HAW HAMBURG

MASTER THESIS
Lukas Hettwer

Evaluation of QUIC-Tunneling

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Lukas Hettwer

Evaluation of QUIC-Tunneling

Master thesis submitted for examination in Master´s degree
in the study course *Master of Science Informatik*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at University of Applied Science Hamburg

Supervisor: Prof. Dr. Martin Becke
Supervisor: Prof. Dr. Klaus-Peter Kossakowski

Submitted on: 15. September 2022

Lukas Hettwer

Thema der Arbeit

Evaluierung von QUIC-Tunneling

Stichworte

QUIC, QUIC-Tunneling Transportprotokolle, Datagramm, Internetverkehr

Kurzzusammenfassung

Seit 2021 ist QUIC offiziell standardisiert. Aufgrund des zunehmenden Interesse der führenden IT-Unternehmen wächst der Anteil der Nutzung von QUIC innerhalb des Internetverkehrs. Derzeit werden weitere Anwendungsszenarien, die QUIC einbeziehen, entwickelt, diskutiert und gegebenenfalls zusätzlich standardisiert. Besondere Aufmerksamkeit gilt dabei dem Einsatz von QUIC zusammen mit der Datagramm-Erweiterung zur Bereitstellung eines IP-Tunnels. Eine der möglichen Anwendungsbereiche ist es, veraltete Software durch QUIC-Tunneling mit moderner Technik zu betreiben.

In dieser Arbeit wird das Verhalten von TCP-Verbindungen innerhalb eines QUIC-Tunnels untersucht. Zum Aufbau eines QUIC-Tunnels wird eine Client/Server Anwendung entsprechend den RFCs der IETF-Gruppe masque implementiert. Eine experimentelle Umgebung mit drei Containern ist entwickelt, um eine QUIC-Tunnelverbindung aufzubauen. Unter Verwendung von iPerf3 werden TCP-Streams mit Congestion Control Algorithmen wie BIC, CUBIC, Vegas, BBR, Reno und Westwood übertragen. Die Parameter der TCP-Streams sind in dieser Arbeit erfasst und ausgewertet. Des Weiteren werden mehrere TCP-Streams zeitlich parallel über den QUIC-Tunnel übertragen und auf Fairness geprüft.

Die Übertragung von TCP durch einen QUIC-Tunnel gelang ohne jegliche Störung. Die Analyse identifizierte Engpässe wie die TUN-Empfangs-Warteschlange, den Datagramm-Sendebuffer sowie den UDP-Empfangsbuffer innerhalb der Tunnelanwendung. Es wurden keine erwähnenswerte Ungleichheiten hinsichtlich Fairness beobachtet.

Lukas Hettwer

Title of Thesis

Evaluation of QUIC-Tunneling

Keywords

QUIC, QUIC-tunneling, transport protocols, datagram, network traffic

Abstract

In 2021, the work on the standardisation of QUIC is completed. Due to the increasing interest of the leading IT companies, the share of the use of QUIC within the internet traffic is growing. Currently, further application scenarios that involve QUIC are being developed, discussed and, if necessary, standardised. Special attention is given to the use of QUIC together with datagram extension to provide an IP tunnel. One of the possible implementation is to operate outdated applications by QUIC-tunneling with modern technology.

This paper investigates the behaviour of TCP connections within a QUIC tunnel. To establish a QUIC tunnel, a client/server application is implemented according to the RFCs of the IETF group masque. An experimental environment with three containers is developed to establish a QUIC tunnel connection. By using iPerf3, TCP streams with congestion control algorithms as BIC, CUBIC, Vegas, BBR, Reno and Westwood are transmitted. The parameters of the TCP streams are captured and evaluated in this paper. Further, investigations of several TCP streams, which are transmitted simultaneously via the QUIC tunnel, are assessed for fairness.

The transmission of TCP through a QUIC tunnel succeeded without any interference. The analysis identified bottle necks such as TUN receive queue, datagram send buffer as well as UDP receive buffer within the tunnel application. No particular unfairness was observed.

Contents

List of Figures

List of Tables

Abbreviations

1-RTT One Round Trip Time.

ACK Acknowledgement.

AIMD Additive increase/multiplicative decrease.

BBR Bottleneck Bandwidth and Round-trip propagation time.

BIC Binary Increase Congestion Control.

cwnd Congestion Window.

DTLS Datagram Transport Layer Security.

DupACK Duplicate Acknowledgment.

ECN Explicit Congestion Notification.

FIFO First In First Out.

HTTP Hypertext Transfer Protocol.

I/O Input/Output.

ID Identifikator.

IETF Internet Engineering Task Force.

IP Internetprotokoll.

IPsec Internet Protocol Security.

IPv4 Internet Protocol Version 4.

IPv6 Internet Protocol Version 6.

L2TP Layer 2 Tunneling Protocol.

MACsec Media Access Control Security.

masque Multiplexed Application Substrate over QUIC Encryption.

MTU Maximum Transmission Unit.

NAT Network Address Translation.

PCAP Packet Capture.

QoS Quality of Service.

RFC Request for Comments.

RTO Retransmission Timeout.

RTT Round Trip Time.

SACK Selective Acknowledgment.

SCTP Stream Control Transmission Protocol.

ssthresh Slow Start Threshold.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

TUN Network TUNnel.

TX Transmit.

UDP User Datagram Protocol.

VPN Virtual Private Network.

WAN Wide Area Network.

1 Introduction

In recent years, the focus has been increasingly set on online data transmission. With that, the need for security in particular has grown and received more attention. In general, to transport data network protocols are crucial. In the past, Transmission Control Protocol (TCP) with Transport Layer Security (TLS) was mainly used for this purpose. Based on the research of the Stream Control Transmission Protocol (SCTP), first Google and now the Internet Engineering Task Force (IETF) built the general-purpose transport layer network protocol QUIC. It is defined as a universal application/transport layer protocol. QUIC is standardised by the IETF since May 2021. This protocol is based on the User Datagram Protocol (UDP) and is connection-oriented. QUIC is characterised and dependent of streams in a connection between two endpoints to transmit multiple messages parallel. This structure prevents head-of-line blocking, since packets with different information are not held up by a blocked packet. It combines the handshake phase of TCP and TLS, which is defined by two to three Round Trip Time (RTT) into one (1-RTT). These two features are intended to reduce the number of connections as well as the transport latency of connection-oriented web applications. QUIC provides a separate identifier for each connection, which maintains the connection even through potential network interruption. Thus, QUIC has its benefit in reliability.

Since UDP is usually integrated within the operating system it can be implemented straight into applications. Due to that fact, no changes are necessary to UDP by QUIC and it runs in the user space. Each software provides its own QUIC implementation - therefore there is also no need for any changes to the kernel in case of QUIC updates. The move of QUIC into the user space enables the fast development of its individual components.

These ongoing improvements and the strong support from Google, Cloudflare, Facebook and Akamai lead to 7.6% [12] of all websites already using QUIC (September 2022). It is a possible scenario that QUIC replaces TCP in the future, and then networks could move from TCP/IP optimisation to UDP/IP, until there are networks that only support

UDP/IP. Looking at the transition from Internet Protocol Version 4 (IPv4) to Internet Protocol Version 6 (IPv6), this is not expected to actually happen in near future. On the other hand, past experience shows that infrequent UDP traffic has led to networks that still block UDP traffic through Network Address Translation (NAT) and strict firewalls. If almost only UDP/IP traffic produced by QUIC exists, it may mean that future networks are less likely to care about supporting TCP/IP traffic. Furthermore, if the resources invested in TCP development are reduced, older applications that rely on TCP and TLS will no longer be able to perform secure data transfers. There is, of course, the option of upgrading these applications to the latest technology. Another possibility is to transport the TCP/IP packets of the legacy application through incompatible networks with the help of tunnel software. The latest extension to QUIC also offers a solution for this.

The Request for Comments (RFC) 9221 "An Unreliable Datagram Extension to QUIC" adds an unreliable datagram extension to QUIC, which allows to receive and send data over a QUIC connection. The RFC describes the two possible use cases for the datagram: the first use case is to send real time data and the second use case is to implement an IP packet tunnel over QUIC. [10]

The working group Multiplexed Application Substrate over QUIC Encryption (masque) has developed the draft "IP Proxying Support for HTTP". The methods for proxying IP packets over HTTP that use the unreliable datagram extension is described.

This paper outlines TCP congestion control and the problem of tunneling TCP connections, the proposals of the masque group and a possible implementation of these. The main focus is to evaluate the behaviour of TCP connections transmitted between two networks using a QUIC connection with the unreliable datagram extension. Therefore, an experiment with the implementation will be set up to investigate how the TCP connections react. In Summary, the purpose of this paper is to clarify whether QUIC is capable of tunneling TCP connections.

1.1 Related Work

The paper "A Comparative Study on Virtual Private Networks for Future Industrial Communication Systems" examines and compares various Virtual Private Network (VPN) solutions. The motivation of the authors is to transform old factory networks, developed and maintained on the concept of perimeter security, into smart factories through VPN

solutions. The connection method should be integrated transparently without the need to configure the old machines. Gateways with a VPN solution ate to be applied, which bridges between the legacy layer 2 Ethernet of the machines and the modular layer 3 network of the smart factories. The study examines various software VPN solutions for their suitability for this task. For this purpose, the non-functional aspects of security, manageability and usability as well as performance are evaluated. The authors use several hardware platforms, ranging from very resource-constrained to very powerful. The performance parameters measured were throughput and latency over the secure tunnels provided by the VPN. The throughput measurements produced a single value and the latency measurements are performed for a variety of different frame sizes up to the standard Ethernet maximum frame size. The authors give a clear recommendation for the software VPN solutions Media Access Control Security (MACsec) and WireGuard based on their results in terms of manageability, security and performance. [7] The demands made on the connection method by the legacy environment of the old machines serves as the basis for the motivation of this work. Unlike in the paper, it is not possible to choose between different solutions based on QUIC. Nevertheless, the referenced paper shows that old factory with outdated applications and network cannot remain operational in the future without spending on upgrades. In this case, the cost of a tunnel was chosen over the cost of updates.

The paper "Performance Evaluation and Analysis of Layer 3 Tunneling between OpenSSH and OpenVPN in a Wide Area Network Environment" examines performance and efficiency between OpenVPN and OpenSSH VPNs in over Wide Area Network (WAN) connections. For the investigation of the tunneling software an experiment setup is developed. This consists of a server which is connected to two clients via the internet. The client connects to the server with the appropriate VPN software and transfers files from 1 MB to 10 MB in size using *iPerf3* over UDP. The test records three parameters -namely: speed, latency and jitter. The authors assume that this real-life scenario will provide more accurate values to evaluate and analyse the performance. The experiments conducted in this paper demonstrate that OpenSSH is faster than OpenVPN in a scale of 1 MB to 10 MB. Results of the analysis between OpenSSH and OpenVPN show that OpenSSH utilizes better the link and significantly improves transfer times. [4] The type of measurement from the paper was adopted for this work along with the measurement tools used such as *iPerf3*. However, this work focuses on measurements in a closed system to avoid errors due to external effects such as on the internet.

The paper "Experimental Performance Comparison between TCP vs. UDP tunnel using OpenVPN" investigates the performance between TCP and UDP tunneling. The authors assume that tunneling TCP into TCP causes problems due to retransmission, TCP meltdown and double retransmit and that this causes increased latency and increased bandwidth consumption. An OpenVPN tunnel is established in an experimental setup with a client and a server connected via a switch and Ethernet cable. For this, OpenVPN is configured with a UDP and a TCP tunnel. With *iPerf3* TCP and UDP traffic is generated and measured. The focus is on latency for different message sizes. The results conclude that UDP tunnel utilizes the link more efficiently and provide a radically improved transfer times and speed compared with TCP tunnel. The results also demonstrates that TCP in UDP tunnel provides better latency. [3] This work illustrates once again why a tunnel based on UDP is so much more relevant than on TCP and why the implementation must be done with the QUIC datagram extension. According to the results, an implementation without the extension would probably lead to similar results.

1.2 Structure of the Work

The section 2.1 describes the congestion control mechanisms of TCP and the importance of this module for successful transmission. Latter section describes the relevance of TCP friendliness and the problems of tunneling TCP connections. This chapter provides a general understanding of the impact on the transport layer by the components and actors below the transport layer.

The RFCs of the masque IETF group and the RFC of the datagram extension of QUIC are discussed in the section 2.2. The technologies used, such as the QUIC libraries, and details of the implementation of the RFCs are explained in the section 2.3. This chapter describes the preparations and decisions that are taken before a TCP packet could be transported through a QUIC tunnel.

The description of the experiment setup and the used measurement tools are in the chapter 2.4. The results of the measurements are presented in the chapter 2.5. The conclusion and proposals for future works is described in the final chapter 3.

1.3 Scope of Work

This study is unable to encompass the entire behaviour of other types of data flows than TCP through the QUIC tunnel. The impact of data streams inside the tunnel on those outside is not investigated. The behaviour of the tunnel with different physical layers is not part of this research. Further, a comparison between different tunnel software like WireGuard or OpenVPN is not carried out. However, the source code of the user space implementation of WireGuard in Rust is used as a template.

The reader should bear in mind that the produced tunneling software is not suitable for use in productive systems. Due to practical constraints, this paper cannot provide a comprehensive implementation of the RFCs. The implementation attempts to comply with the RFCs. If parts have not given an additional value for this explicit measurement, they have been omitted for this experiment.

2 Concept, Implementation and Evaluation of QUIC-Tunneling

2.1 Congestion Control and Problem of Stacked TCP Connections

To get an overview of possible issues which can occur in tunnels this chapter discusses the basics of TCP congestion and flow control. This subchapter will be structured as follows: The need for congestion control in networks as well as general TCP congestion control is discussed first. Then, a selection of different variants of congestion control algorithms is introduced. In the next step, the definition of TCP friendliness is described. Finally, the details of tunnels and VPN are addressed and an overview of the problems of stacked TCP connections is provided.

2.1.1 TCP Congestion Control

The term congestion is being used when a resource demand exceeds its capacity. Internet performance is largely governed by the inevitable natural fluctuations caused by the coming and going of different data streams. If the capacity is exceeded, packets cannot be transferred across the link, there are only two options for these packets: buffer them or drop them. To handle temporary traffic peaks, standard internet routers store excess packets in a First In First Out (FIFO) queue. Once the queue has reached its maximum, the packets must be dropped. If the subsequent traffic is reduced, the queue can be drained. The queue serves an ample device for compensating for short traffic bursts. In general, however, it is not a good choice to increase the queue to absorb any traffic peaks. [15]

There are two basic problems:

1. Storing packets in the queue adds a significant delay that increases proportionally to the length.
2. Internal traffic does not follow a constant average rate independently over a fixed time interval, so the assumption that there are as many upward as downward fluctuations may be incorrect.

