300 ms, indicating that the picoquic sender is not ACK-clocked.

Table 6.1 states that an interval of 1.19 ms would be optimal for picoquic, since this is the interval between packets on the bottleneck link. In Figure 6.15a, most packets are sent with an interval of just 0.94 ms. In theory, this results from using a factor of $N = 1.25$ in Equation 6.2.

$$\text{interval} = \frac{\text{frame size}}{N \cdot \text{bandwidth}} = \frac{1482\,\text{B}}{1.25 \cdot 10\,\text{Mbit/s}} \approx 0.95\,\text{ms}$$

Within the source code of picoquic, the pacing rate is calculated by dividing the current congestion window by the calculated smoothed RTT [55].

$$\text{pacing rate} = \frac{\text{congestion window}}{\text{smoothed RTT}}$$

Further research has to be done to answer why picoquic paces packets according to a 25 % higher pacing rate than the bottleneck rate.

Chapter 7

Conclusion

This chapter concludes our results and major contributions. Additionally, we provide an outlook on future work.

7.1 Major Contributions

Our major contributions to the research field around QUIC can be summarized by the answers to our research questions.

To which extend can the pacing mechanism of current QUIC implementations be analyzed?

The open-source QUIC implementations we examined are not optimized for high bandwidths. For the analysis of the pacing mechanism in QUIC, implementations must not be in overload because packets would not be actively paced by the sender but spread because of performance limitations. On bandwidths in the range of 1 Gbit/s, QUIC implementations not implemented in C cannot fully utilize the link with a single stream. A value between 1 Gbit/s and 2 Gbit/s seems to be the maximum throughput the examined QUIC implementations programmed in C can achieve with a single stream. The main reason might be the context switch for each packet sent, which is one reason for a delay of at least around 5 µs between bursted packets. Consequently, the QUIC implementations we examined show a link utilization between 0.47 % and 18.3 % at 10 Gbit/s.

This performance limitation on high bandwidths constrains the evaluation of the pacing mechanism to bandwidths of below 1 Gbit/s until the implementations are further optimized.

For a basic evaluation of the pacing mechanism, a simple measurement setup can be used. Ideally, the setup allows capturing timestamps for sent packets passively. The setup needs a high and low bandwidth link, which functions as the bottleneck. On the high bandwidth link, the intervals between sent packets can be analyzed.

How does the acknowledgment frequency differ between QUIC implementations and TCP?

Between most of the examined QUIC implementations, the acknowledgment mechanism behaves similar to TCP on 10 Mbit/s. In general, the QUIC receiver sends an acknowledgment after two incoming QUIC packets. A quic-go receiver seems to be not waiting for a second packet to arrive if it received a packet introducing a gap in the packet number space due to packet loss. Instead, it seems to send an acknowledgment without delay, similarly to TCP sending duplicate acknowledgments immediately.

On 100 Mbit/s the acknowledgment frequency is more diverse between QUIC implementations and sets apart from TCP.

One outstanding exception is the acknowledgment frequency of the QUIC implementation picoquic. Instead of sending an acknowledgment after a certain amount of received packets, the picoquic receiver sends an acknowledgment once approximately every 8.2 ms on both 10 Mbit/s and 100 Mbit/s if picoquic is configured to use BBR. If picoquic uses CUBIC, the acknowledgment frequency is much lower with an interval of up to 360 ms between acknowledgments.

How important is ACK-clocking for packet pacing in QUIC?

In our measurements, most QUIC senders using loss-based congestion controllers rely on ACK-clocking for packet pacing on bandwidths around 10 Mbit/s. Therefore, the observed pacing behavior between implementations implementing a pacing algorithm does not differ from implementations, which do not implement a pacer, on this bandwidth.

One exception is picoquic. Picoquic implements BBR and also loss-based congestion controllers. In both cases, the sender does not rely on ACK-clocking for packet pacing but sends packets according to a calculated pacing rate.

On higher bandwidths, the relationship between ACK-clocking and the pacing mechanism of QUIC implementations is less tightly coupled and requires further investigation.

How does the pacing mechanism differ between QUIC implementations and TCP?

Since ACK-clocking is the dominant factor for packet pacing in our measurements, TCP and QUIC pace packets similarly in most cases. Our findings indicate that a TCP sender

or receiver might be able to time the sending of a packet up to five times more precisely than a QUIC application.

The QUIC draft proposes a leaky bucket algorithm for a pacer implementation [1]. The bucket fills according to the calculated pacing rate. A perfectly paced sender would send packets evenly spread over time.

