Chair of Network Architectures and Services
School of Computation, Information, and Technology
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

SCHOOL OF COMPUTATION, INFORMATION, AND TECHNOLOGY

INFORMATICS

BACHELOR'S THESIS IN INFORMATICS

Analyzing the Effect of Transport Parameters on QUIC‘s Performance

Simon Karan Guayana

TECHNICAL UNIVERSITY OF MUNICH

SCHOOL OF COMPUTATION, INFORMATION, AND TECHNOLOGY

INFORMATICS

Bachelor's Thesis in Informatics

Analyzing the Effect of Transport Parameters on QUIC's Performance

Analyse der Auswirkung von Transportparametern auf die Leistung von QUIC

Author:	Simon Karan Guayana
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Johannes Zirngibl
	Benedikt Jaeger
Date:	February 15, 2023

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

Garching, February 15, 2023	
Location, Date	Signature

Abstract

The QUIC protocol aims to improve user experience by replacing the TCP + TLS + HTTP/2 stack. Among many features, such as optimized handshakes and connection migration, QUIC introduces transport parameters that are negotiated by the client and server during connection establishment. Previous research on the protocol's Internet deployments shows that a wide variety of transport parameter sets are being advertised with values ranging across multiple orders of magnitude. QUIC transport parameters define potentially impactful values, such as flow control limits or the acknowledgement frequency.

This work explains, how and in which conditions, the advertised transport parameters impact the performance of QUIC connections. We analyze the parameters used by popular web browsers and those used by QUIC servers on the Internet. We show that for the LSQUIC implementation, the transport parameters do not have a major impact on connections in a network with perfect conditions, but when introducing packet delay we identify that some parameters advertised by the client can bottleneck the goodput of a connection.
In a network with 30 ms delay, setting the client's `initial_max_data` transport parameter to values notably lower than 6.29 MB can reduce the goodput of a file download by more than 90%. Lowering the parameter to 196.61 kB decreases the goodput from 519.03 Mbit/s down to 31.93 Mbit/s.
We also find that in a network with high packet delay, a QUIC client configuration that advertises the transport parameters used by the Mozilla Firefox browser achieves higher throughput in a file download than a configuration using Google Chrome's parameters. This increase of around 170 Mbit/s in the throughput is caused by the different `intial_max_stream_data_bidi_local` values used.

Based on these results, it could be of interest to further investigate the impact of these parameters on different use cases, on other QUIC implementations and to keep track of future protocol extensions which may introduce new performance-relevant transport parameters.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The QUIC protocol, initially designed and implemented by Google, is a connection-oriented, reliable and cryptographically secure transport layer protocol. The protocol also implements features from higher layers of the International Organization for Standardization (ISO)/Open Systems Interconnection (OSI) model [1]. QUIC was firstly introduced back in 2012 [2], and it was standardized by the Internet Engineering Task Force (IETF) in May 2021 with the release of RFC 9000 [3]. The protocol was introduced to address the limitations of the traditional Transmission Control Protocol (TCP), Transport Layer Security (TLS) and Hypertext Transfer Protocol (HTTP)/2 stack. Some of QUIC's main advantages are: the optimized handshake, which reduces connection establishment times; the introduction of connection migration, which allows a connection to be IP address- and port-independent; it solves the Head-of-Line (HOL) blocking problem and it prevents ossification by being implemented in user space.

During QUIC's initial handshake, cryptographic and transport parameters are exchanged and negotiated between a QUIC client and server. As suggested by Zirngibl et al. [4] the values used for QUIC's transport parameters could allow analyzing current deployments of the protocol in more detail. There is related work on the topic of QUIC's transport parameters, but they mainly focus on exploring the impact of a proposed protocol extension that introduces a new type of transport parameter [5] and on checking for conformance to the RFC in interoperability tests [6]. This Bachelor's Thesis aims to analyze different parameter sets used by actual endpoints on the Internet as well as understand their effect on the protocol's performance. This way, clarity on their impact when considering the future support and evaluation of the protocol could be achieved [4].

1.1 Research Questions

We pretend to answer the following research questions with this work:

- Which transport parameters are used by QUIC deployments on the Internet?
- Do these parameters have an effect in the first place?
- If they do, how do they affect the performance?

1.2 Outline

We divide this work into different chapters. Chapter 2 serves the purpose of transmitting the basic knowledge of the transport layer and the QUIC protocol is introduced in more detail. This chapter also contains a description of the QUIC's transport parameters, which will be used throughout this work. In Chapter 3 related work is presented; this includes research on the topic of QUIC transport parameters, as well as useful discoveries that help us configure and optimize our testing setup. The methodology of this work is explained in detail in Chapter 4. The methodology chapter provides insight into the procedures used to collect the transport parameter data used in this work. It describes our measurement environment and presents our chosen QUIC reference implementation. It also introduces the QUIC Interop Runner and our contributions to it as well as the techniques we used to perform our measurements. The obtained results and findings are presented in various plots and tables and are analyzed in detail in Chapter 5. The observed results are summarized and evaluated in order to answer our research question in Chapter 6. Additionally, we propose future work.

Chapter 2

Background

The background chapter serves the purpose of transmitting the basic knowledge of the transport layer in Section 2.1. The QUIC protocol is introduced in more detail, along with the main features that differentiate it from the TCP protocol in Section 2.2. Then the QUIC's transport parameters are presented in more detail. Finally, the LSQUIC implementation is introduced.

2.1 Transport Layer

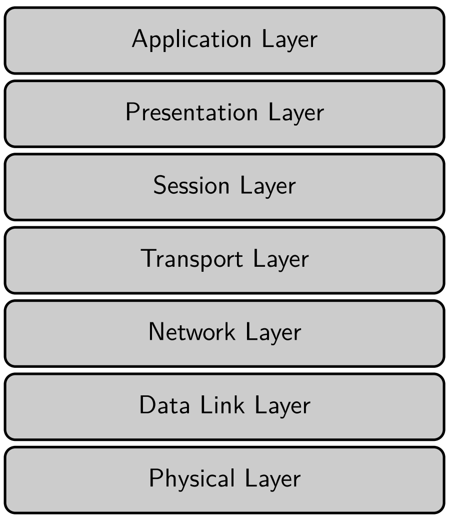


Figure 2.1: ISO/OSI model.