Therefore, queues should generally be as short as possible. By increasing the queue, the network takes a higher risk of being congested which is resulting in an increased delay. Thinking further this scenario, it could lead to a packet loss in the worst case. The goal of congestion control mechanisms is to use the network as efficiently as possible to reach the highest possible throughput while maintaining a low loss rate and low delay. A congestion control mechanism depends on the implicit feedback of the network. It should not depend on control and/or measurement actions from the network. From the end-node perspective, there are basically three possibilities for what can happen with the packet: Either it can be delayed, dropped or changed. [15]

In a congestion network, the queues on the path fill up and are responsible for two of the three properties. First, the packet is delayed, and when the queue fills up, it is dropped. A congestion control algorithm can use these factors 'delay' and 'packet dropped' to determine when the network is congested. Not every delayed or dropped packet is a sign of a congested network. If the packet is transmitted to the destination by a different path, the different RTT can be misinterpreted as a delay. If an error-prone transmission medium is used, such as wireless, a lost packet can be misinterpreted as a congestion. [15]

The reliability of TCP is based on the fact that segments are sent and their arrival is confirmed by an Acknowledgement (ACK). The sender must wait a certain time for the confirmation by an ACK. The determination of the waiting time is of critical importance. If the time is too short, a Retransmission Timeout (RTO) will be triggered and cause packets to be mistakenly resent that are still in flight or the ACK has not yet arrived. If the time is too long, a packet that has actually been lost is inefficiently not detected for a long time. Not only is the determination of the RTO crucial, but it must also be developed dynamically in order to react to what is happening in the network. When calculating RTO, it is crucial to obtain a good estimate of RTT. [15]

TCP uses two methods based on the end-to-end response to limit the amount of data that can be sent. The first method is Slow Start, which starts the ACK clock to quickly reach a reasonable rate. This manages the Congestion Window (cwnd) in addition to the window maintained by the sender. The minimum of the two windows allows the amount of data that can be sent. The cwnd is initialised and increased by one segment for each incoming ACK. The RTO timer detects when a packet is lost. This is interpreted as an implicit congestion feedback signal and the cwnd is reset to one. [15]

The second algorithm, called congestion avoidance, does not increase cwnd by one for each ACK, but normally increases by at most one segment per RTT. The algorithms slow start with the exponential increase and congestion avoidance with the additive increase are merged by a threshold variable Slow Start Threshold (ssthresh). When receiving an ACK, cwnd is increased by one segment if cwnd is below the threshold, otherwise additive at each RTT. Should an RTO occur, then cwnd is set to the minimum value and the ssthresh is set to half the current window size. [15]

Congestion control is extended by two algorithms on the sender side, which are designed to correct poor performance in long fat pipes. If a packet does not arrive at the receiver but the following packets in the sequence, the receiver responds with ACKs containing the highest sequence number without interruption. These ACKs are called Duplicate Acknowledgment (DupACK) and are an indication of packet loss, reordering or duplication. Since reordering can happen, the first mechanism should be used when the threshold of three DupACKs (four identical ACKs) is reached. [15]

The sender does not set cwnd to the minimum value to continue with Slow Start, but sets ssthresh to half of the packets in transit, usually the half of cwnd, and increases cwnd by the number of received DupACKs. The cwnd must be increased even though a loss has been detected, because received packets that have been taken from the network cannot yet be subtracted from the cwnd, as they have not been explicitly acknowledged. This allows to send additional packets. [15]

If new packets are acknowledged, fast recovery is replaced by setting cwnd with the new value of ssthresh by the congestion avoidance mechanism. The new packets need to be confirmed by another ACK before the RTO is triggered - otherwise the system switches to the Slow Start mechanism. However, the RTO mechanism is still considered as a backup mechanism that only invoke when everything else fails. [15]

The four mechanism can be found in TCP Reno and NewReno and are the basic definition of the standard behavior of TCP. Building up on these four mechanisms, there are extensions such as Selective Acknowledgment (SACK), Explicit Congestion Notification (ECN) or Timestamp option, which give more precise information about the (implicit) signals. [15]

In the following sections additional congestion control algorithms are described that are based on the four mechanisms.

TCP BIC

There are refined versions that implement other strategies. One of them, called Binary Increase Congestion Control (BIC), increases its rate like a normal TCP sender up to a predefined threshold. When this is passed, it continues in steps of a fixed size. As soon as a typical TCP congestion event such as packet loss occurs, a binary search strategy is used to find the new size between the maximum (the window where the packet loss occurred) and the minimum (the new window). The new window is assumed to be the new minimum window, and if another packet loss occurs, then this window is taken as the maximum. This process is repeated until the update steps are smaller than a predefined threshold and the process converges. [15]

TCP CUBIC

The CUBIC versions is a refinement of BIC. The main feature is the used growth function. This is a cubic function that takes as input parameters the maximum window size (from normal BIC), a constant scaling factor, a constant multiplicative reduction factor and the time since the last drop event. This preserves the strengths of BIC, increases TCP-friendliness, simplifies window control and is not dependent on RTT. The result of the function depends on the time since the last loss. [15]

TCP Westwood

The TCP Westwood a optimised version changes the response function so that it does not respond in a fixed way, but in relation to the current state of the system. It does this by monitoring the rate of incoming ACKs to determine the actual rate of packets

arriving at the receiver. The product of the estimated bandwidth and the minimum RTT determines the update of ssthresh in the event of a loss. This results in a more drastic reduction in the case of severe congestion compared to a minor congestion. In the case of faulty networks, such as wireless networks, this variant works particularly well, since no drastic reduction follows in the case of a loss due to corruption. [15]

TCP Vegas

The TCP specification does not forbid sending less than the congestion window. The TCP Vegas variant takes advantage of this. It determines the minimum RTT and uses cwnd to calculate the expected throughput. The actual throughput is calculated by counting the segments sent within an RTT. Ideally, both values should match or their difference should not exceed a threshold. If the difference is greater, the transmission rate must be adjusted accordingly. This mechanism should converge to a stable operating point, which is a significant difference between TCP Vegas and TCP Reno. In addition, Vegas is able to detect and react to incipient congestion at an early stage. [15]

TCP BBR

The Bottleneck Bandwidth and Round-trip propagation time (BBR) variant developed by Google is similar to TCP Vegas in that it also attempts to sustainably relieve the queue. It relies on all RTT measurements within a time window to determine the minimum RTT and on congestion capacity measurement using the correlation of the data stream with the ACK stream. Unlike Vegas, BBR actively defends displacement by conventional Additive increase/multiplicative decrease (AIMD) protocols. However, the behaviour of flow matching in BBR differs significantly from TCP Vegas. [5]

BBR's approach is based on increasing the flow by a factor of 0.25 at regular intervals, leading to the formation of a queue at the bottleneck. Through the RTT and bottleneck capacity measurement, BBR determines this effect. To empty the buffers again, it over-corrects to a reduced transmission rate. If the testing increases the available bandwidth, BBR corrects exponentially, unlike Vegas' linear correction. [5]

2.1.2 TCP Friendliness

The complex definition of fairness is followed by a more pragmatic approach in the network. Most streams are TCP streams that are controlled by the same rules of congestion control. Flows that do not react similarly cause great damage by pushing the TCP flows away. Therefore, the common definition of fairness are TCP friendliness flows or TCP compatible flows if they do not displace TCP flows. [5]

> "A TCP-compatible flow is responsive to congestion notification, and in steady-state it uses no more bandwidth than a conformant TCP running under comparable conditions (drop rate, RTT, MTU, etc.)" [2]

2.1.3 Tunnels and Virtual Private Networks

Tunnels are a method of transmitting data with incompatible address spaces or even incompatible protocols over a network. The tunnel carries the traffic data in the payload of a protocol supported by the underlying network. Normally, the tunneling protocol operates at the same layer or higher than the payload protocol. Example of Tunnels are L2TP (Protocol 115): Layer 2 Tunneling Protocol and IP in IP (Protocol 4): IP in IPv4/IPv6.

The VPNs using encrypted tunnels, which means that within the tunnel the payload is encrypted. This connects private networks via a public network and enables private network communication from one network to another. It creates a secure connection between two machines, a machine and a network or two networks. Example of VPN software are IPsec, WireGuard, OpenVPN and OpenSSH.

2.1.4 TCP Meltdown, Retransmission Problems and Double Retransmits

Sending TCP traffic over a TCP tunnel will force the algorithms of both TCP connections to work in parallel. The designers of the TCP protocol did not address the problem of TCP running within itself. TCP was not designed to work this way and problems are likely to occur in different situations. The protocol is meant to be reliable and uses adaptive timeouts to decide when a resend should occur. The retransmission problems, network slowdown is known as a TCP meltdown and double retransmit, are problems

caused by having two TCP connections stacked together. These problems can occur when the stacked connections have to retransmitting packets. TCP as a tunnel should only be used when restricted networks prevent other solutions.

Instead of TCP as tunnel protocol UDP is more recommended. UDP is not reliable and does not use timeouts to start a retransmission. The data protocol is solely responsible for the loss. In the case of a TCP connection, a loss triggers a timeout in the TCP connection at the sender side. For data such as real time traffic that does not rely on TCP but on UDP, retransmission through a TCP tunnel is unnecessary and consumes bandwidth that may be needed. In this case a UDP tunnel is more appropriate than a TCP tunnel.

2.2 Required RFC for QUIC-Tunneling

In this thesis, three IETF RFC are implemented to establish a tunnel with QUIC. The following subchapter discusses the RFC 9221 with the title of "Unreliable Datagram Extension to QUIC". Further also the "HTTP Datagrams and the Capsule Protocol' and "IP Proxying Support for HTTP" drafts of the working group masque are elaborated.

2.2.1 Unreliable Datagram Extension to QUIC

QUIC provides a secure, multiplexed connection for transmitting reliable streams of application data. The RFC 9221 "Unreliable Datagram Extension to QUIC" provides support of sending and receiving unreliable datagrams. The methodology of the QUIC extension contains different types of frames than in the original RFC to transmit data. Each frame type determines whether this should be retransmitted or not. If the frame type is a unreliable datagrams it must not be retransmitted. [10]

Most applications that need to transmit real-time data prefer unreliable data transmission. These applications are built directly on UDP and use Datagram Transport Layer Security (DTLS) for a secure transport. If reliable data should be transmitted, often another connection with TCP/TLS is used to transmit the data. The unreliable datagram extension provides another option for secure datagrams where the benefit of sharing the cryptographic and authentication is within one connection. [10]

Using unreliable data via QUIC, it offers advantages over existing solutions. Through a single QUIC connection there is a possibility that a reliable stream and an unreliable flow are transmitted. In this case a single handshake and authentication context is shared. This can reduce the latency required for handshakes compared to opening both a TLS connection and a DTLS connection. Compared to the DTLS handshake, QUIC implements a more nuanced loss recovery mechanism, that allows faster loss recovery. A shared connection also allows a single congestion control to be used for both reliable and unreliable data - which is more effective and efficient. [10]

The unreliable datagram extension along with the benefits can be helpful to optimise audio/video streaming, gaming or other real-time networking applications. This extension can also be used to implement an IP packet tunnel over QUIC. Internet-layer tunneling protocols often require a reliable and authenticated handshake followed by unreliable secure transmission of IP packets. This can require a TLS connection for the control data and DTLS for tunneling IP packets. QUIC connection with the unreliable datagrams could support both parts. [10]

The extension introduces two new datagram QUIC frame types that carry application data without requiring retransmissions. The frame types are described in the Listing 2.1. The difference between the types is the presence of the length field. [10]

Listing 2.1: Datagram Frame Format

```
DATAGRAM Frame {
  Type (i) = 0x30..0x31,
  [Length (i)],
  Datagram Data (..),
}
```

When the application sends a datagram over a QUIC connection, QUIC must create a datagram frame and sends it with the first available packet. The QUIC endpoint should immediately forward a received valid datagram to the application, if it is able to process the frame and can store the contents in memory. Datagram frames may be dropped by the receiver if it cannot process them due to lack of memory. If datagram frames are lost, they are not sent again. However, the loss of this is recognisable, because the frames are ack-eliciting. The datagram is not affected by explicit flow control signaling and do not contribute to any per-flow or connection-wide data limit. This also means that there is a risk that the receiver does not have enough resources to process the frame. Within

the process there is possibility that a connection is unable to send a datagram frame until the congestion controller allows it. The datagrams are also affected by this. As a consequence, the sender must either delay or drop the packet. [10]

2.2.2 HTTP Datagrams and the Capsule Protocol

The Internet draft "HTTP Datagrams and the Capsule Protocol" specifies the HTTP datagrams format. This allows the transfer of multiplexed, unreliable datagrams inside an HTTP connection. The draft is compatible with all versions of HTTP. The main focus is the natively use of this format in HTTP/3, together with the QUIC datagram extension. If QUIC datagram extension is unavailable, the document describes the Capsule Protocol, which can be implemented in all versions. This protocol allows a more general convention for transmitting data in HTTP connections. [13]

The HTTP datagram format is intended for use by HTTP extensions (such as the CONNECT method). It is associated with HTTP requests and not part of message content. In HTTP/3, the datagram data field of QUIC datagrams frames uses the format in the Listing 2.2. The endpoint indicates to its peer that it is ready to receive HTTP/3 datagrams with the *SETTINGS_H3_DATAGRAM (0x33)* setting entry. In cases where the HTTP version uses a transport protocol that only allows reliable delivery, the datagrams should be sent using the Capsule Protocol. [13]

Listing 2.2: HTTP/3 Datagram Format

```
HTTP/3 Datagram {
  Quarter Stream ID (i),
  HTTP Datagram Payload (..),
}
```

The Capsule Protocol is a sequence of type-length-value tuples which allows endpoints to reliably exchange request-related information. In the draft its mainly described to exchange HTTP datagrams on end-to-end HTTP request streams when HTTP does not support the QUIC datagrams frame. It can also be used to exchange reliable and bidirectional control messages. This protocol can be transmitted in all versions of HTTP in the data stream. In HTTP/1.x, the data stream consists of all bytes on the connection that follow the blank line that concludes either the request header section, or the response header section. In HTTP/2 and HTTP/3, the data stream of a given HTTP request

consists of all bytes sent in data frames with the corresponding stream Identifikator (ID). [13]

The endpoint uses a Capsule Protocol header field with a true value to signal that it is being used on a data stream. The contents of the associated request's data stream uses the format in the Listings 2.3 and 2.4. [13]