Across different QUIC implementations, there are deviating approaches for a pacer implementation. Aioquic uses a similar kind of leaky bucket algorithm, limiting the size of a burst to 16 packets. Quic-go does not evenly spread packets if the pacing rate results in an interval of below 1 ms between packets. Instead, it sends packets in a burst and then waits for 1 ms. Consequently, quic-go tolerates higher bursts on higher bandwidths.

Picoquic implements BBR, which handles packet pacing [27]. At 10 Mbit/s, the observed intervals between packets sent by a picoquic sender match the theoretically calculated values closely. At 100 Mbit/s, the intervals between packets show more clusters than expected. This behavior might be caused by inaccuracies within TBF that we used to limit the bandwidth on the bottleneck. Picoquic applies BBR to individual packets, while TCP paces bursts of two packets according to BBR. If the picoquic sender is configured to use CUBIC, the pacing mechanism uses a leaky bucket algorithm. In contrast to other QUIC implementations and TCP, picoquic sends 91 % packets evenly spread instead of relying on ACK-clocking.

7.2 Future Work

This thesis had a look at the pacing mechanism of four different open-source QUIC implementations in a basic network. More implementations implement a pacing mechanism as well, which can be analyzed. Additionally, the influence on the pacing mechanism of various network conditions, like shared or lossy links, can be investigated.

Our measurements were performed in a static network, which does not change over the lifetime of a QUIC or TCP connection. Changing network conditions, like a new RTT or bottleneck bandwidth, have an influence on the pacing rate. It can be investigated how fast different QUIC implementations can react and how well they can adapt to network changes.

Our results show that in the case of senders using loss-based congestion controllers, ACK-clocking is the primary factor for packet pacing on a bandwidth around 10 Mbit/s. Statements about higher bandwidths are often not possible due to the performance

limitations of the QUIC sender. If implementations get further optimized, the pacing mechanism can be evaluated on higher bandwidths.

Some implemented pacing strategies only show an effect if higher bursts occur during the transmission. In our measurements, we did not generate enough of these large bursts to make well-founded statements about how well these strategies can pace packets. Future work can find techniques forcing a QUIC sender to send higher bursts.

A pacing mechanism is just one optimization for a QUIC sender. This optimization conflicts with GSO, for example, since it does not allow the sender to set individual timestamps for each packet [20]. GSO would improve the throughput on higher bandwidths, at the cost of packet pacing. Pacing can be offloaded to the kernel by configuring the FQ queueing discipline [56]. It can be investigated how well FQ can pace multiple QUIC flows if a single UDP socket is used by multiple QUIC connections.

In our measurements, we used pairs of servers and clients of the same QUIC implementation. We discovered different strategies for the acknowledgment and pacing mechanism. Further research can investigate how these strategies perform if a server and client of different QUIC implementations communicate.

Chapter A

List of acronyms

BDP	Bandwidth-delay product
BtlBw	Bottleneck bandwidth
FQ	Fair Queue
GSO	Generic segmentation offload
HTTP	Hypertext Transfer Protocol
HTTP/2	Hypertext Transfer Protocol version 2
HTTP/3	Hypertext Transfer Protocol version 3
HTTPS	Hypertext Transfer Protocol Secure
IETF	Internet Engineering Task Force
MDS	Maximum datagram size
MSS	Maximum segment size
MTU	Maximum transmission unit
NetEm	Network Emulator
NIC	Network interface card
QUIC	Quick UDP Internet Connections
RTprop	Round-trip propagation time
RTT	Round trip time
TBF	Token-Bucket Filter
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TLS 1.2	Transport Layer Security protocol version 1.2
TLS 1.3	Transport Layer Security protocol version 1.3
TSO	TCP segmentation offload
UDP	User Datagram Protocol

Bibliography

[1] J. Iyengar and I. Swett, “QUIC Loss Detection and Congestion Control”, Internet Engineering Task Force, Internet-Draft draft-ietf-quic-recovery-34, Jan. 2021, Work in Progress, 52 pp. [Online]. Available: `https://datatracker.ietf.org/doc/html/draft-ietf-quic-recovery-34`.

[2] A. Aggarwal, S. Savage, and T. Anderson, “Understanding the Performance of TCP Pacing”, in *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*, vol. 3, 2000, pp. 1157–1165.

[3] M. Belshe, R. Peon, and M. Thomson, *Hypertext Transfer Protocol Version 2 (HTTP/2)*, RFC 7540, May 2015. [Online]. Available: `https://rfc-editor.org/rfc/rfc7540.txt`.