The transport layer is one of the seven layers defined in the ISO's OSI model [1]. This model, seen in Figure 2.1, classifies the different networking protocols according to their functionality and their purpose. In this work, we focus on layer-four protocols mainly. This layer is responsible for providing end-to-end communication between one or multiple programs in a system, i.e., it multiplexes and demultiplexes the data received from the layers below to the layers above. In practice, this is achieved with the source port and destination port fields in the headers of the different protocols. Depending on the actual transport layer protocol, other functionalities such as flow control, congestion control and reliable data transfer can also be implemented. Historically, there have been two transport layer protocols, the more widespread TCP [7], and the much simpler User Datagram Protocol (UDP). [8]

2.2 QUIC

QUIC is considered a transport layer protocol, it is built at the user level on top of UDP as shown in Figure 2.2. This is beneficial because existing middleboxes are by default already compatible. From their point of view, QUIC traffic is just UDP traffic. The user-level implementation of the protocol also allows faster development and rolling out updates is less complicated, but can lead to less optimization in general.

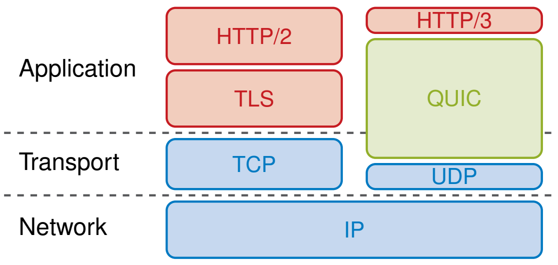


Figure 2.2: TCP and QUIC protocol stacks.

One way in which QUIC optimizes the classic TCP+TLS stack is accomplished by integrating TLS 1.3 directly into the protocol and thus increasing efficiency. By merging the TCP and TLS handshakes into a single handshake, QUIC can save up to one complete round-trip-time (RTT) when establishing a new connection. The possibility of performing 0-RTT handshakes is also introduced with the protocol. This 0-RTT handshake is performed when the client uses previously acquired encryption details from the server (in a previously established connection) to send data instantly with the first packet sent. It is important to note, that the data sent in 0-RTT handshake packages are susceptible to replay attacks. [9]

Another problem that QUIC addresses is the HOL blocking problem. This is solved by implementing the concept of streams directly in the transport layer and transmitting each resource in a separate and uniquely identified stream (using `STREAM` frames).

When packet loss occurs, not all resources are held back until the missing packet is re-transmitted, just the needed `STREAM` frames from the affected resource are re-transmitted, while the unaffected data can be passed to the upper layered protocols instantly.

QUIC also introduces the concept of connection migration, which means, that a connection is independent of the IP address and the port being used. Connections are no longer identified by the classic 4-tuple (source IP, source port, destination IP, destination port), instead, the source- and destination connection ID fields in the protocol's headers are used. This change is beneficial, especially for mobile devices which encounter frequent changes to their IP address (e.g., changing from a wi-fi connection to a cellular connection), because connections are kept alive and the overhead of a connection establishment is saved.

2.2.1 Transport Parameters

QUIC's transport parameters are exchanged between the client and server during the initial handshake. The protocol's transport parameters are encapsulated in a `quic_transport_parameters` extension in the TLS 1.3 `ClientHello` and `EncryptedExtensions` messages, sent by the client and server respectively during the initial handshake. The declaration of these parameters is done independently of the other endpoint's choice. The use of 0-RTT depends on the parameters previously negotiated between the client and server. Not only cryptographic details but also values of stored transport parameters are used when performing this type of handshake.

According to the Internet Assigned Numbers Authority (IANA), there are currently 20 different transport parameters assigned to be 'permanent' and 6 other parameters which are assigned as 'provisional' by the time of writing this work [10]. The firstly introduced permanent parameters are defined in RFC 9000 [3] section 18.2 and these are:

1. `original_destination_connection_id`: This parameter has the same value as the Destination Connection ID field from the first Initial packet sent by the client, this is done to authenticate Connection IDs.
2. `max_idle_timeout`: Connection timeout in ms. If a connection stays idle for longer than the minimum value advertised with this parameter by the client and the server, the connection is closed.
3. `stateless_reset_token`: This parameter is used to verify a stateless reset. A stateless reset can be conducted when a peer in a connection can no longer associate a received packet to an active connection due to e.g., a crash. This parameter may only be sent by a server.

4. `max_udp_payload_size`: This parameter advertises the size of the largest UDP payload in B that the endpoint is willing to receive. This constraint is similar to the one imposed by the path Maximum Transmission Unit (MTU), but it is not dependent on the path itself but on the actual endpoints.

5. `initial_max_data`: The initial max data parameter indicates the maximum amount of data allowed to be transmitted throughout the connection as a whole in B. This value is used for flow control but can be updated later in the connection with a `MAX_DATA` frame.

6. `initial_max_stream_data_bidi_local`: The value of this parameter sets the flow control limit in B for bidirectional streams initiated locally and applies to the streams created by the endpoint that advertises the parameter.

7. `initial_max_stream_data_bidi_remote`: This parameter is analogous to the `initial_max_stream_data_bidi_local` parameter, but applies to the streams created by the endpoint that receives the parameter.

8. `initial_max_stream_data_uni`: This value sets the initial flow control limit in B for unidirectional streams and applies to streams initiated by the endpoint that receives the parameter.

9. `initial_max_streams_bidi`: This integer value limits the maximum amount of bidirectional streams that are allowed to be created by the receiver of the parameter.