Listing 2.3: Capsule Protocol Stream Format

```
Capsule Protocol {
  Capsule (..) ...,
}
```

Listing 2.4: Capsule Format

```
Capsule {
  Capsule Type (i),
  Capsule Length (i),
  Capsule Value (..),
}
```

The Capsule Protocol is not used unless the response contains a 2xx status code which stands for successful. Since the approach of the Capsule Protocol is to transmit the capsule over the data stream, the associated HTTP request and response do not carry HTTP content. [13]

To send datagrams using the Capsule Protocol the draft defines the first Capsule type *Datagram (0x00)*. The Listing 2.5 contains the mentioned format. This allows HTTP datagrams to be sent on a stream using the Capsule Protocol. Through this extension an unreliable version of the CONNECT method can be implemented. [13]

Listing 2.5: Datagram Capsule Format

```
Datagram Capsule {
  Type (i) = 0x00,
  Length (i),
  HTTP Datagram Payload (..),
}
```

2.2.3 IP Proxying Support for HTTP

The draft paper "IP Proxying Support for HTTP" describes the CONNECT-IP protocol which allows endpoints to create a tunnel to forward IP packets through an HTTP proxy. To perform this it relies on the HTTP datagram support for efficient sending IP packets. CONNECT-IP can be used for general-purpose packet tunnelling, such as for a point-to-point or point-to-network VPN, or for limited forwarding of packets to specific hosts. [11]

To allow endpoints to exchange IP configuration information with each other, the draft defines multiple new capsule types. [11]

The *Address Assign* capsule type described in the Listing 2.6, allows an endpoint to assign an IP address or prefix to its peer. By that, the endpoint can indicate a preference for the IP address or prefix from its peer. Multiple *Address Assign* capsules can be sent, especially necessary when assigning both IPv4 and IPv6 addresses. [11]

Listing 2.6: Address Assign Capsule Format

```
Address Assign Capsule {
  Type (i) = 0xfff100,
  Length (i),
  IP Version (8),
  IP Address (32..128),
  IP Prefix Length (8),
}
```

The *Address Request* capsule type described in the Listing 2.7 allows an endpoint to request an IP address or prefix from its peer. This allows the endpoint to indicate a preference for the IP address or prefix from its peer. For simple client/proxy communication this type is not necessary. [11]

Listing 2.7: Address Request Capsule Format

```
Address Request Capsule {
  Type (i) = 0xfff101,
  Length (i),
  IP Version (8),
  IP Address (32..128),
  IP Prefix Length (8),
}
```

The *Route Advertisement* capsule type, described in the Listings 2.8 and 2.9, allows an endpoint to announce to its peer which route will be used to route traffic. The packet has a set of IP address ranges which indicates that the sender has existing routes. The Route Advertisement capsule type receiver sends its IP packets for one of these ranges to its peer, which forwards them along its routes. [11]

Listing 2.8: Route Advertisement Capsule Format

```
Route Advertisement Capsule {
  Type (i) = 0xfff102,
  Length (i),
  IP Address Range (..) ...,
}
```

Listing 2.9: IP Address Range Format

```
IP Address Range {
  IP Version (8),
  Start IP Address (32..128),
  End IP Address (32..128),
  IP Protocol (8),
}
```

The HTTP Datagram Payload field has the format defined in Listing 2.10. The draft allows different semantics of IP packets in the payload and identifies them by the context ID. If the context ID is set to zero, it means that the payload field contains a full IP packet. [11]

Listing 2.10: IP Proxying HTTP Datagram Format

```
IP Proxying HTTP Datagram Payload {
  Context ID (i),
  Payload (..),
}
```

The endpoint that receives an HTTP datagram containing an IP packet checks the IP header and performs any local policy checks. It then checks its routing table to send the IP packet over the correct outbound interface. [11]

An endpoint that receives an IP packet checks that the packet matches the advertisement routes. If a route exists, then the IP packet is transmitted within an HTTP datagram through the forwarding tunnel. To prevent an infinite loop due to the existence of routing loop, the endpoint reduces the hop count beforehand. [11]

2.3 Implementation Details

This chapter covers key elements of the implementation of a client/server application that connects two private networks with a QUIC connection using the unreliable datagram extension. First, the communication between the application and the operating system network stack is discussed. This is followed by a general description of the QUIC implementation used together with the unreliable datagram extension. To exchange control data between server and client, the HTTP/3 implementation is described in the section 2.3.3. The section 2.3.4 covers informations about the used concurrent programming model.

2.3.1 Virtual Network Device

The TUN (derived from *TUNnel*) devices of the Linux kernel is used as the virtual network device. This network device is completely supported in software and simulates a network layer device which is then mainly used for tunnelling purposes. It works on layer 3 and transports IP packets. Packets sent by the operating system via the TUN device are read by a user space program which attaches itself to the device. Vice versa, this

means that packets written to the TUN device by the user-space program are delivered to the operating system's network stack. [6]

The virtual network device is used by the client and server to read and write IP packets. The TUN devices are configured to cover the IP namespace range of the virtual private network. This means that the network stack of the operating system routes all connections with the IP address from this range via the TUN devices.

The Listing 2.11 shows how the application creates the TUN device. The clone device is opened and due to its support of the system call *ioctl(2)* and the request object *Ifreq* the TUN device is created. With the raw file descriptor the application can read and write IP packets. But the TUN device is down and the IP address and routing must be configured.

Listing 2.11: Creation of a Virtual Network Device

```
const TUNSETIFF: u64 = 0x4004_54ca;
const CLONE_DEVICE_PATH: &[u8] = b"/dev/net/tun\0";

let name = b"tun";

// construct request struct
let mut req = Ifreq {
  name: [0u8; libc::IFNAMSIZ],
  flags: (libc::IFF_TUN | libc::IFF_NO_PI) as c_short,
  _pad: [0u8; 64],
};
req.name[..name.len()].copy_from_slice(name);

// open clone device
let fd: RawFd = match unsafe {
  libc::open(CLONE_DEVICE_PATH.as_ptr() as _, libc::O_RDWR)
} {
  -1 => return Err(TunError(
    "Failed to obtain fd for clone device",
  )),
  fd => fd,
};
assert!(fd >= 0);

// create TUN device
if unsafe { libc::ioctl(fd, TUNSETIFF as _, &req) } < 0 {
  return Err(TunError(
    "set iff ioctl failed (insufficient permissions?)",
  ));
}

let file = unsafe { File::from_raw_fd(fd) };
```

The *Netlink* socket, a Linux kernel interface used for inter-process communication between the kernel and user space, is used to configure the TUN device with IP addresses and routes. In this paper it is used to configure the TUN devices with the data from the Capsule protocol.

Once the TUN device is created and configured, the user space program can read and write IP packets from the device. The Listing 2.12 shows how to read/receive and write/send IP packets in three lines.

Listing 2.12: Read and write from TUN device

```
let mut f = tokio::fs::File::from_std(file);

let mut buffer = [0; 1500];
let n = f.read(&mut buf).await? == 0 {
println!("The bytes: {:?}", &buffer[..n]);

let n = f.write(&payload).await?;
```

2.3.2 QUIC Connection

The QUIC connection between client and server is handled by the crate *Quinn*. *Quinn* is a pure-rust, async-compatible implementation of the QUIC transport protocol. The main reason *Quinn* was used is that it contains the unreliable datagram extension. *Quinn* offers the possibility to skip the verification of the server certification, which reduces the complexity of the local experiment setup. [9]

Listing 2.13: Build a *Quinn* endpoint connection

```
let conn = endpoint.connect(server_addr(), SERVER_NAME)?;
let quinn::NewConnection {
    connection,
    mut datagrams,
    ..
} = conn.await?;

while let Some(payload) = datagrams.next().await {
  println!("The bytes: {:?}", payload?);
}

connection.send_datagram(payload).unwrap();
```

The Listing 2.13 shows a while loop in which datagrams are received from a previously initialised connection and represents how datagram packets can be sent on the same connection.

2.3.3 HTTP/3 Connection

The specification *Using Datagrams with HTTP* introduces the *Capsule Protocol*. This protocol is a sequence of type-length-value tuples that new HTTP Upgrade Tokens can choose to use. It provides endpoints with reliable end-to-end delivery of request-related information over HTTP request streams. [13]

In order to exchange IP addresses and routes, information needs to be exchanged via a bidirectional stream control. For this purpose the crate *H3* [1] is used which provides an HTTP/3 implementation. A request is sent by the client via this control stream. The server checks this request and if the validation is successful, the server sends a responds code which identifies the success. Once the request is successful, the Capsule Control protocol is used with the packet types address assignment, address request and route indication. This exchanges all the necessary information about the stream. This information is used with the help of the Netlink socket to configure the TUN device.

The crate *H3* must be extended to support the Capsule Control protocol for this application. In addition, the QUIC datagram extension must be implemented in the crate.

2.3.4 Asynchronous Runtime Environment

Tokio is an asynchronous runtime environment for the Rust programming language. It was explicitly designed for writing network applications and it is used for running asynchronous applications. Asynchronous programming is a concurrent programming model. It allows the execution of a large number of concurrent tasks on a small number of operating system threads. [8]

The application mainly consists of reading, writing, encrypting, receiving and sending IP packets. These are Input/Output (I/O) intensive tasks that block the thread. The use of the asynchronous runtime environment shifts the numerous blocking tasks into a relatively few operating system threads.

2.4 Experiment Environment

This chapter describes the setup of the experiment environment that was configured to investigate the properties and network performance of the QUIC tunnel application. The tools used to generate and record traffic are described and obtain the network statistics. Finally, a baseline measurement is given to determine the maximum of the network.

2.4.1 Design of Experiment Environment

The experiment setup contains three containers (client, emulator, server), which are connected by two networks (left, right). The Docker Compose Configuration is included in the Appendix A.1. The network traffic of the two networks is routed through the emulator container. The client and server are connected via the emulator. This one-to-one design allows as much control over the testbed network as possible without external unwanted influences.

In the server container, the QUIC tunnel application runs in server mode and waits for a QUIC initial packet. The client container starts the QUIC tunnel application in client mode together with the IP:Port of the server and tries to establish a QUIC connection to the server. The emulator container is used for routing and to emulate different network conditions by using the network tools *tc* and *netem*. The network tools provide certain Quality of Service (QoS) characteristics: bandwidth limitation, delay, bit errors and

packet losses. The tools are provided by the traffic control infrastructure of the Linux kernel. Additionally the network traffic is recorded by *tcpdump* in the emulator container. Furthermore, the traffic in the client and server between origin and the TUN devices is recorded. The Listing 2.14 shows an image of the experiment setup.

Listing 2.14: Experiment setup with three containers

```
+--------+ IP A        IP B +--------+ IP C        IP D +--------+
|        |------------------|        |------------------|        |
| Client | IP Subnet Left   |Emulator| IP Subnet Right  | Server |
|        |------------------|        |------------------|        |
+--------+        <->       +--------+        <->       +--------+
```

The measurement tool *iPerf3* is used to generate TCP traffic. In the server container *iPerf3* is started in server mode listing on the TUN link. In the client container the tool is started together with the IP of the server TUN link. The traffic is transmitted through the QUIC tunnel. The tool measures the maximum achievable bandwidth, packet loss, delay and jitter.

2.4.2 Parameters, Traffic and Monitoring

The network emulation tool *iPerf3* allows to measure the parameters as already mentioned such as latency, jitter and throughput in a network. The tool generates unidirectional or bidirectional UDP and TCP data streams between the two ends. This tool is used to investigate the behaviour of TCP data streams within the QUIC tunnel.

The default setting is 10 s per run for *iPerf3*, which is configurable. The time for measurement with exceptions is set to 180 s. The Linux kernel was extended by further congestion control modules so that *iPerf3* can be extended by the algorithms from the chapter 2.1.1.

The congestion control algorithms are BIC, CUBIC, Vegas, BBR, New Reno and Westwood as mentioned in chapter 2.1.1. CUBIC and BBR were selected because they are currently the most widely used. New Reno was included because it is an older algorithm that was widely used. The algorithm has found applications especially in older systems. Within this work also the transferability of older systems in the future is investigated. Thus, New Reno is also applied. BIC is the predecessor of CUBIC, Vegas works similarly

to BBR and Westwood has similar characteristics to New Reno. Therefore, the three were contrasted as a direct comparison.

The open source packet Wireshark, which is responsible for trace capturing and analysis, is highly used during this work. It is allowed to capture the data in real time from a network interface or read it from a Packet Capture (PCAP) file. Wireshark contains dissectors with which packets can be decoded. This allows fields of selected packets to be analysed. Furthermore, it is possible to use Wireshark packet trace analysis to examine the behaviour of the used protocols.

A characteristic of QUIC is that the communication is encrypted. This accounts also when using TCP/TLS. Wireshark allows to decrypt the packet together with a file containing the used TLS secrets. The encrypted payload is then displayed as on the wire and additionally the decrypted payload.

The RFCs described in the chapter 2.2 are not yet supported by Wireshark, so the decrypted payloads of the QUIC packets are not decoded. Therefore Wireshark has also been equipped with dissectors for this Protocols. The dissectors of these protocols will be taken up again in the chapter future works.

2.4.3 Network Monitoring Commands

The network monitoring command *ss* is used in the client as well as in the server to read out flow control and congestion control values from the TCP sockets. It is a command to dump socket statistics. In this paper, the cwnd, RTT and ssthresh are of special interest. The Listing 2.15 contains a line of output from *ss*. This shows a TCP connection with the congestion control algorithm Reno from the client to the server together with the mentioned values.

Listing 2.15: *ss* output of TCP flow and congestion control values

```
root@client:/# ss --no-header -ein 10.8.0.1
tcp ESTAB 0      298680 10.8.0.3:41766 10.8.0.1:5201
timer:(on,049ms,0) ino:91548 sk:100b cgroup:/ <->
ts sack reno wscale:7,7 rto:231 rtt:30.234/2.161 mss:1048
pmtu:1162 rcvmss:536 advmss:1110 cwnd:72 bytes_sent:140469
bytes_acked:65014 segs_out:137 segs_in:56 data_segs_out:135
send 19965866bps lastsnd:14 lastrcv:96 lastack:14
pacing_rate 39930576bps delivery_rate 9226800bps delivered:64
busy:73ms unacked:72 rcv_space:11100 rcv_ssthresh:64426
notsent:223224 minrtt:4.499
```

The command does not offer the possibility to give a new output when the value changes. Therefore, a shell script A.2 was used in the experiment, which periodically polls the current values with *ss*.

The network monitoring command *netstat* is the predecessor of *ss* and also allows to display network connections, routing tables, masquerade connections, and multicast memberships. It also enables to show interface statistics as well as buffer errors. The Listing 2.16 shows such an output. In this paper it was used to debug unwanted retransmissions.