[4] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi, “The QUIC Transport Protocol: Design and Internet-Scale Deployment”, in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM ’17, Los Angeles, CA, USA: Association for Computing Machinery, 2017, 183–196.

[5] J. Iyengar and M. Thomson, “QUIC: A UDP-Based Multiplexed and Secure Transport”, Internet Engineering Task Force, Internet-Draft draft-ietf-quic-transport-34, Jan. 2021, Work in Progress, 207 pp. [Online]. Available: `https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-34`.

[6] M. Bishop, “Hypertext Transfer Protocol Version 3 (HTTP/3)”, Internet Engineering Task Force, Internet-Draft draft-ietf-quic-http-33, Dec. 2020, Work in Progress, 73 pp. [Online]. Available: `https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-33`.

[7] D. Saif, C.-H. Lung, and A. Matrawy, *An Early Benchmark of Quality of Experience Between HTTP/2 and HTTP/3 using Lighthouse*, 2020. arXiv: `2004.01978 [cs.NI]`.

[8] K. Nepomuceno, I. N. d. Oliveira, R. R. Aschoff, D. Bezerra, M. S. Ito, W. Melo, D. Sadok, and G. Szabó, “QUIC and TCP: A Performance Evaluation”, in *2018 IEEE Symposium on Computers and Communications (ISCC)*, 2018, pp. 45–51.

[9] K. Wolsing, J. Rüth, K. Wehrle, and O. Hohlfeld, “A Performance Perspective on Web Optimized Protocol Stacks: TCP+TLS+HTTP/2 vs. QUIC”, in *Proceedings of the Applied Networking Research Workshop*, ser. ANRW ’19, Montreal, Quebec, Canada: Association for Computing Machinery, 2019, 1–7.

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

[11] J. Rüth, K. Wolsing, K. Wehrle, and O. Hohlfeld, “Perceiving QUIC: Do Users Notice or Even Care?”, in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, ser. CoNEXT ’19, Orlando, Florida: Association for Computing Machinery, 2019, 144–150.

[12] Y. Yu, M. Xu, and Y. Yang, “When QUIC meets TCP: an Experimental Study”, in *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, 2017, pp. 1–8.

[13] A. S. Tanenbaum and H. Bos, *Modern operating systems*. Pearson, 2015.

[14] T. Hruby, T. Crivat, H. Bos, and A. S. Tanenbaum, “On Sockets and System Calls: Minimizing Context Switches for the Socket API”, in *2014 Conference on Timely Results in Operating Systems (TRIOS 14)*, Broomfield, CO: USENIX Association, Oct. 2014.

[15] K. Oku and J. Iyengar, *Can QUIC match TCP’s computational efficiency?*, `https://www.fastly.com/blog/measuring-quic-vs-tcp-computational-efficiency`; last accessed on 10.01.2021.

[16] *Transmission Control Protocol*, RFC 793, Sep. 1981. [Online]. Available: `https://rfc-editor.org/rfc/rfc793.txt`.

[17] E. Blanton, D. V. Paxson, and M. Allman, *TCP Congestion Control*, RFC 5681, Sep. 2009. [Online]. Available: `https://rfc-editor.org/rfc/rfc5681.txt`.

[18] V. Jacobson, “Congestion Avoidance and Control”, in *Symposium Proceedings on Communications Architectures and Protocols*, ser. SIGCOMM ’88, Stanford, California, USA: Association for Computing Machinery, 1988, 314–329.

[19] W. Lautenschlaeger, “A Deterministic TCP Bandwidth Sharing Model”, Apr. 2014.

[20] A. Ghedini, *Accelerating UDP packet transmission for QUIC*, https://calendar.perfplanet.com/2019/accelerating-udp-packet-transmission-for-quic/; last accessed on 28.12.2020.

[21] *TCP: Internal implementation for pacing*, https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net-next.git/commit/?id=218af599fa635b107cfe10acf3249c4dfe5e4123; last accessed on 05.01.2021.

[22] *FQ - Fair Queue traffic policing*, https://man7.org/linux/man-pages/man8/tc-fq.8.html; last accessed on 05.01.2021.

[23] T. Henderson and S. Floyd, *The NewReno Modification to TCP's Fast Recovery Algorithm*, RFC 2582, Apr. 1999. [Online]. Available: https://rfc-editor.org/rfc/rfc2582.txt.

[24] 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.

[25] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, and R. Scheffenegger, *CUBIC for Fast Long-Distance Networks*, RFC 8312, Feb. 2018. [Online]. Available: https://rfc-editor.org/rfc/rfc8312.txt.