10. `initial_max_streams_uni`: This parameter is analogous to the `initial_max_streams_bidi` parameter, but for unidirectional streams.

11. `max_ack_delay`: This value indicates the maximum time delay that an endpoint may have before sending acknowledgments

12. `ack_delay_exponent`: This parameter is used to decode the ACK Delay field in `ACK` frames. This parameter allows having a larger range of values that can be used in the ACK Delay field.

13. `disable_active_migration`: This parameter is sent if the endpoint does not support active connection migration.

14. `preferred_address`: When a server advertises this parameter, the server changes the IP address (and port) to a ‘preferred’ one once the handshake is completed. This parameter may only be sent by servers.

15. `active_connection_id_limit`: This value limits the number of connection IDs that an endpoint can store. A QUIC connection has a set of connection IDs to identify the connection. Multiple connection IDs are used during a connection to avoid packets being identified to the same connection by a third party. Therefore, the minimum value for this parameter must be 2. After the handshake, new connection IDs can be issued and retired with the `NEW_CONNECTION_ID` and `RETIRE_CONNECTION_ID` frames.

16. `initial_source_connection_id`: This is the same value that the sender included in the Source Connection ID field on the connection's very first Initial packet.

17. `retry_source_connection_id`: The value of this parameter is the same as the one sent by the server in the Source Connection ID field of a Retry packet. It is used to authenticate connection IDs. A server can respond to a client's Initial packet with a Retry packet and the client will then send another Initial packet with the connection ID advertised in the Retry packet. This can be useful to redirect to another server as explained in [11]. The parameter is only sent by servers.

Other transport parameters have been introduced later in different RFCs:

1. `max_datagram_frame_size`: This transport parameter is introduced with the unreliable datagram extension in RFC 9221 [12]. Values greater than 0 indicate support for `DATAGRAM` frames and the maximum size of a `DATAGRAM` frame willing to be received.

2. `grease_quic_bit`: This transport parameter is introduced with RFC 9287 [13], and the presence of this parameter indicates that the 'QUIC Bit' (the second-most significant bit of the first Octet of QUIC packets) is allowed to be used for other purposes other than to always set this bit to 1 to effectively distinguish QUIC from other protocols.

3. `version_information`: The Internet-Draft (by February 2023) 'Compatible Version Negotiation for QUIC' [14] introduces this new transport parameter which carries the information of the chosen QUIC version for the current connection and a list of the endpoint's available versions sorted by preference. Even though this RFC has still not been released, IANA already classifies this parameter as permanent.

2.2.2 LSQUIC

LiteSpeed QUIC (LSQUIC) is an open-source implementation of the QUIC protocol [15]. LSQUIC is the chosen reference QUIC implementation for this work. This implementation was selected due to it being implemented in the performant C programming language, for its flexibility when configuring and for being one of the most deployed implementations as observed by Zirngibl et al. [4]. The LSQUIC library also provides an HTTP client and server which are also used during this work to test the protocol. The client and server are highly customizable with so-called 'engine settings', these settings let us set different values for the following transport parameters:

- `initial_max_data`,
- `initial_max_stream_data_bidi_local`,
- `initial_max_stream_data_bidi_remote`,
- `initial_max_stream_data_uni`,
- `initial_max_streams_bidi`,
- `initial_max_streams_uni`,
- `max_idle_timeout`,
- `max_udp_payload_size` and
- `grease_quic_bit`.

All other parameters are not configurable as engine settings. There are also other interesting configurable settings, such as the congestion control algorithm. LSQUIC offers three different options for congestion control: CUBIC, BBRv1 and an adaptive mechanism that switches between CUBIC and BBRv1 depending on the RTT.

LSQUIC uses different default values for the client's and server's transport parameters. These default values are presented in Table 2.1.

Table 2.1: LSQUIC's default transport parameter values for the client and server.

Transport Parameter	Client	Server
`max_idle_timeout`	30000	30000
`max_udp_payload_size`	65527	65527
`initial_max_data`	15728640	1572864
`initial_max_stream_data_bidi_local`	6291456	0
`initial_max_stream_data_bidi_remote`	0	1048576
`initial_max_stream_data_uni`	32768	12288
`initial_max_streams_bidi`	100	100
`initial_max_streams_uni`	100	3
`ack_delay_exponent`	3	3
`max_ack_delay`	25	25
`disable_active_migration`	False	False
`active_conn_id_limit`	8	8

Chapter 3

Related Work

This chapter introduces relevant related work on the topic of QUIC's transport parameters and in LSQUIC's performance in particular.

Wolsing et al. [16] present a comparison between QUIC and TCP. This comparison is done by 'tuning' TCP parameters such as the congestion window and buffer sizes and thus achieving significant improvements in the performance of the protocol. It is shown, that in studies where QUIC outperforms the TCP stack (TCP, TLS, and HTTP), the comparisons are done with default TCP configurations. The authors claim, that large content providers adjust TCP's parameters to offer better performance, therefore this study focuses on making a fair comparison between an adjusted and finely-tuned TCP implementation and the QUIC protocol, which is optimized for today's networks and devices by design. They conclude that QUIC offers superior performance when compared to optimized TCP, mainly because of the more efficient connection establishment, but the performance difference is smaller when compared to an unoptimized configuration. This finding further rises the question if tuning QUIC's parameters can produce a similar impact and improve performance, which we look at in our work.

Experimentation with QUIC's transport parameters has been done already. Volodina and Rathgeb's work [5] explores a protocol extension draft that introduces the new frame type 'ACK frequency frame' and a new transport parameter `min_ack_delay` to dynamically adapt the acknowledgement (ACK) ratio [17]. They show the impact of adjusting the ACK ratio with the new transport parameter and frame type by benchmarking the performance under different network conditions. A similar testing approach is performed in our work to study the protocol's transport parameters while taking into account the values used in actual QUIC deployments.