Listing 2.16: *netstat* output of interface statistics

```
root@server:/# netstat -s
Ip:
    Forwarding: 1
    1098281 total packets received
    0 forwarded
    0 incoming packets discarded
    1098281 incoming packets delivered
    510998 requests sent out
Tcp:
    0 active connection openings
    2 passive connection openings
    0 failed connection attempts
    0 connection resets received
    0 connections established
    514230 segments received
    255502 segments sent out
    0 segments retransmitted
    0 bad segments received
    110 resets sent
Udp:
    582619 packets received
    0 packets to unknown port received
    1432 packet receive errors
    255496 packets sent
    1432 receive buffer errors
    0 send buffer errors
TcpExt:
    504588 packet headers predicted
    4 acknowledgments not containing data payload received
    7 predicted acknowledgments
    1 connections reset due to unexpected data
    TCPRcvCoalesce: 2
    TCPOFOQueue: 7725
    TCPOrigDataSent: 10
    TCPDelivered: 9
    TCPAckCompressed: 6748
```

2.4.4 Baseline

In this chapter, the maximum achievable capacity of the experiment setup for each congestion control algorithm is determined. This serves to identify the limits of the experiment setup and to exclude performance losses caused by the arrangement. For each congestion control algorithm in chapter 2.1.1, *iPerf3* is used to exchange data between client and server for 180 s. The experiment setup is not artificially limited by *tc-netem*. For each measurement, the data from *iPerf3* and the information from the TCP socket are captured by *ss*. The Figure 2.1 contains the data of the congestion control algorithms of the baseline measurement.

The Figure 2.1a and Figure 2.1b contain the data for the cwnd and the RTT. It is noticeable that both algorithms jump to a fixed value after 50 s seconds and hold it. There are no signs of congestion in the system. This is also confirmed by the output of *iPerf3*, which shows that there is no retransmission. BIC transmits at a bitrate of $6.87\,\mathrm{Gbit\,s^{-1}}$ and transfers 144 GB during the measurement. The bitrate of CUBIC is $6.77\,\mathrm{Gbit\,s^{-1}}$ and the total amount of 142 GB. Although BIC has a cwnd twice as large, this has no advantage in the total amount and bitrate. The system does not show any congestion symptoms such as packet loss. It can be assumed that either the transmitter limits the rate by a lower receiver window or the transmitter is not able to increase the bitrate.

The measurement of the Vegas congestion control algorithm can be inspected in the Figure 2.1c. In this Figure the y-axis is scaled by 10^6. The cwnd increases linearly without any change on RTT. Especially in the case of such high cwnd it can be assumed that a smaller value like the receiver window determines the bitrate. The evaluation shows that there was no retransmission and the bitrate is $6.99\,\mathrm{Gbit\,s^{-1}}$ and the total amount is 146 GB.

Measurements with BBR is graphically represented in the Figure 2.1d. Compared to the other Figures, the RTT has regular, strongly deviating peaks. This corresponds to the method used by the algorithm to determine the cwnd by examining the network with peaks in irregular intervals. Once again, no losses due to retransmission were measured. The bitrate is $6.06\,\mathrm{Gbit\,s^{-1}}$ and a total of 127 GB was transmitted.

The Figure 2.1e shows the typical spikes of Reno. It is notable that the RTT rises in the immediate proximity of the peaks of the spikes. The y-axis of the RTT is multiplied by

10^2, so Reno has the most extreme RTT in this experiment. Reno has a bitrate of only $2.73\,\mathrm{Gbit\,s^{-1}}$ and transmits 57.20 GB.

Westwood in the Figure 2.1f is very similar to the BIC measurement. Incremental increase, high constant cwnd value, the bitrate is $6.74\,\mathrm{Gbit\,s^{-1}}$ and the total amount of transfer is 141 GB.

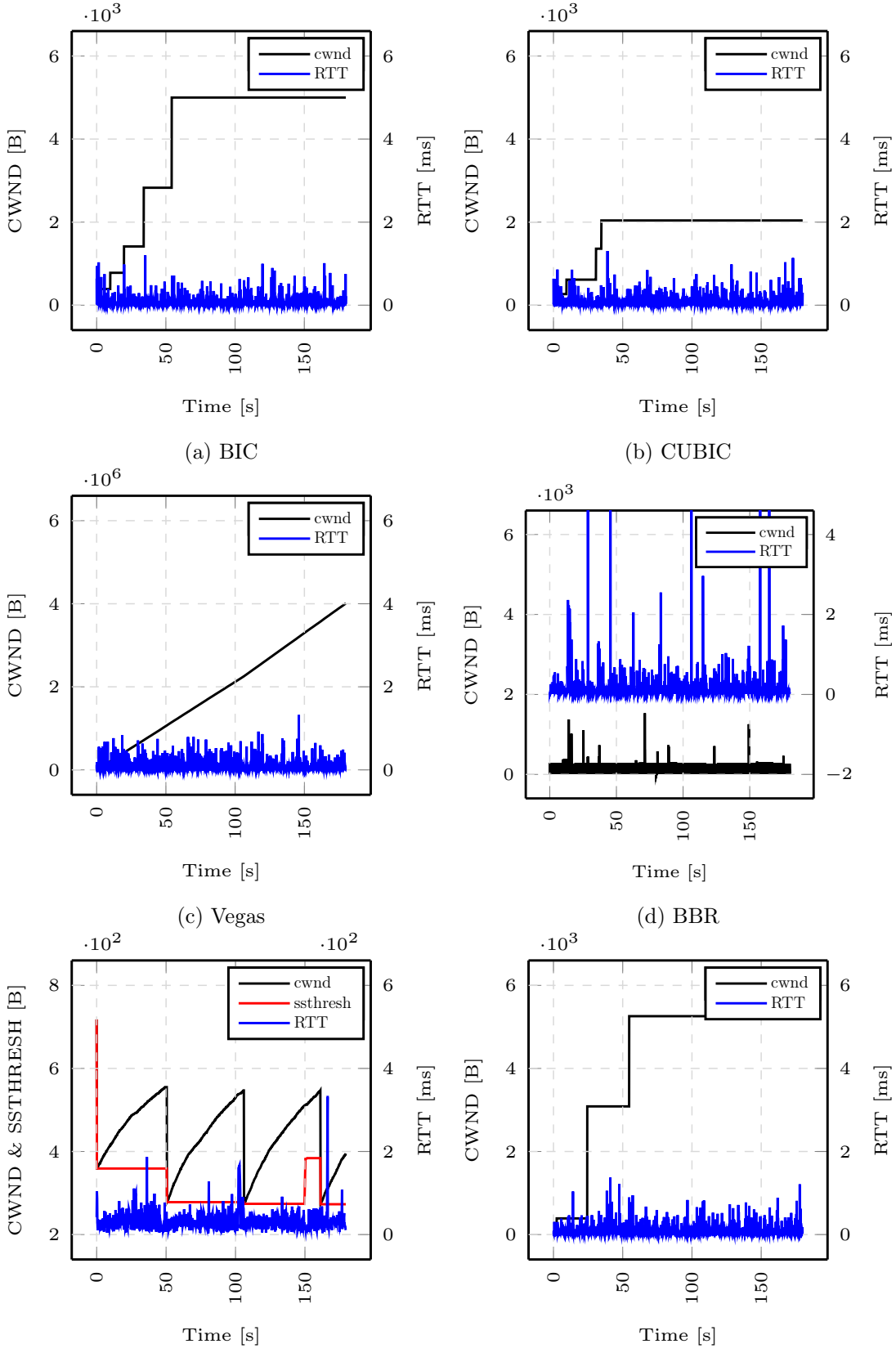


Figure 2.1: Baseline measurement of experiment environment.

2.5 Analysis of Results

2.5.1 UDP Receive Buffer Size

For the investigation, a TCP connection was established with *iPerf3* using the congestion control algorithm CUBIC between client and server via the QUIC connection and random bytes were transmitted from the client to the server for 60 s. A total of 595 MB were transferred at a bitrate of 83.10 Mbit s^{-1}. These results are from the output of *iPerf3* shown in the Listing 2.17. Noticeable in this measurement are the 579 retransmitted TCP packets and indicates the number of TCP packets that had to be retransmitted. But the experimental setup was configured without artificial losses from the network tools *tc* and *netem*. The Figures 2.2 shows the values of the cwnd and the ssthresh.

Listing 2.17: An *iPerf3* measurement shows an increased number of retransmissions

```
root@client:/# iperf3 -c 10.8.0.1 -t 60 -C cubic
[ ID] Interval           Transfer     Bitrate         Retr
[  5] 0.00-60.00  sec  595 MBytes  83.1 Mbits/sec  579  sender
[  5] 0.00-60.02  sec  593 MBytes  82.8 Mbits/sec       receiver
```

The Figure 2.2 shows the TCP standard startup phase at the beginning. The first losses reduce the cwnd and the window starts to increase again to the maximum value. In the long term, there are so many losses that the cwnd falls to a value below 200 B. Except for the first 20 s, the concave portion of the cubic function is not noticeable.

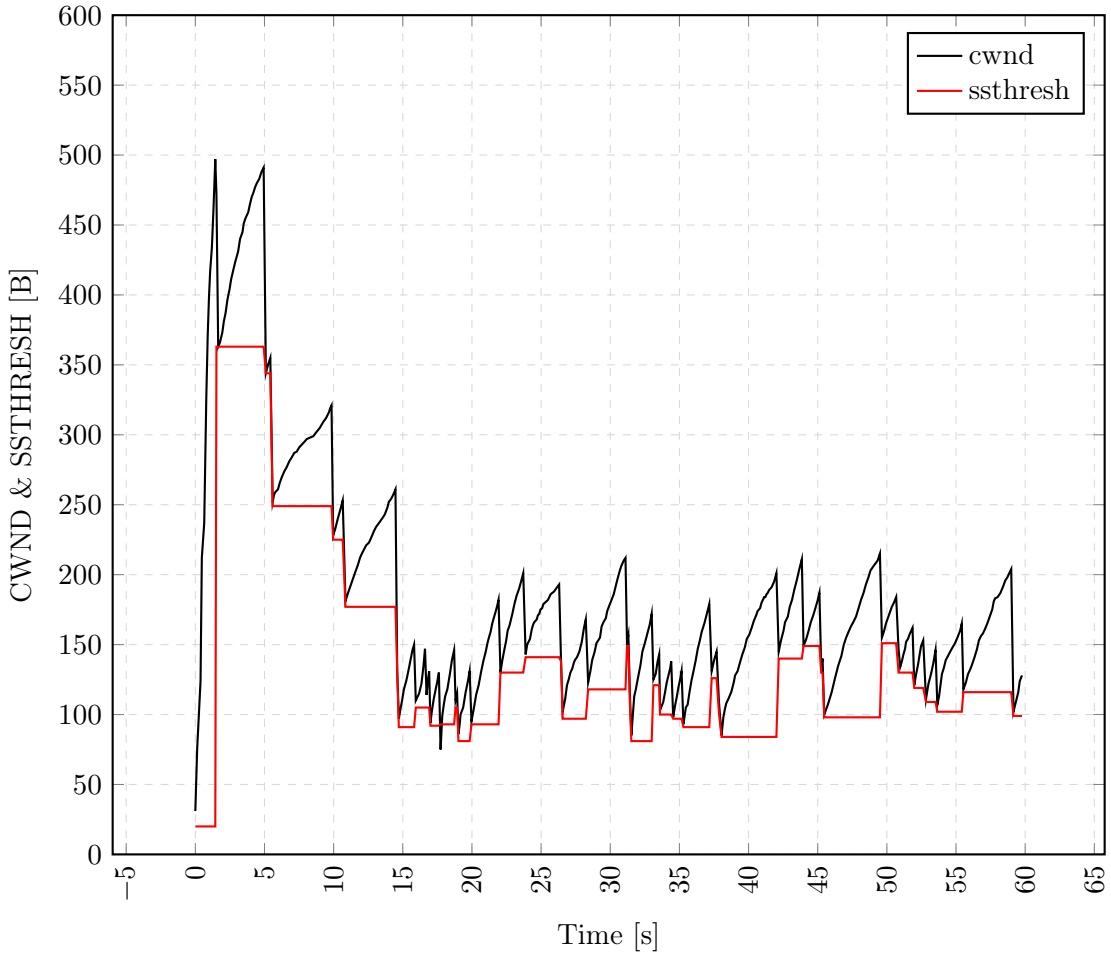


Figure 2.2: Measurement with CUBIC, the data show that the network loses packets

The *netstat* tool is used to examine the links in all three containers. The Listing 2.18 shows the statistics of the servers. Here it can be seen that the receive buffer for UDP packets shows errors. This buffer filled up during the measurement and did not forward the UDP to the QUIC application.

Listing 2.18: The network information shows that the receive buffer drops UDP packets

```
root@server:/# netstat -s
Ip:
    Forwarding: 1
    1292263 total packets received
    0 forwarded
    0 incoming packets discarded
    1292263 incoming packets delivered
    599983 requests sent out
Udp:
    688538 packets received
    0 packets to unknown port received
    579 packet receive errors
    298735 packets sent
    579 receive buffer errors
    0 send buffer errors
```

The Listing 2.19 shows a section of the necessary terminal commands to read the buffer size and replace it with a larger value. Since the config file is not part of the namespace it must be changed in the host system. The change of this value was not straight-forward since the host system is a virtual machine within the Docker Desktop application, which is not intended to be modified. The Listing 2.19 also describes the command used to break out of the container to change the value in the host system.

Listing 2.19: Container breakout to increase UDP receive buffer size

```
$ docker run -it --rm --privileged --pid=host alpine:edge \
nsenter -t 1 -m -u -n -i sh
# sysctl net.core.rmem_max
net.core.rmem_max = 212992
# sysctl net.core.rmem_default
net.core.rmem_default = 212992
# sysctl -w net.core.rmem_max=26214400
net.core.rmem_max = 26214400
# sysctl -w net.core.rmem_default=26214400
net.core.rmem_default = 26214400
```

The measurement in the Listing 2.20, shows that the number of retransmissions has reduced. The Figures 2.3 show that the congestion control algorithm CUBIC has to react to fewer losses and the cubic function is clearly visible.