[26] *TCP: Make cubic the default congestion controller*, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=597811ec167fa01c926a0957a91d9e39baa30e64; last accessed on 17.01.2021.

[27] 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.

[28] *User Datagram Protocol*, RFC 768, Aug. 1980. [Online]. Available: https://rfc-editor.org/rfc/rfc768.txt.

[29] M. Piraux, Q. De Coninck, and O. Bonaventure, "Observing the Evolution of QUIC Implementations", in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, ser. EPIQ'18, Heraklion, Greece: Association for Computing Machinery, 2018, 8–14.

[30] *IETF QUIC Working Group (quic)*, https://datatracker.ietf.org/wg/quic/documents/; last accessed on 28.12.2020.

[31] *Mininet*, http://mininet.org; last accessed on 03.01.2021.

[32] R. Marx, J. Herbots, W. Lamotte, and P. Quax, "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*, ser. EPIQ '20, Virtual Event, USA: Association for Computing Machinery, 2020, 14–20.

[33] *quiche*, https://github.com/cloudflare/quiche; last accessed on 08.02.2021.

[34] W. Almesberger, *Linux network traffic control—implementation overview*, 1999.

[35] S. Hemminger, “Network emulation with NetEm”, in *Linux conf au*, 2005, pp. 18–23.

[36] *tcpdump*, https://www.tcpdump.org; last accessed on 03.01.2021.

[37] D. S. E. Deering and J. Mogul, *Path MTU discovery*, RFC 1191, Nov. 1990. [Online]. Available: https://rfc-editor.org/rfc/rfc1191.txt.

[38] *qlog: QUIC and HTTP/3 logging schema*, https://github.com/quiclog/internet-drafts; last accessed on 30.12.2020.

[39] *Open-source QUIC implementations*, https://github.com/quicwg/base-drafts/wiki/Implementations; last accessed on 30.12.2020.

[40] *ngtcp2*, https://github.com/ngtcp2/ngtcp2; last accessed on 30.12.2020.

[41] *picoquic*, https://github.com/private-octopus/picoquic; last accessed on 30.12.2020.

[42] *netcat*, https://linux.die.net/man/1/nc; last accessed on 30.12.2020.

[43] *Apache HTTP server project*, https://httpd.apache.org; last accessed on 30.12.2020.

[44] E. Rescorla, *HTTP Over TLS*, RFC 2818, May 2000. [Online]. Available: https://rfc-editor.org/rfc/rfc2818.txt.

[45] *curl*, https://curl.se; last accessed on 30.12.2020.

[46] *passive fiber TAPs*, https://www.gigamon.com/content/dam/resource-library/english/white-paper/wp-network-taps-first-step-to-visibility.pdf; last accessed on 04.01.2021.

[47] *Token Bucket Filter queueing discipline*, https://linux.die.net/man/8/tc-tbf; last accessed on 30.12.2020.

[48] D. Scholz, B. Jaeger, L. Schwaighofer, D. Raumer, F. Geyer, and G. Carle, “Towards a Deeper Understanding of TCP BBR Congestion Control”, in *IFIP Networking 2018*, Zurich, Switzerland, May 2018.

[49] J. D. Beshay, A. Francini, and R. Prakash, “On the Fidelity of Single-Machine Network Emulation in Linux”, in *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2015, pp. 19–22.

[50] *MoonSniff*, https://github.com/gallenmu/MoonGen/tree/master/examples/moonsniff; last accessed on 10.01.2021.

[51] *ngtcp2*, https://github.com/ngtcp2/ngtcp2/tree/20d04c3abea00beeef10d9e0bacb8b717504f2ed; last accessed on 02.02.2021.

[52] *Linux kernel BBR implementation*, https://elixir.bootlin.com/linux/v4.19.8/source/net/ipv4/tcp_bbr.c; last accessed on 02.02.2021.

[53] *aioquic*, https://github.com/aiortc/aioquic/tree/04b28d8c632cc62fa4d63cdcce427f0257d00c8a; last accessed on 02.02.2021.

[54] *quic-go*, https://github.com/lucas-clemente/quic-go/tree/eff36f3057c96a60a25d40a72b6eb1d6ff4aa962; last accessed on 02.02.2021.

[55] *picoquic*, https://github.com/private-octopus/picoquic/tree/bf84867d82de93be052dad9d9049c5fc280d3902; last accessed on 02.02.2021.

[56] W. de Bruijn and E. Dumazet, "Optimizing UDP for content delivery: GSO, pacing and zerocopy", in *Linux Plumbers Conference*, 2018.