In the paper by Piraux et al., [6] early QUIC implementations' interoperability is tested. Among the protocol features that are tested, the protocol's flow control tests are interesting. They advertise a small value for the `initial_max_stream_data_bidi_local` parameter (80 B) to check if the servers would respect this limit and stop sending the data. The researchers would then update the limit with `MAX_STREAM_DATA` frames to see that the servers continued transmitting the data. In their findings, they noticed that some implementations did not behave as expected. This work sets different values for the mentioned transport parameter and evaluates its effect on the protocol's behavior, but does not consider the impact on the performance that values used by Internet deployments may produce.

The work by Zirngibl et al. [4] analyzes QUIC deployments on the Internet in more detail. For this study, Internet-wide scans were conducted to retrieve information about active QUIC deployments by performing stateful scans. The authors evaluate the transport parameter configurations advertised by the different scan targets. On the one hand, they found parameters such as `active_connection_id_limit`, `max_ack_delay` and `ack_delay_exponent` for which default values are commonly used. On the other hand, parameters that can potentially influence performance, have large deviations in the values used. For `max_udp_payload_size`, `initial_max_data` and `initial_max_stream_data` seen values range between 1500 B and 65 527 B for the first parameter, between 8192 B and 16 777 216 B for the second one and between 32 000 B and 10 000 0000 B for the last one. The researchers collected 45 different transport parameter sets that QUIC servers were advertising throughout the Internet. The authors argue that a server's advertised transport parameters can help to analyze QUIC deployments and their differences and that configuring these parameters can have an impact on QUIC connections.

The master's thesis by Kempf [18] is of great interest to our work because it takes a deep dive into the performance of QUIC implementations, particularly that of LSQUIC. The author observed that in the case of the LSQUIC implementation, the choice of the congestion control algorithm causes significant performance differences. The BBR implementation did not keep up with CUBIC when it came to goodput in the measurements. Additionally, the UDP receive buffer sizes can affect performance. It was concluded, that the buffer size is not enough for high bandwidth usage, since a significant amount of dropped packets was detected. Furthermore, it was identified, that there is a large deviation in ACK frequency between measurements. Unnecessarily many ACKs resulted in additional overhead for the server, and thus it affects performance. The work by Kempf also contributed to the QUIC Interop Runner used in our work [19], especially by extending its functionality. Support to run scripts on the client and server before and after measurement was added, passing environmental variables to the client and server is now possible, and support for `YAML` configuration files was introduced.

Chapter 4

Methodology

This chapter introduces the procedures used to produce the results and findings during this thesis. It includes information on the data collection methods and the measurements performed in this work.

4.1 Transport Parameters Data

In this section, we explain how we get the data of transport parameters used by QUIC deployments on the Internet.

4.1.1 Internet Wide Scans

In order to get an insight into the transport parameters used by QUIC servers on the Internet, we analyze the results from two Internet-wide scans like the ones performed by Zirngibl et al. [4]. The first scan was performed on January 27, 2022; the second scan on October 13, 2022. The raw data analyzed consists of the transport parameters advertised by IPv4 QUIC servers. We filter out connection-specific parameters to analyze the data. Connection-specific parameters are:

- `original_dst_conn_id`,
- `stateless_reset_token`,
- `preferred_address`,
- `initial_src_conn_id` and
- `retry_src_conn_id`.

The parameters we analyze from the scans in more detail are:

- `max_idle_timeout`,
- `max_udp_payload_size`,
- `initial_max_data`,
- `initial_max_stream_data_bidi_local`,
- `initial_max_stream_data_bidi_remote`,
- `initial_max_stream_data_uni`,
- `initial_max_streams_bidi`,
- `initial_max_streams_uni`,
- `ack_delay_exponent`,
- `max_ack_delay`,
- `disable_active_migration` and
- `active_conn_id_limit`.

The most common parameter combinations were extracted and we also present the parameters' most commonly used values.

4.1.2 Internet Browsers

It is also in the interest of our work, to analyze transport parameters advertised by popular QUIC clients. We extract the transport parameters used by two different browsers:

- Google Chrome for Windows 109.0.5414.119 (64-bit)
- Mozilla Firefox for Windows 109.0 (64-bit)

To get this data, we host a QUIC server and then enforce the browsers to use QUIC as the transport layer protocol when trying to connect to our server's IP address. Both browsers have different ways to force the use of QUIC and HTTP/3. For the Chrome browser, the options shown in Listing 4.1 have to be present when executing the browser from the command line.

Listing 4.1: Options to force QUIC in Google Chrome

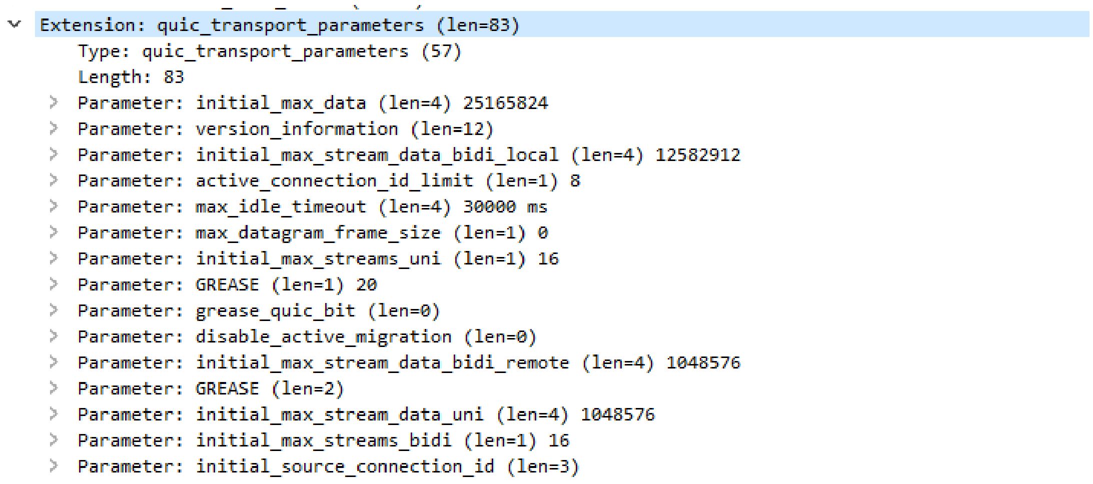
```
chrome --enable-quic --origin-to-force-quic-on=<server-hostname>:443
```

The Firefox browser has a configuration option to force QUIC on selected hosts. To enable it, it is necessary to type `about:config` in the address bar. Then, in the preference search bar, search for the `network.http.http3.alt-svc-mapping-for-testing` option and add an entry in the format seen in Listing 4.2. Finally, save and restart the browser.