Listing 2.20: A measurement with the extended buffer shows reduced retransmissions

```
root@client:/# iperf3 -c 10.8.0.1 -t 60 -C cubic

[ ID] Interval           Transfer     Bitrate         Retr
[  5] 0.00-60.00  sec  597 MBytes  83.4 Mbits/sec   50   sender
[  5] 0.00-60.03  sec  596 MBytes  83.2 Mbits/sec        receiver
```

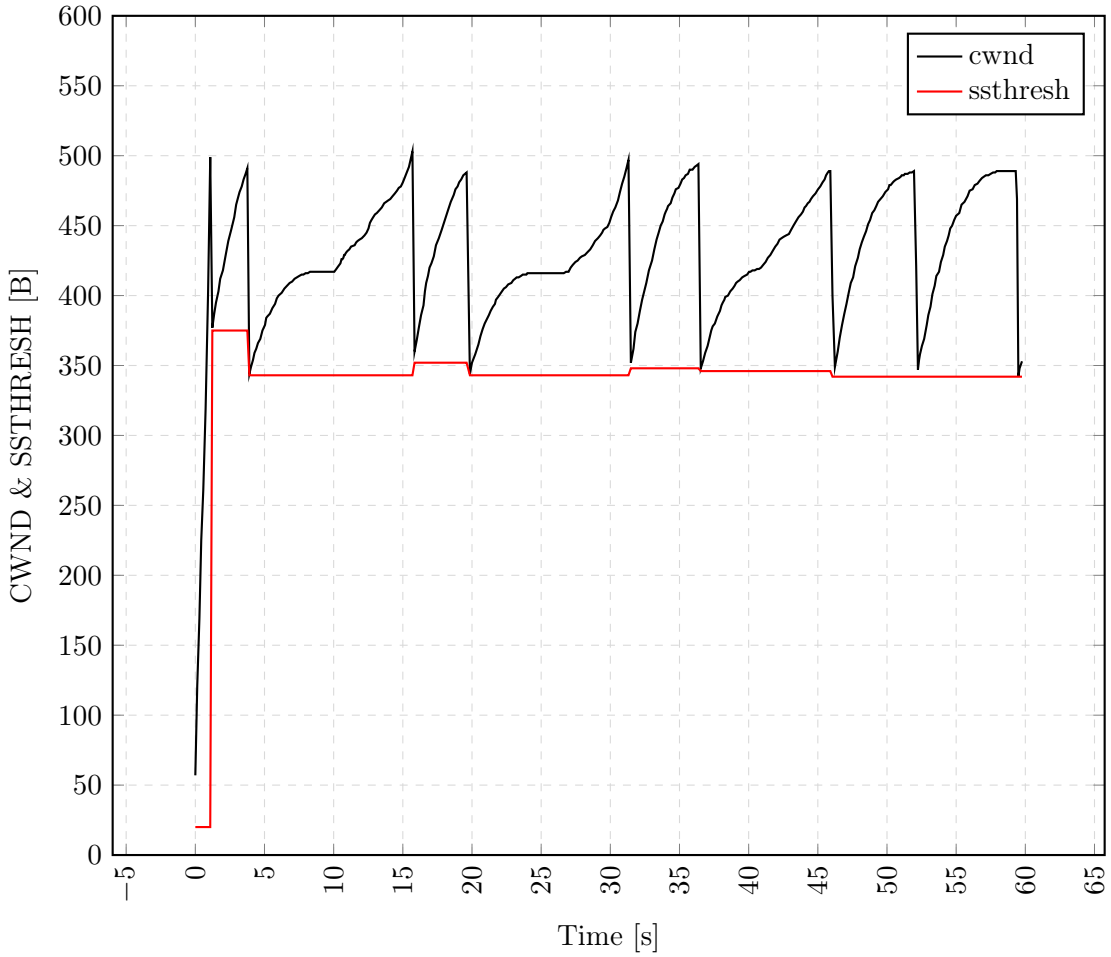


Figure 2.3: The extension of the UDP buffer shows reduced congestion

The Figure 2.4 shows the RTT of a measurement with and without the buffer extension. It can be seen that the RTT increases with an increased buffer. Due to that, the buffer is able to store more packets in the FIFO queue before they are processed and acknowledged. Before the increase, these packets were lost and sent at a later time.

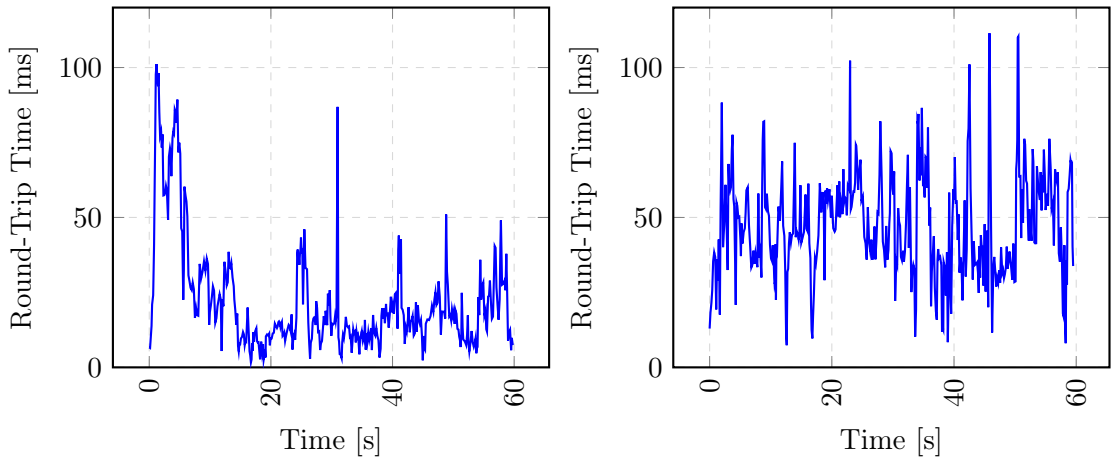


Figure 2.4: Increment of the UDP buffer shows an increased RTT

Part of this investigation is to assess whether the extended buffer leads to an improvement in the transmission of data via the QUIC connection. To measure this, data was transferred for at least ten times with *iPerf3* for 60 seconds via a TCP connection from the client to the server. The congestion control algorithms CUBIC and Reno were used. The buffer was initialised with the default values of the Linux kernel and increased by the value from Listing 2.19 in the second measurement. Figure 2.5 shows the results of this measurement. The left side shows that with both Reno and CUBIC the bitrate increased by about 4-6%. The retransmissions are reduced with both congestion control algorithms.

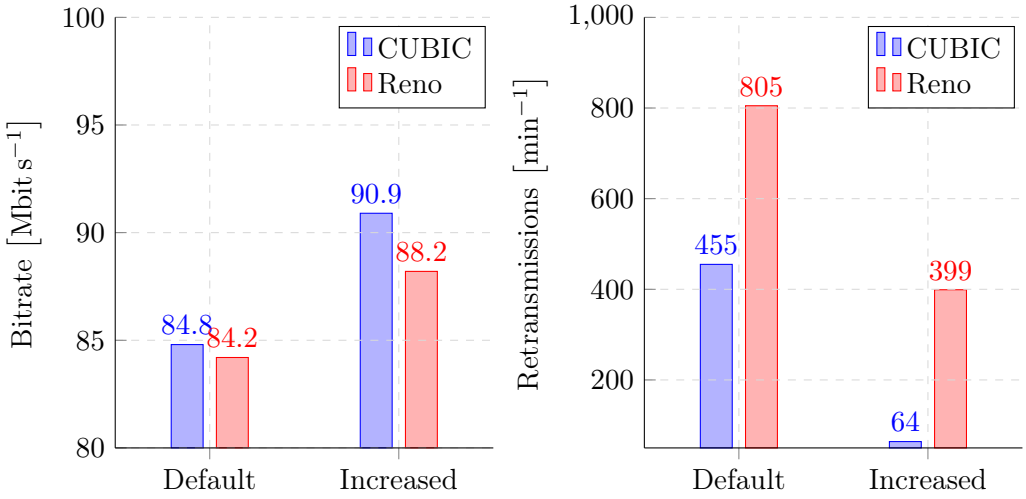


Figure 2.5: Expansion of the buffer reduces retransmission and increases the bitrate

After the above described issue was identified, an internet search revealed that the developers of the QUIC reference implementation also found this problem.

> "Experiments have shown that QUIC transfers on high-bandwidth connections can be limited by the size of the UDP receive buffer. This buffer holds packets that have been received by the kernel, but not yet read by the application[...]. Once this buffer fills up, the kernel will drop any new incoming packet." [14]

2.5.2 TUN Transmit Queue Length

The measurements 2.5 show retransmission even after extending the UDP receive buffer. In the Listing 2.21 a measurement for 20 s is shown. It results in 198 retransmission and as before, losses were not artificially added.

To find out why this retransmission exists, the packets are recorded during a measurement within the server and client container with *tcpdump* at the *TUN* interface as well as at the *eth0* interface.

Listing 2.21: iPerf3 still reports retransmissions

```
root@client:/# iperf3 -c 10.8.0.1 -t 20 -C cubic
Connecting to host 10.8.0.1, port 5201
local 10.8.0.2 port 50418 connected to 10.8.0.1 port 5201
Interval          Transfer    Bitrate         Retr Cwnd
0.00-1.00   sec  12.6 MBytes 106 Mbits/sec  0    563 KBytes
1.00-2.00   sec  7.37 MBytes 61.8 Mbits/sec 123  490 KBytes
2.00-3.00   sec  6.56 MBytes 55.0 Mbits/sec 2    374 KBytes
...
15.00-16.00 sec  8.55 MBytes 71.7 Mbits/sec 56   443 KBytes
16.00-17.00 sec  12.1 MBytes 101 Mbits/sec  0    493 KBytes
17.00-18.00 sec  8.48 MBytes 71.1 Mbits/sec 17   370 KBytes
...
- - - - - - - - - - - - - - - - - - - - - - - - -
Interval          Transfer   Bitrate         Retr
0.00-20.00  sec  186 MBytes 78.0 Mbits/sec  198
```

The TCP congestion control algorithm is required to respond with a DupACK when a gap exists between the received packets. DupACKis the name for an ACK that contains a sequence number that has already been confirmed. This DupACK is intended to trigger the fast retransmission part of the congestion control algorithm.

These DupACKs can be found in the records. The TCP data packets with these sequence numbers are also present, but only in the client network records and not in the server records. The decrypted QUIC traffic shows that the TCP packets are not in the datagram frames of the QUIC packet. This indicates that the TCP packet between the TUN device and the first Ethernet interface has been lost.

After examining the logs of the application, it appears that the reader of the TUN device was never able to read this packet and therefore could not process it further. This leads to the conclusion that either the TUN device has lost the packet, or there might have been problems with the reading by the TUN reader.

The Linux *ifconfig* command in the Listing 2.22 shows in the line of the transmitted TX 198 packets were dropped, which is the same number from the measurement 2.21. In addition, the output shows that the *txqueuelen* can contain 500 packets.

Listing 2.22: TUN Configuration shows that the size of the transmit queue is 500

```
root@client:/# ifconfig quic-vpn
flags=4305<UP,POINTOPOINT,RUNNING,NOARP,MULTICAST>  mtu 1162
inet 10.8.0.2  netmask 255.255.255.0  destination 10.8.0.2
unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
txqueuelen 500  (UNSPEC)
RX packets 92197  bytes 4808348 (4.5 MiB)
RX errors 0  dropped 0  overruns 0  frame 0
TX packets 184416  bytes 202807184 (193.4 MiB)
TX errors 0  dropped 198 overruns 0  carrier 0  collisions 0
```

With the same Linux command, this queue can be extended, in this case from 500 to 1000 packets. The measurements for this change are shown in Figure 2.6. The data shows a reduction in the bitrate and the number of retransmissions has increased. The TX queue itself shows no drops. The Wireshark output confirm again that TCP packets are being lost between the TUN interface and the Ethernet interface. The application logs in Listing 2.23 show the reason for the packet loss. The send buffer of the QUIC implementation fills up faster than the content can be sent.

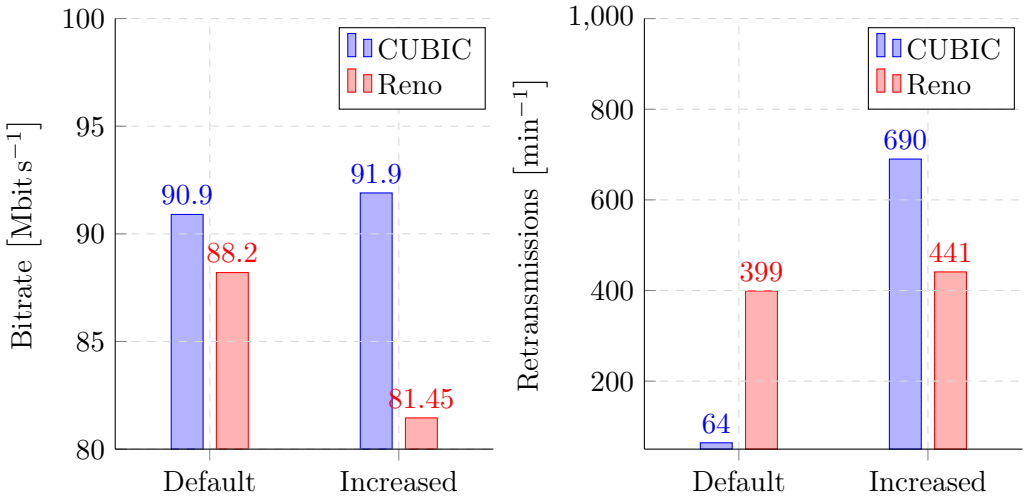


Figure 2.6: Expansion of the TUN queue has a negative effect

Listing 2.23: Logs warns that datagrams are dropping

```
WARN quinn_proto: dropping outgoing datagram len=1100
WARN quinn_proto: dropping outgoing datagram len=1100
WARN quinn_proto: dropping outgoing datagram len=1100
```

The datagram send buffer is increased to a ratio of 1.2 and the measurement is performed again. The Figure 2.7 shows the data of this measurement, it can be seen that the bitrate has increased again and there is no retransmission. In Figure 2.8 shows a measurement of the RTT of a TCP stream with the congestion control algorithm CUBIC. Within this measurement, both the TX queue and the datagram send buffer of the QUIC application are increased. The measurement clearly shows high latency and jitter, the increase of the values leads to a Bufferbloat.