LISTING 4.2: Entry in the `network.http.http3.alt-svc-mapping-for-testing` Firefox's option to force a QUIC connection.

```
<server-hostname>;h3=":443";h3-29=":443"
```

We capture the packets sent by the client to our server using Wireshark [20] and inspect their content. The client's transport parameters in the Initial packet are not confidentiality protected and, as explained in Section 2.2.1, the client's transport parameters are selected independently from the server's configuration. Therefore, we can simply read the values from the packet as seen in Figure 4.1.



```
Extension: quic_transport_parameters (len=83)
    Type: quic_transport_parameters (57)
    Length: 83
  > Parameter: initial_max_data (len=4) 25165824
  > Parameter: version_information (len=12)
  > Parameter: initial_max_stream_data_bidi_local (len=4) 12582912
  > Parameter: active_connection_id_limit (len=1) 8
  > Parameter: max_idle_timeout (len=4) 30000 ms
  > Parameter: max_datagram_frame_size (len=1) 0
  > Parameter: initial_max_streams_uni (len=1) 16
  > Parameter: GREASE (len=1) 20
  > Parameter: grease_quic_bit (len=0)
  > Parameter: disable_active_migration (len=0)
  > Parameter: initial_max_stream_data_bidi_remote (len=4) 1048576
  > Parameter: GREASE (len=2)
  > Parameter: initial_max_stream_data_uni (len=4) 1048576
  > Parameter: initial_max_streams_bidi (len=1) 16
  > Parameter: initial_source_connection_id (len=3)
```

FIGURE 4.1: Firefox's transport parameters inspected with Wireshark.

4.2 MEASUREMENTS

To perform the measurements for this work, we use the blockchain testbed infrastructure provided by their Chair of Network Architectures and Services along with the QUIC Interop Runner's fork [19].

The topology of the setup consists of two test nodes (Uniswap and Solana) and a management node (Coinbase). One of the test nodes acts as the QUIC client and the other as the server. The two test nodes communicate over a 10 Gbit Ethernet link and both are controlled directly via the management node. From the management node, the test nodes are booted, configured (with the Debian 11 operating system) and controlled completely using the Plain Orchestrating Service (POS) [21]. Both test nodes have an Intel® Xeon® E51650 v3 CPU, 64 GB of memory and an Intel® 10G X550T network adapter.

The QUIC Interop Runner's original purpose is to test the interoperability between different QUIC implementations (by setting different implementations for client and server) under several test cases such as a handshake test, a version negotiation test or a 0-RTT test between others [22]. It can also be used to test the performance of single implementations with benchmarking tests. The Interop Runner's fork we use is adapted to run directly on the hardware and thus saves the overhead which would be introduced by using Docker (which the original QUIC Interop Runner uses).

To perform our measurements, we use the 'goodput' test. This test creates a file with a relatively large size (e.g., 500 MB or 1 GB) on the server node, then it downloads the file from the server to the client. It is made sure, that the file is downloaded correctly and then the throughput of the download is returned as result in Mbit/s. To analyze our results, we measure with the same parameters 10 or 20 times (depending on the measurement), and then we compute the mean of those measurements. After completing the measurements, we plot our results to compare them.

Additionally, in our measurements, we use Ethtool [23]. This tool extracts information from the network interface and get statistics of packets transmitted, packets received and packets dropped. The pre- and post-measurement scripts provided by Kempf [18] are used to generate this information before and after each measurement.

4.2.1 WAN Emulation

For this work, we extend the functionality of the QUIC Interop Runner. We implement a feature to emulate different network conditions when using the QUIC Interop Runner. TC's NetEm can be used to add delay on packets, add random packet loss, add random noise corruption (which introduces a single bit error at a random offset in a packet) and add random packet reordering. All of these options are configurable as command line arguments when executing the Interop Runner. TC's NetEm is configured on the client's network interface to impact ingress traffic. A model of this can be seen in Figure 4.2.

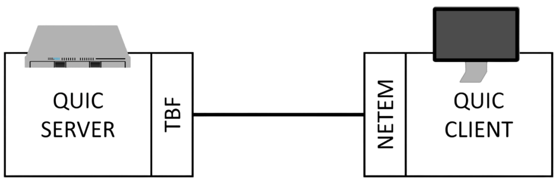


Figure 4.2: TBF on the server's interface and NetEm on the client's interface.

By default NetEm only affects egress traffic. The Interop Runner already had a feature to limit the bandwidth using TC's Token Bucket Filter (TBF) on the server's outgoing traffic. To avoid having to dynamically create and adapt a child queueing discipline

depending on the different possible usage combinations of TBF and NetEm, we opt to follow the approach of applying NetEm to the client's ingress traffic. It is important to note, that in our testing environment, NetEm and TBF have a significant impact when used for the server's outgoing traffic or the client's incoming traffic and not so much in the other traffic direction.
To apply NetEm's rules on ingress traffic, we use an Intermediate Functional Block (IFB) pseudo network interface to redirect the client's ingress traffic to the IFB. The IFB handles the client's ingress traffic as 'egress' traffic, so we can apply NetEm to the IFB and emulate different network environments.

4.2.2 LSQUIC Configuration

To use a specific QUIC implementation in the Interop Runner, three bash scripts are required:

- `setup-env.sh`, as the name suggests, the necessary environment to use the QUIC implementation is set up. For example, this can be the installation of requirements.
- `run-client.sh`, this script starts the client with any necessary options (such as setting the IP address).
- `run-server.sh`, the server is started when this script is executed.

For our work, we extend the `run-client.sh` and the `run-server.sh` scripts used for the LSQUIC implementation (which is our chosen reference implementation as explained in Section 2.2.2). We add the option to configure and set different values for the transport parameters that are advertised by the client and server. The values used for the parameters are passed to the script using POS environmental variables for the client and server separately. This allows us to modify the values being used for the transport parameters easier using configuration files. In case no POS variable is set for a parameter, the default value in the script is used. The default values in our scripts are the same default values seen in the LSQUIC source code. Nonetheless, this script lets us change the parameters' default values eventually. A snippet of the used script to run the client and set the values for the transport parameters can be seen in Listing 4.3.

Listing 4.3: Transport parameter configuration in run-client.sh

```
...
IDLE_TIMEOUT=$(pos_get_variable -r IDLE_TIMEOUT)
MAX_UDP_PAYLOAD=$(pos_get_variable -r MAX_UDP_PAYLOAD)
INITIAL_MAX_DATA=$(pos_get_variable -r INITIAL_MAX_DATA)
INITIAL_MAX_STREAM_DATA_BIDI_LOCAL=$(pos_get_variable -r
    INITIAL_MAX_STREAM_DATA_BIDI_LOCAL)
INITIAL_MAX_STREAM_DATA_BIDI_REMOTE=$(pos_get_variable -r
    INITIAL_MAX_STREAM_DATA_BIDI_REMOTE)
INITIAL_MAX_STREAM_DATA_UNI=$(pos_get_variable -r INITIAL_MAX_STREAM_DATA_UNI)
```

```
INITIAL_MAX_STREAMS_BIDI=$(pos_get_variable -r INITIAL_MAX_STREAMS_BIDI)
INITIAL_MAX_STREAMS_UNI=$(pos_get_variable -r INITIAL_MAX_STREAMS_UNI)

if [[ $TESTCASE == "goodput" ]]; then
    # Default values for the transport parameters
    # are the same defaults used in the lsquic implementation
    LSQUIC_O_FLAG=
        ...
        ‘"-o␣idle_timeout=${IDLE_TIMEOUT:-30}␣"‘
        ‘"-o␣max_udp_payload_size_rx=${MAX_UDP_PAYLOAD:-0}␣"‘
        ‘"-o␣init_max_data=${INITIAL_MAX_DATA:-15728640}␣"‘
        ‘"-o␣init_max_stream_data_bidi_local=${INITIAL_MAX_STREAM_DATA_BIDI_LOCAL
            :-6291456}␣"‘
        ‘"-o␣init_max_stream_data_bidi_remote=${
            INITIAL_MAX_STREAM_DATA_BIDI_REMOTE:-0}␣"‘
        ‘"-o␣init_max_stream_data_uni=${INITIAL_MAX_STREAM_DATA_UNI:-32768}␣"‘
        ‘"-o␣init_max_streams_bidi=${INITIAL_MAX_STREAMS_BIDI:-100}␣"‘
        ‘"-o␣init_max_streams_uni=${INITIAL_MAX_STREAMS_UNI:-100}"
    ./http_client \
        ...
        $LSQUIC_O_FLAG
```

For our measurements, we fix the LSQUIC congestion control algorithm to CUBIC. Furthermore, we increase our UDP receive buffers. As presented in Chapter 3, these changes made by Kempf [18] enable LSQUIC to deliver better performance overall.

We make use of `YAML` configuration files to create different transport parameter combinations for our measurements. In these scripts, we set values for the POS variables which are then read by the client and server using the run-client.sh and run-server.sh scripts respectively. An example configuration file is presented in Listing 4.4.

Listing 4.4: Example transport parameter configuration file

```
client_implementation_params:
- IDLE_TIMEOUT=120
- MAX_UDP_PAYLOAD=1472
- INITIAL_MAX_DATA=196608
- INITIAL_MAX_STREAM_DATA_BIDI_LOCAL=131072
- INITIAL_MAX_STREAM_DATA_BIDI_REMOTE=131072
- INITIAL_MAX_STREAM_DATA_UNI=131072
- INITIAL_MAX_STREAMS_BIDI=100
- INITIAL_MAX_STREAMS_UNI=103
server_implementation_params:
- IDLE_TIMEOUT=30
- MAX_UDP_PAYLOAD=1472
- INITIAL_MAX_DATA=16777216
- INITIAL_MAX_STREAM_DATA_BIDI_LOCAL=1048576
- INITIAL_MAX_STREAM_DATA_BIDI_REMOTE=32768
- INITIAL_MAX_STREAM_DATA_UNI=1048576
- INITIAL_MAX_STREAMS_BIDI=100
- INITIAL_MAX_STREAMS_UNI=10
```

Chapter 5

Results

This chapter presents the results of the analyzed data and the measurements performed according to the methodology presented in Chapter 4.

5.1 Transport Parameters Data

This section shows the data of the collected transport parameters. We get this data from two sources. The first one is from two Internet-wide scans performed on January 27, 2022 and on October 13, 2022. The second source is directly from inspecting QUIC packets sent by two web browsers.

5.1.1 Internet Wide Scans

The scan on January 27, 2022 (scan 22-01-27) reaches 290746 targets and we identify 91 different transport parameter combinations. For the scan on October 13, 2022 (scan 22-10-13), we recognize 148 different transport parameter sets for 516877 reached targets. We can see an increase in QUIC endpoints on the Internet, but we can also notice that more transport parameter combinations are in use.