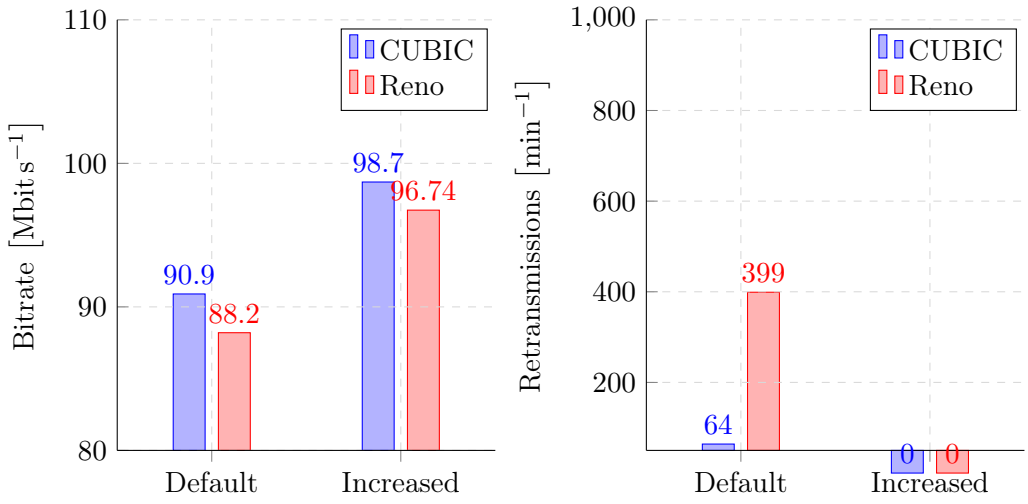


Figure 2.7: Expansion of the send buffers reduces retransmission and increases the bitrate

The expansion of the queue and the datagram send buffer does not lead to the intended success of improving the speed of the tunneling application, but rather to an increase in latency and jitter. The packets are in the queue longer and the loss-dependent congestion control algorithms are not informed that they are sending too fast.

Figure 2.8 shows a measurement with the CUBIC congestion control algorithm and all three buffers have been increased. Comparing this Figure with the Figure 2.4 from the last experiments, it is clear that the RTT has doubled and jitter is developing.

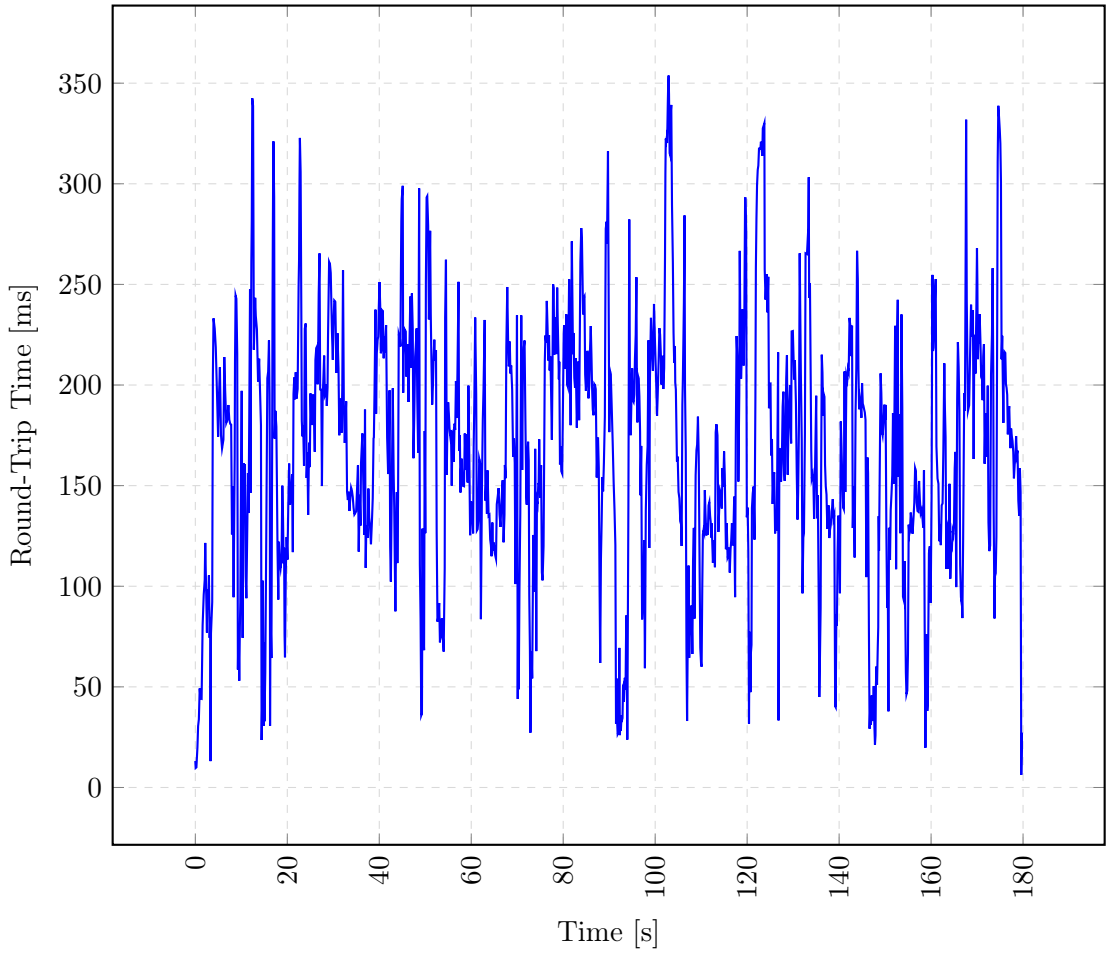
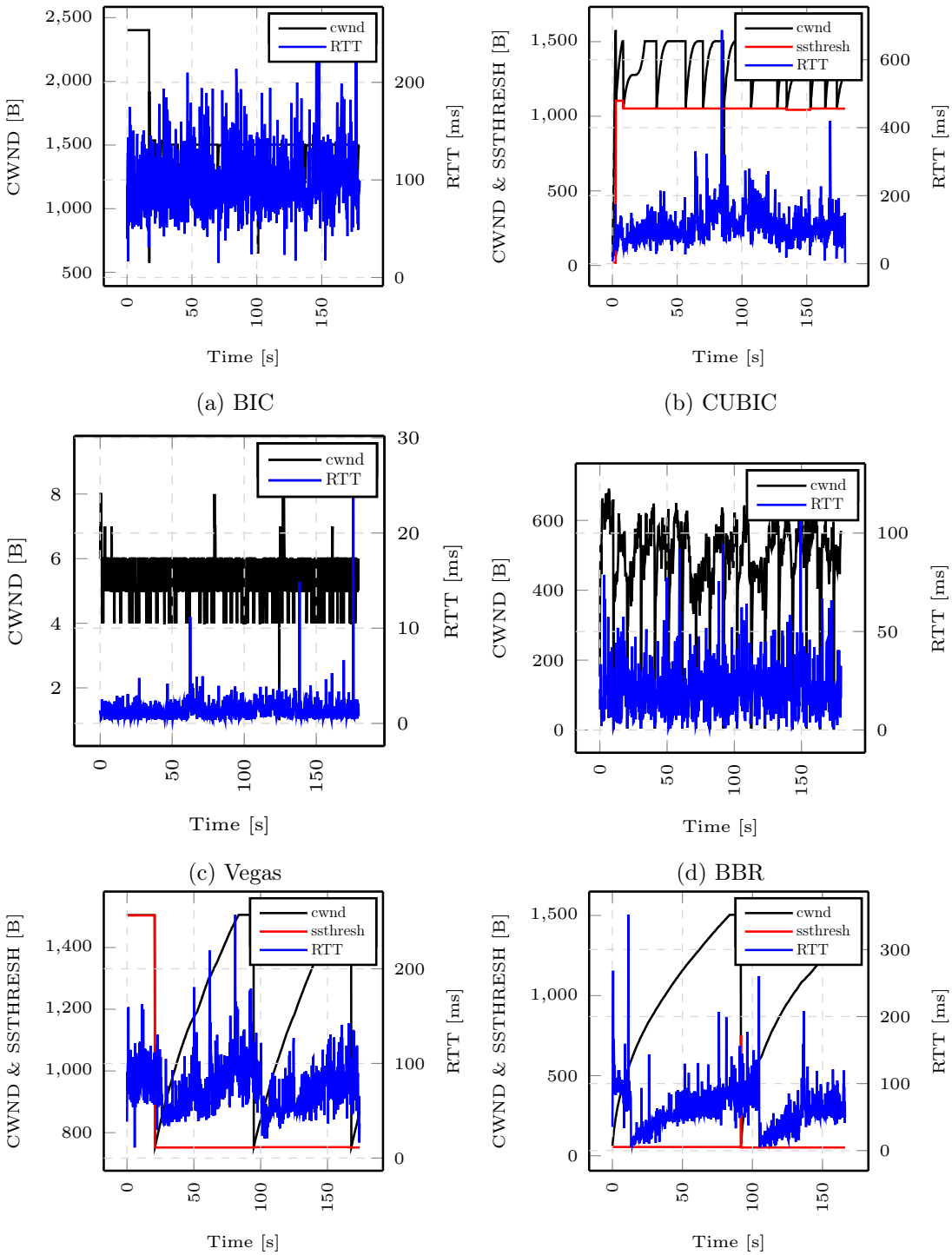


Figure 2.8: Expansion of the buffers lead to longer RTT

This excessive buffering of packets leads to an increase in latency and jitter, which is called bufferbloat. The Figure 2.9 impressively shows how the various congestion control algorithms behave when the network caches an excessive number of packets. The algorithms receive a congestion signal less often and assume for longer that the network can handle more packets. Figure 2.9e shows the Reno corrected only twice within three minutes and has an extended RTT time. The same is true for Westwood. BIC does not correct at all. Only Vegas and BBR, which include the RTTs in the feedback loop, are able to keep the RTT low.

The understanding that the three buffers exist and that emptying them too late will result in retransmission was achieved. Expanding the buffers is not necessarily a complete solution.



(a) BIC

(b) CUBIC

(c) Vegas

(d) BBR

(e) Reno

(f) Westwood

Figure 2.9: Measurement with maximum buffer size

2.5.3 Impact of Buffer Expansions on the Experiment

Three different queues/buffers were examined within the experiment. To determine their impact, this section uses a 2^k full factorial experimental design with the different congestion control algorithms in the experimental system. The intention is to determine which combinations have an effect on the RTT and bitrate. The following inputs are used:

- Congestion control: Vegas / BBR / BIC / CUBIC / Westwood / Reno
- Transmit queue: 500 / 1000
- Datagram send buffer [B]: 1048576 / 2097152
- Receive buffer [B]: 212992 / 26214400

The experiment is set up with each combination. For 180 s, a TCP stream from the client to the server is established and analysed. The Tables 2.1, 2.2 and 2.3 contains all combinations as well as the results of the experiments.

Table 2.1: Full factorial experiment for BIC and CUBIC with different buffers sizes

Congestion	Transmit Queue	Datagram Send	UDP Receive	RTT ms	Bitrate Mbit s^{-1}
BIC	-1	-1	-1	18.19	105
	-1	-1	1	28.45	99.9
	-1	1	-1	35.26	59.5
	-1	1	1	55.36	53.7
	1	-1	-1	19.59	110
	1	-1	1	62.89	109
	1	1	-1	28.41	71.6
	1	1	1	83.22	129
CUBIC	-1	-1	-1	10.16	101
	-1	-1	1	24.41	110
	-1	1	-1	17.82	55.2
	-1	1	1	35.18	79.7
	1	-1	-1	9.37	102
	1	-1	1	73.89	90.8
	1	1	-1	18.47	55.7
	1	1	1	72.03	136

The Table 2.1 shows all possible combinations of input parameters. Clearly, almost all increases in the size of the queues and buffers will result in a longer RTT. Increasing the datagram send buffer has the effect of significantly increase RTT. One assumption for this behaviour is that the implementation only empties the buffer when a timer has expired or the buffer is full and starts dropping packets. It therefore results that a larger buffer is emptying itself less often. This needs to be checked. If only the transmit queue of the TUN device is increased, this has no negative effect on RTT and bitrate. As observed in the previous chapter, increasing the size of all buffers/queues will increase the bitrate the most. The disadvantage is that the RTT increases significantly. This examination of the two congestion algorithms shows that an increase in buffers/queues does not necessarily lead to an improvement.

Table 2.2: Full factorial experiment for Vegas and BBR with different buffers sizes

Congestion	Transmit Queue	Datagram Send	UDP Receive	RTT ms	Bitrate Mbit s^{-1}
Vegas	-1	-1	-1	1.25	60.9
	-1	-1	1	1.08	64.7
	-1	1	-1	2.19	28
	-1	1	1	0.95	76.1
	1	-1	-1	1.31	60.9
	1	-1	1	1.7	50.1
	1	1	-1	1.8	36.4
	1	1	1	0.91	73.5
BBR	-1	-1	-1	19.1	77.7
	-1	-1	1	18.2	98.5
	-1	1	-1	24.51	47.6
	-1	1	1	15.93	117
	1	-1	-1	15.78	88.9
	1	-1	1	25.56	76.1
	1	1	-1	23.73	54.9
	1	1	1	15.50	118

The Table 2.2 shows the full factorial experiment with the Vegas and BBR algorithms. As already described in the chapter 2.1.1, these two congestion control algorithms change the cwnd based on the RTT time. In this way they try to keep the RTT as low as possible.

This can also be observed in the measurements. In this series of experiments it can be observed that an increase of the datagram send buffer has a negative effect, as already observed in 2.1. In the case where all buffers and queues are increased, both algorithms are able to maintain a minimum RTT as well as increase the bitrate. They may find the sweet spot that buffers are felt to such an extent that they are immediately flushed again in the next step without the packets having to stay there for a long time. By increasing the size of the buffers, additional packets are not lost, which can also be processed in the next step. While CUBIC and BIC are not able to recognise a bufferbloat, Vegas and BBR are.

Table 2.3: Full factorial experiment for Reno and Westwood with different buffers sizes

Congestion	Transmit Queue	Datagram Send	UDP Receive	RTT ms	Bitrate Mbit s^{-1}
Reno	-1	-1	-1	9.95	102
	-1	-1	1	23.77	102
	-1	1	-1	18.12	52.2
	-1	1	1	18.79	134
	1	-1	-1	9.45	97.9
	1	-1	1	63.01	91.4
	1	1	-1	18.83	50.6
	1	1	1	60.58	135
Westwood	-1	-1	-1	8.5	103
	-1	-1	1	30.42	71.1
	-1	1	-1	14.31	64.5
	-1	1	1	16.73	133
	1	-1	-1	8.9	100
	1	-1	1	63.09	75.4
	1	1	-1	17.59	47.4
	1	1	1	52.88	139

The Table 2.3 shows the results for Reno and Westwood. In this series of experiments, it is noticeable that when only the datagram send buffer is increased, the bitrate is reduced and the RTT increases, but as soon as the UDP receive buffer is also increased, the bitrate increases above the standard value. This effect can also be observed with BBR. Unlike the other algorithms, increasing the transmit queue has no particular effect.