We group the entries in our data by the transport parameters that we are interested in. On the one hand, the most common set in scan 22-01-27 (set α) is advertised by 68892 endpoints, which is 23,7% of all reached targets. On the other hand, the most common parameter combination in scan 22-10-13 (set β) is advertised by 148728 QUIC servers, this accounts for 28,8% of the scanned targets. Table 5.1 displays the values of the most common transport parameter combination for scan 22-01-27 and scan 22-10-13.

Table 5.1: Values of the most common transport parameter set for each scan.

Transport Parameter	Set α	Set β
`max_idle_timeout`	30000	120000
`max_udp_payload_size`	1472	1472
`initial_max_data`	16777216	196608
`initial_max_stream_data_bidi_local`	1048576	131072
`initial_max_stream_data_bidi_remote`	32768	131072
`initial_max_stream_data_uni`	1048576	131072
`initial_max_streams_bidi`	100	100
`initial_max_streams_uni`	10	103
`ack_delay_exponent`	10	3
`max_ack_delay`	25	25
`disable_active_migration`	False	False
`active_conn_id_limit`	4	0[a]

[a] The value 0 indicates that the default limit of 2 connection IDs is used.

We observe, that the most used combination changes between the two dates. In scan 22-01-27, set β is advertised by 50805 of the 290746 scanned targets (17,5%). This makes set β, the third most common combination in scan 22-01-27. Set α is advertised by 100115 of the 516877 reached endpoints in scan 22-10-13 (19,4%), making it the second most common parameter combination in scan 22-10-13.

We also set focus on the individual parameters. The five most common values for the `max_idle_timeout` can be seen in Table 5.2

Table 5.2: Five most common `max_idle_timeout` values for each scan

`max_idle_timeout`			
scan 22-01-27		scan 22-10-13	
Value in ms	# Endpoints	Value in ms	# Endpoints
30000	71829	120000	180489
180000	66735	30000	113033
300000	56088	180000	108323
120000	51037	300000	62047
240000	36140	240000	37760

The advertised values for this parameter vary but are all roughly within an order of magnitude (excluding some outliers which reach a value of up to 90 000 000 000 ms ≈ 1042 d). This parameter does not have a direct impact on the performance (throughput) of a connection due to the nature of the parameter, but it could give an insight into the purpose of the endpoint. Use cases where user interaction is expected could opt for longer or no idle timeouts at all.

QUIC endpoints on the Internet also show different values for the `max_udp_payload_size` transport parameter as seen in table Table 5.3.

Table 5.3: 5 most common `max_udp_payload_size` values for each scan

`max_udp_payload_size`			
scan 22-01-27		scan 22-10-13	
Value in B	# Endpoints	Value in B	# Endpoints
1472	212996	1472	350751
1404	30680	65527	94222
65527	25437	1404	38143
1500	18761	1500	20521
1452	2833	1452	12318

In total, there are 9 and 12 different values for this parameter seen in scans 1 and 2 respectively. The predominant value for this parameter is 1472 B, which is advertised by 73,3% of servers in scan 22-01-27 and by 67,9% in scan 22-10-13. In scan 22-10-13, we observe an increase in the use of the value 65 527 B between the two scan dates. This value is the protocol's maximum permitted payload size as well as its default value as defined in RFC 9000 [3]. The `max_udp_payload_size` transport parameter cannot have much of an effect on the performance (especially for larger values such as 65 527 B), since the size of the UDP payload is also dependant on the path's MTU. Theoretically, a low value for this parameter could hinder the performance, but the protocol's minimum allowed value is 1200 B. In the scans less than 1% of endpoints advertise values below 1404 B (i.e., 1350 B, 1280 B and 1200 B). Additionally, it is stated in the LSQUIC documentation, that this limit is not enforced for incoming packets. Therefore we assume that in the case of traditional Ethernet-based networks with an MTU of 1500 B, the impact of this parameter is not significant.

The `initial_max_data` transport parameter serves as a flow control limit on the total amount of data sent by the connection's streams. The performed scans demonstrate that there is a large number of different values being advertised for this parameter. In scan 22-01-27, there are 34 different values, while in scan 22-10-13, 50 can be seen. As seen in Table 5.4, the values used for this parameter vary within multiple orders of magnitudes.

Table 5.4: 10 most common `initial_max_data` values for each scan

`initial_max_data`			
scan 22-01-27		scan 22-10-13	
Value in B	# Endpoints	Value in B	# Endpoints
196608	143091	196608	248617
16777216	68949	16777216	100128
1048576	54926	1048576	53623
10485760	19058	137363456	31676
786432	2798	10485760	29516
50331648	1069	68681728	26797
8585216	341	786432	12204
34359738368	205	8585216	10861
16384	113	50331648	1068
2097152	60	10000000	770

The smallest advertised value for this parameter is 8192 B (observed 21 times in scan 22-10-13) and the largest is roughly 4 611 686 TB (observed by one endpoint in both scans). Since this is a potentially significant parameter for the performance of a connection, it is one of the focuses of this work to analyze it in more detail in Section 5.2.

The three stream-level flow control parameters are analyzed, i.e.:

- `initial_max_stream_data_bidi_local`,
- `initial_max_stream_data_bidi_remote` and
- `initial_max_stream_data_uni`.

These three parameters serve as flow control limits. In contrast to `initial_max_data`, they limit the data that can be sent on each stream instead of limiting the connection as a whole.
It should be remarked, that servers tend to advertise these parameters as a group. There are relatively few value combinations for this three-parameter set in relation to all the theoretically possible combinations. As seen in Table 5.5, more than 99% of the scanned QUIC servers (in both scans) advertise one of the 10 most common values sets for the maximum stream data transport parameters.