The reason for the deviation can be assumed to be due to the special nature of the algorithms. These cause an overload of the network by a linear growth of the cwnd and then overcorrect the cwnd. This leads to a sharp increase in RTT at the peak, which is followed by a sharp reduction in transmission and the emptying of the larger buffers. This leads to an increased average bitrate.

2.5.4 Fairness

As in subchapter 2.1.2 defined, TCP friendliness plays a crucial role when when monitoring multiple simultaneous streams within the QUIC tunnel. The latter section observes the handling of different connections within the QUIC tunnel given with limited bandwidth. The main focus lies on the distribution and the assessment of whether this is done fairly.

At first, two TCP connection with CUBIC congestion controls are compared and analysed. After that gained knowledge a more sophisticated comparison is done: one stream agains two with different congestion controls. These results are analysed in Table 2.4, 2.5 and 2.6.

The first measurement has two TCP connections established simultaneously from the client to the server, transmitting data for 60 s. Both connections use the CUBIC congestion control algorithm. The sent bytes of both connections are recorded and the established measurement sequence for this approach was carried out several times. The Figure 2.10 shows one of these measurements. The analysis shows that the two streams share the connection equally. It can be observed that both streams react in a similar manner to events in the connection. It can be concluded that both connections are capable of sending data.

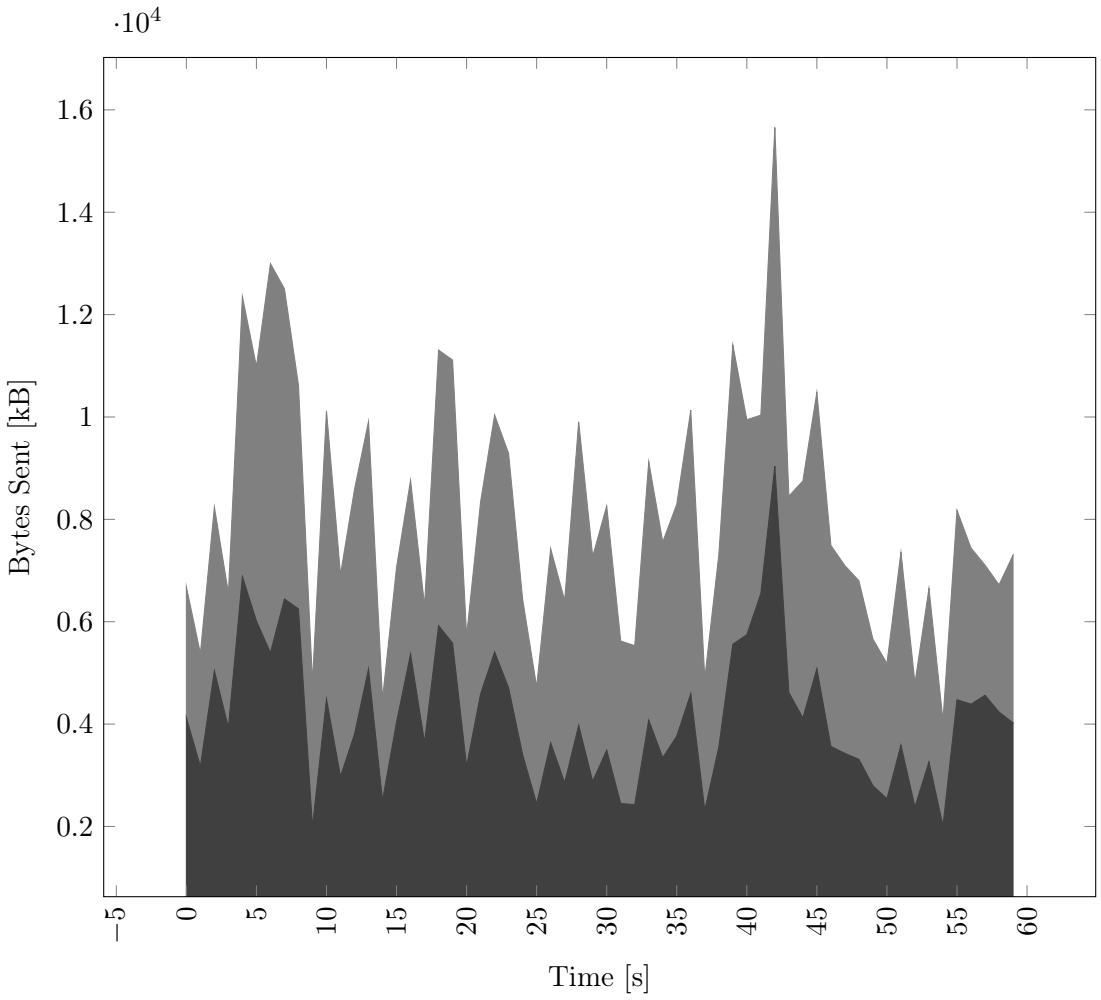


Figure 2.10: Measurements of sent bytes from two concurrent TCP connections

As measured before, the congestion window size and the slow start threshold of both TCP connections are recorded and shown in the Figure 2.11. The start phase of both connections can be identified through the represented peaks at the beginning. It can be evaluated that the first connection (left upper subfigure) represents a more severe starting phase than measured in the second connection (left lower subfigure). The reason for this differences is likely due to the fact that *iPerf3* starts the connection sequentially. Therefore, the second connection has less time to increase the congestion window between the start and the first losses packet.

Regardless of the referenced Figure 2.2 it was observed that the window is reduced less and looses less packets by a single connection. Additionally, it can be concluded that the slow start threshold is reduced by half when observing a single connection compared to two streams. The algorithm is not able to find a constant value.

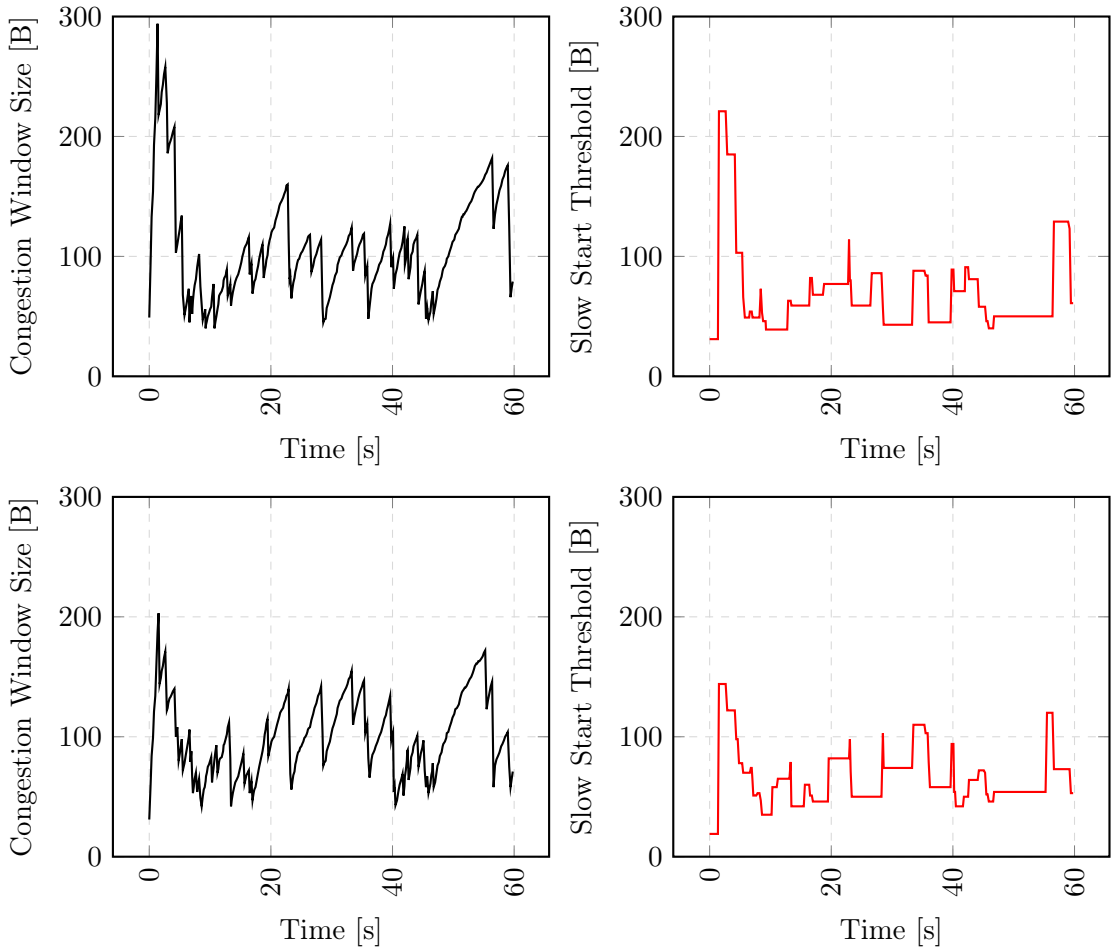


Figure 2.11: Measurements of concurrent TCP connections

For further visualisation the Figure 2.12 represents several connections. Each TCP stream is shown in a different color code. The numbering along the y-axis stands for the amount of indicated TCP streams which are parallel. Generally, it can be determined that within the QUIC tunnel, the TCP connections shares the bandwidth fairly. It can be concluded that the connections represents almost the same bitrate. Especially noticeable is that a single stream is able to get the most out of the connection.

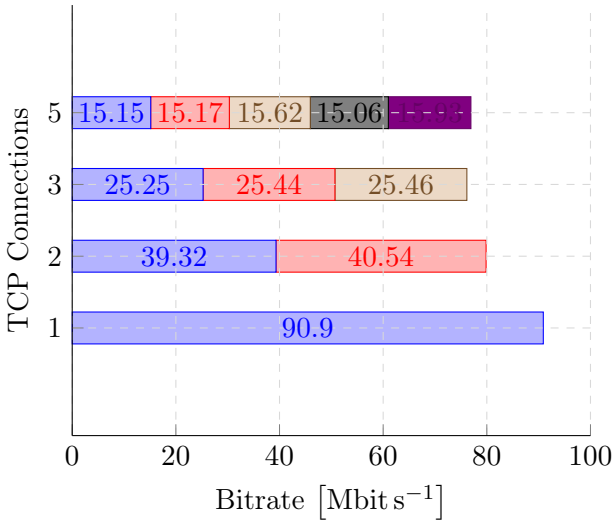


Figure 2.12: Bitrate of concurrent TCP connections in the QUIC tunnel

As mentioned in the beginning of this subchapter, the latter description is based on two different congestion controls. The Table 2.4 shows the results of a measurement series which contains several TCP connections simultaneously in the QUIC tunnels. The connections use the congestion control algorithms BBR and Reno. The left side describes the number of connections used per congestion control algorithm. The RTT values of the individual congestion controls show that all connections results in similar RTT per measurement. This is to be expected, as the connection uses the same path and the tunnel does not prioritise any connection. This does not apply to the bitrate. The first three lines show a BBR connection that has to share the tunnel with 1-3 Reno connections. In case there is a Reno connection, the bandwidth is shared fairly between BBR and Reno. The bitrate of the BBR connection with three Reno connections is then not fairly divided. In this case, the BBR connection is able to secure the double of a Reno connection. BBR connections are more aggressively in securing its own part of the tunnel's bandwidth. The same behaviour cannot be observed when a second BBR connection is added. Then, the measured values are not representing that the BBR connections secure more of the bandwidth. Presumably, the more aggressive behaviour of the BBR connections means that neither connection can secure more of the tunnel's bandwidth. This goes so far that in the measurement with three BBR and one Reno connection, the Reno connection has the largest part of the bitrate. The RTT increase depends on how many Reno connections are available. It has already been observed in

the previous measurements that the Reno algorithm cannot maintain a low RTT. It can be concluded that the bandwidth was not always fairly distributed, but no connection was left out.

Table 2.4: Fairness measurement for BBR and Reno

BBR	Reno	BBR RTT ms	Reno RTT ms	BBR Bitrate Mbit s^{-1}	Reno Bitrate Mbit s^{-1}
1	1	18.29	20.35	49.5	43.2
1	2	24.06	25.55	31.2	24.2
			25.73		23.3
1	3	30.15	31.1	40.3	20.2
			31.04		18.9
			30.72		18.2
2	1	21.56	22.15	27.5	36.2
		21.4		36.9	
2	2	27.63	28.19	13.9	21.6
		27.43	27.78	17.1	22.9
2	3	40.28	40.48	12.5	12.6
		40.62	40.22		12.4
			40.15	15.7	12.4
3	1	26.03	26.81	13.5	29
		26.53		16.4	
		26.32		21.5	
3	2	30.13	30.57	13.6	21.8
		30.32	30.48	11.7	
		30.28		14.1	21.7
3	3	34.85	34.93	13	14.2
		34.75	35.01	11.2	15.3
		34.59	35.3	11.5	13.9

Further, the Table 2.5 shows the measurement series for the congestion control algorithms BBR and CUBIC. Here it was measured that by using a ratio of 1 to 3, the single algorithm is capable of occupying more than twice of the bandwidth. This behaviour for both algorithms could not be observed in earlier experiments with BBR and Reno (see

Table 2.2). It might be that the algorithms are developed in such a way that they are particularly good at sharing the bandwidth among their equals, but less good with other algorithms. The RTT is particularly reduced when there are more BBR connections than CUBIC connections. BBR cannot keep the RTT low if the connections have to be shared with more CUBIC streams. This was also observed with Reno. In conclusion, the bandwidth of the QUIC tunnel is almost fully consumed by BBR and CUBIC.

Table 2.5: Fairness measurement for CUBIC and BBR

CUBIC	BBR	CUBIC RTT ms	BBR RTT ms	CUBIC Bitrate $\mathrm{Mbit\,s^{-1}}$	BBR Bitrate $\mathrm{Mbit\,s^{-1}}$
1	1	20.71	18.57	38.6	48.1
1	2	25.77	25.14	25.7	28.4
			24.95		33
1	3	25.71	24.93	31.9	14.7
			24.84		14.2
			24.88		15.2
2	1	24.05	23.26	25.9	39.8
		23.9		24.4	
2	2	30.77	30.2	19.0	16.3
		30.22	30.44	21.4	24.5
2	3	25.19	25.83	22.7	19.1
		26.1	25.78		13.3
			25.82	22.7	18.4
3	1	31.25	30.65	17.5	35.6
		31.18		17.8	
		31.11		18.3	
3	2	31.92	31.92	16.2	20.2
		31.84	32.1	15.8	
		31.95		16.3	16.1
3	3	37.24	36.59	12.3	12.9
		37.38	36.47	12.2	13.2
		37.2	36.39	12.7	15.2

At last, the congestion control algorithms CUBIC and Reno are compared against each other as shown in the Table 2.6.

Table 2.6: Fairness measurement for CUBIC and Reno

Reno	CUBIC	Reno RTT ms	CUBIC RTT ms	Reno Bitrate Mbit s^{-1}	CUBIC Bitrate Mbit s^{-1}
1	1	11.88	11.87	58.6	59.4
1	2	15.95	15.99	37.7	36.1
			15.79		36
1	3	20.9	20.8	29.4	26.5
			20.79		26.7
			20.82		28.4
2	1	17.81	17.61	35.4	35.2
		17.79		35.2	
2	2	21.38	21.32	27.5	26.8
		21.23	21.39	27.9	26.2
2	3	24.4	24.51	23.8	22.1
		24.32	24.35		23.4
			24.26	23.7	21
3	1	23.53	23.87	26	24.9
		23.45		25.6	
		23.75		27.9	
3	2	24.66	24.74	21.7	20.7
		24.54	24.64	22.4	
		24.64		22.7	20.1
3	3	25.97	26.03	22.1	20.3
		26.05	26.13	21.6	19.8
		26.17	26.15	20.9	18.8

The number of connections used per algorithm is shown on the left side. It is noticeable that the RTTs in all combinations are lower than in the Tables 2.4 and 2.5 which represents the measurements with BBR. Based on the evaluation done before, this was not expected. BBR shows lower RTT in the other experiments. Apparently Reno and CUBIC are better matched than with BBR. A transmission in the QUIC tunnel shows no negative

effect on the connections. The measurement results from the QUIC tunnel confirm that the Reno and CUBIC algorithms can keep the RTT lower than in a combination with BBR. In this measurement sequence, there is no such effect that a single connection takes up twice the bitrate of the other connection.

Finally, this subchapter shows that there are differences of the fairness when comparing several congestion control algorithms. Even though some differences are observed and discussed it can be concluded by applying the QUIC tunneling there is no major impact on the stream distribution. It in fact could be of interest to check further protocols within this QUIC tunnel for fairness.

3 Conclusion and Future Work

3.1 Conclusion

This work focus on TCP applied within QUIC tunneling and analysis its characteristics by using it in an experimental environment. The implementation of the QUIC tunnel, as presented in this paper, is based on the published draft of the masque IETF group. This mentioned draft is investigated with the datagram extension of QUIC. Through that, it was possible to prove that TCP streams can be successfully transmitted through a QUIC tunnel. The major benefit of using the datagram extension compared to the default QUIC protocol, as described in subchapter 2.1.4 and 2.2.1, is the reduction of overhead risk. Thus, the application is developed in Rust, since useful libraries are already available. Specifically the libraries *Quinn*, which provides the datagram extension, and *h3* are integrated in the client server application. The runtime environment *Tokio* was used in combination with the asynchronous programming model. It is to mention that the implementation of the interface between the network stack of the operation system and the developed application, TUN is applied. By that, the possibility of reading and writing of IP packets is given. The combination of the above mentioned together with three containers is then declared as the experiment environment in this paper.

The focus of the investigation of the results obtained through the experimental environment lies mainly on the two buffers and the identified queue. These components are received during the development and the first analysis of the results. On closer inspection, it could be verified in particular that increased size of the buffer and queue is not resulting in any advantage. This statement is correct unless it is a short burst that can be processed immediately.

The buffers and queues should be prevented from dropping packets. But not by enlarging them, but by enabling the responsible functions to empty these buffers in a short period

of time. The measurement series confirm that increasing the transmit queue has no negative effect on RTT as well as on the bitrate.

Considering the fairness it is focused on the behaviour of the congestion control algorithms CUBIC, Reno and BBR. The algorithms are executed concurrently with different numbers of streams in the tunnels. The achievable bitrates and RTT of the individual connections are logged and documented. In the series of measurements with CUBIC and Reno, the previously determined maximum bitrate, as described in subchapter 2.5.3, is achieved without one specific stream receiving significantly more than the others. The series of measurements with BBR shows that this algorithm consumes way more of the bandwidth. Only when several BBR connections use the bandwidth, it is divided fairly.

As mentioned in the subchapter 2.1.1, BBR is supposed to use the RTT as a feedback signal in the decision making process. Interestingly, during the fairness review it turned out that BBR together with Reno and CUBIC in the same connection is not able to keep the RTT low. However, it was observed that all three algorithms within the QUIC tunnel did not prevent other connections from sending data.

In conclusion it can be said that the experiments show that a transmission is possible without the core elements having a negative effect on the congestion control and friendliness of TCP. Furthermore, the intention of this work was to investigate the possibility of the application of QUIC for outdated application. Along the development of this work there are no issues identified and therefore it is claimed as an alternative approach to update existing software.

3.2 Future Work

Within this work, a research framework was set at the beginning which is described in subchapter 2.4. Nevertheless, there are further opportunities to investigate this subject. One main focus is to be set on the implementation and the use of buffers. Specifically the size and appropriate emptying is crucial for further improvements. The fact that increasing the size of the datagram-send buffer results in the negative effect, that less data is sent, should be part of future work. This phenomenon is discussed by the used measurement series and presented in the Tables 2.1, 2.3 and 2.2. When finding a sufficient solution, this can lead to changes in the applied *Quinn* implementation. This also applies to the UDP receiver buffers. The *Quinn* implementation must be able to empty it faster

or on a more regular basis. In both of these cases, the library is the slowest element in the system. Increasing the size of the TUN queue has no negative effect, but the implementation in the prototype can still be optimised. The queue is then flushed more effectively, and avoids the current case of the default configuration, where the queue fills up and drops packets.

The presented experiment environment can be also further extended by a more complex network topology. Especially the implementation of a topology which is characterised by a more heterogen network.

Further, a recommendation can be given to investigate other protocol tunneling concepts with the same/similar methodology. Then, a comparison to the presented results within this work is interesting.

Whilst looking for the lost packets in the chapter 2.5.2 it was discovered following: although the QUIC packets were decrypted by Wireshark, there was no dissector that extracted the packets type of the RFCs similarly as the tunnelled TCP. In order to simplify the research of QUIC tunnels, a dissector for Wireshark should be implemented in the future work.

Bibliography

[1] Begue, Jean-Christophe: *H3*. – URL https://github.com/hyperium/h3. – Zugriffsdatum: 2022-03-08. – original-date: 2016-09-09T22:31:36Z

[2] Clark, David D. ; Minshall, Greg ; Zhang, Lixia ; Peterson, Larry ; Ramakrishnan, K. K. ; Wroclawski, John T. ; Shenker, Scott ; Partridge, Craig ; Crowcroft, Jon ; Braden, Robert T. ; Deering, Steve E. ; Floyd, Sally ; Davie, Bruce S. ; Jacobson, Van ; Estrin, Deborah: *Recommendations on Queue Management and Congestion Avoidance in the Internet*. – URL https://datatracker.ietf.org/doc/rfc2309. – Zugriffsdatum: 2022-08-14. – Num Pages: 17

[3] Coonjah, Irfaan ; Catherine, Pierre C. ; Soyjaudah, K. M. S.: Experimental performance comparison between TCP vs UDP tunnel using OpenVPN. In: *2015 International Conference on Computing, Communication and Security (ICCCS)*, S. 1–5

[4] Coonjah, Irfaan ; Catherine, Pierre C. ; Soyjaudah, K. M. S.: Performance evaluation and analysis of layer 3 tunneling between OpenSSH and OpenVPN in a wide area network environment. In: *2015 International Conference on Computing, Communication and Security (ICCCS)*, S. 1–4

[5] Huston, Geoff: *BBR, the new kid on the TCP block*. – URL https://blog.apnic.net/2017/05/09/bbr-new-kid-tcp-block/. – Zugriffsdatum: 2022-08-13

[6] Krasnyansky, Maxim: *Universal TUN/TAP device driver*. – URL https://www.kernel.org/doc/Documentation/networking/tuntap.txt. – Zugriffsdatum: 2022-03-08

[7] Lackorzynski, Tim ; Köpsell, Stefan ; Strufe, Thorsten: A Comparative Study on Virtual Private Networks for Future Industrial Communication Systems.

In: *2019 15th IEEE International Workshop on Factory Communication Systems (WFCS)*, S. 1–8

[8] Lerche, Carl: *Tokio*. – URL https://github.com/tokio-rs/tokio. – Zugriffsdatum: 2022-03-08. – original-date: 2016-09-09T22:31:36Z

[9] Ochtman, Dirkjan: *quinn-rs/quinn*. – URL https://github.com/quinn-rs/quinn. – Zugriffsdatum: 2022-03-08. – original-date: 2018-04-03T07:47:41Z

[10] Pauly, Tommy ; Kinnear, Eric ; Schinazi, David: *An Unreliable Datagram Extension to QUIC*. – URL https://datatracker.ietf.org/doc/rfc9221. – Zugriffsdatum: 2022-05-11. – Num Pages: 9

[11] Pauly, Tommy ; Schinazi, David ; Chernyakhovsky, Alex ; Kühlewind, Mirja ; Westerlund, Magnus: *IP Proxying Support for HTTP*. – URL https://datatracker.ietf.org/doc/draft-ietf-masque-connect-ip. – Zugriffsdatum: 2022-05-12. – Num Pages: 20

[12] Q-Success: *Usage Statistics of QUIC for Websites, September 2022*

[13] Schinazi, David ; Pardue, Lucas: *HTTP Datagrams and the Capsule Protocol*. – URL https://datatracker.ietf.org/doc/draft-ietf-masque-h3-datagram. – Zugriffsdatum: 2022-05-12. – Num Pages: 16

[14] Seemann, Marten: *UDP Receive Buffer Size · lucas-clemente/quic-go Wiki*. – URL https://github.com/lucas-clemente/quic-go. – Zugriffsdatum: 2022-07-07

[15] Welzl, Michael: *Network Congestion Control: Managing Internet Traffic*. 1. edition. Wiley. – ISBN 978-0-470-02528-4

A Appendix

A.0.1 Experiment Setup Docker Compose Configuration

The Listing A.1 represents the Docker Compose file that is used to create the three containers for the experiment environment. The document additionally describes the two networks which connect the client and server via the emulator. In order to create an TUN device, more capabilities had to be added to the containers.

Listing A.1: Experiment Docker Compose configuration

```
version: "3.9"
services:
  emulator:
    build:
      dockerfile: Dockerfile.emulator
      context: .
    command: /bin/sh -c "while sleep 1000; do :; done"
    cap_add:
      - NET_ADMIN
    networks:
      leftnet:
        ipv4_address: 193.167.0.2
      rightnet:
        ipv4_address: 193.167.100.2
  client:
    build:
      context: .
      dockerfile: Dockerfile
      target: client
    depends_on:
```

```
21        - server
22        - emulator
23      command: >
24        sh -c "ip␣route␣add␣193.167.100.0/24␣via␣193.167.0.2
25 ␣␣␣␣␣␣&&␣RUST_LOG=warn,connect-ip::client=debug
26 ␣␣␣␣␣␣connect-ip␣https://server4:4433/vpn"
27      cap_add:
28        - NET_ADMIN
29        - SYS_PTRACE
30        - CAP_SYS_ADMIN
31      privileged: true
32      security_opt:
33        - seccomp:unconfined
34      devices:
35        - /dev/net/tun:/dev/net/tun
36      networks:
37        leftnet:
38          ipv4_address: 193.167.0.100
39      extra_hosts:
40        - "server4:193.167.100.100"
41    server:
42      build:
43        context: .
44        dockerfile: Dockerfile
45        target: server
46      command: >
47        sh -c "ip␣route␣add␣193.167.0.0/24␣via␣193.167.100.2
48 ␣␣␣␣␣␣&&␣RUST_LOG=warn␣connect-ip␣-s"
49      environment:
50        - SSLKEYLOGFILE=/var/log/connect-ip/sslkeylogfile.txt
51      cap_add:
52        - NET_ADMIN
53        - SYS_PTRACE
54      security_opt:
55        - seccomp:unconfined
56      devices:
```

```
57        - /dev/net/tun:/dev/net/tun
58      networks:
59        rightnet:
60          ipv4_address: 193.167.100.100
61  networks:
62    leftnet:
63      driver: bridge
64      driver_opts:
65        com.docker.network.bridge.enable_ip_masquerade: 'false'
66      ipam:
67        config:
68          - subnet: 193.167.0.0/24
69    rightnet:
70      driver: bridge
71      driver_opts:
72        com.docker.network.bridge.enable_ip_masquerade: 'false'
73      ipam:
74        config:
75          - subnet: 193.167.100.0/24
```

A.0.2 Monitoring Script

The shell script in Listing A.2 is used to read the TCP congestion control data. To do this, an infinite loop is executed which gets the data at regular frequencies using the command line tool *ss*. This data is written to a file together with a timestamp. If the script is terminated by a kill signal, a shutdown process is executed. The sequence parses the relevant data from the collected information and writes it to a csv file.

Listing A.2: Monitoring script to read congestion control values

```
#!/bin/bash
trap cleanup SIGINT SIGTERM
rm -f sender-ss.txt && touch sender-ss.txt

cleanup () {
  echo "time,cwnd,ssthresh,rtt,rttvar,b_sent,b_acked"\
          > sender-ss.csv
  ts=$(cat sender-ss.txt |  grep "unacked" \
          | awk 'NR==1{D=$1}{$1=($1-D);print$1}')
  cwn=$(cat sender-ss.txt |  grep "unacked" \
          | grep -oP '\bcwnd:\d*\sssthresh:\d*\b' \
          | awk -F '[: ]' '{print$2","$4}')
  rtt=$(cat sender-ss.txt |  grep "unacked" \
          | grep -oP '\brtt:\d*\.\d*/\d*\.\d*\b' \
          | sed "s/rtt://g" \
          | awk -F '[/]' '{print$1","$2}')
  bytes_sent=$(cat sender-ss.txt |  grep "unacked" \
          | grep -oP '\bbytes_sent:\d*\b' \
          | awk -F '[: ]' '{print$2}')
  bytes_acked=$(cat sender-ss.txt |  grep "unacked" \
          | grep -oP '\bbytes_acked:\d*\b' \
          | awk -F '[: ]' '{print$2}')
  paste -d ',' <(printf %s "$ts") <(printf %s "$cwn") \
          <(printf %s "$rtt") <(printf %s "$bytes_sent") \
          <(printf %s "$bytes_acked") >> sender-ss.csv
  exit 0
}

while [ 1 ]; do
  ss --no-header -ein dst $1 \
          | awk '/<->$/{printf("%s\t",$0);next}1' \
          | ts '%.s' >> sender-ss.txt
done
```

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original