Table 5.5: 10 most common `initial_max_stream_data_{bidi_local, bidi_remote, uni}` values for each scan

initial_max_stream_data_{bidi_local, bidi_remote, uni}							
scan 22-01-27				scan 22-10-13			
bidi_local	bidi_remote	uni	$\sum^{a}$	bidi_local	bidi_remote	uni	$\sum^{a}$
131072	131072	131072	143151	131072	131072	131072	248668
1048576	32768	1048576	68892	1048576	32768	1048576	100124
67584	67584	67584	47598	67584	67584	67584	51583
0	1048576	1048576	17422	524288	524288	524288	39001
65536	65536	65536	7668	1048576	1048576	1048576	31678
524288	524288	524288	2798	0	1048576	1048576	23046
10485760	10485760	10485760	1638	65536	65536	65536	12906
3145728	3145728	3145728	1069	10485760	10485760	10485760	6470
16777216	16777216	16777216	211	3145728	3145728	3145728	1068
16384	16384	16384	113	1000000	1000000	0	766

Parameter values are in B

[a] Number of endpoints advertising the given three-parameter set.

As seen in Table 5.5, values for the `initial_max_stream_data_bidi_local` parameter varies by multiple orders of magnitude. Even some servers advertise 0 B as a limit. This parameter limits locally initiated bidirectional streams. On the first date, 23 different values are observed, while the second date presents 43 different values for this parameter. There are use cases where the server does not initiate any streams themselves (such as our file download measurement). In such cases, this server parameter is not relevant to the connection and therefore it might be set to 0 B. The largest seen parameter is

1073.7 MB and it is sent by 10 endpoints in scan 22-10-13.
The `initial_max_stream_data_bidi_remote` parameter limits stream-level flow control, analogous to `initial_max_stream_data_bidi_local`, but for streams initiated by the peer. The values for this parameter shown in Table 5.5, hint that this parameter is also given values that vary similarly to `initial_max_stream_data_bidi_local`. This parameter can potentially influence the connection's performance in our measurements and we take a look at it in more detail in Section 5.2. In the first scan, 27 different values are identified, while in the second scan, we can see 45 different values for this parameter. The smallest seen value for this parameter is 1000 B (by one endpoint in both scans) and the largest one is 1048.6 GB (by two endpoints in scan 22-01-27 and three endpoints in scan 22-10-13).
The `initial_max_stream_data_uni` transport parameter limits unidirectional streams opened by the receiver of the parameter. For this parameter, the values advertised are also very similar to those being advertised for `initial_max_stream_data_bidi_local` and `initial_max_stream_data_bidi_remote` as can be observed in Table 5.5. The first scan found 27 different values for this parameter and the second scan found 47. The smallest advertised value is 0 B (by 766 servers in scan 22-10-13) and the largest is also 1048.6 GB (by two endpoints in scan 22-01-27 and three endpoints in scan 22-10-13).

`initial_max_streams_bidi` and `initial_max_streams_uni` determine how many bidirectional and unidirectional streams the receiver of the parameter is allowed to initiate respectively. Again, for these parameters, we can observe a large variety of values being sent by servers on both dates. This is presented for `initial_max_streams_bidi` in Table 5.6. The large range of different values can also be seen for `initial_max_streams_uni` in Table 5.7.

Table 5.6: 5 most common `initial_max_streams_bidi` values for each scan

`initial_max_streams_bidi`			
scan 22-01-27		scan 22-10-13	
Value	# Endpoints	Value	# Endpoints
100	213478	100	361209
100000	49236	128	79628
256	10005	100000	58175
128	7998	256	15102
16	7619	16	1798

Table 5.7: 5 most common `initial_max_streams_uni` values for each scan

`initial_max_streams_uni`			
scan 22-01-27		scan 22-10-13	
Value	# Endpoints	Value	# Endpoints
103	143747	103	250265
10	68896	10	100129
100000	49236	3	94332
3	25438	100000	58175
100	2694	100	10093

The transport parameter `ack_delay_exponent` only has two main advertised values: 3, which is sent by 76,3% and 80,6% of servers in scan 22-01-27 and scan 22-10-13 respectively; and 10, which is used by 23,7% and 19,4% in scan 22-01-27 and scan 22-10-13. There exists one server in scan 22-01-27 and two servers in scan 22-10-13 which advertise the value of 8.

More than 99% of QUIC endpoints in scan 22-01-27 and more than 97% in scan 22-10-13, send the `max_ack_delay` parameter set to 25 ms. The second most seen value is 26 ms (used by less than 1% in scan 22-01-27 and 2,4% in scan 22-10-13). Other rarely seen values are: 20 ms, 41 ms and 251 ms.

In scan 22-01-27, 33,2% of endpoints do not allow active connection migration due to them advertising the `disable_active_migration` transport parameter. In scan 22-10-13, this percentage is reduced to 22%.

For all the scanned QUIC servers in both scans, the values set for the `active_conn_id_limit` range between 2 (the allowed minimum) and 8. 2 is the most advertised value for this parameter with 75,3% and 78,1% of servers having sent this limit in scan 22-01-27 and scan 22-10-13 respectively.

5.1.2 Internet Browsers

By capturing the QUIC Initial packets sent by the Google Chrome and Mozilla Firefox browsers, we can retrieve the transport parameters that they advertise. This provides an insight into what transport parameters QUIC clients use.

Chrome sends additional transport parameters which are Google-specific. These parameters are:

- `google_version`,
- `google_connection_options` and
- `initial_rtt`.

The Google-specific parameters are ignored in the rest of this work because we cannot test the impact of this and cannot set them using LSQUIC.

In Table 5.8, the values advertised by both browsers are presented.

Table 5.8: Chrome and Firefox's transport parameters.

Transport Parameter	Chrome	Firefox
`max_idle_timeout`	30000	30000
`max_udp_payload_size`	1472	65527[a]
`initial_max_data`	15728640	25165824
`initial_max_stream_data_bidi_local`	6291456	12582912
`initial_max_stream_data_bidi_remote`	6291456	1048576
`initial_max_stream_data_uni`	6291456	1048576
`initial_max_streams_bidi`	100	16
`initial_max_streams_uni`	103	16
`ack_delay_exponent`	3[a]	3[a]
`max_ack_delay`	25[a]	20
`disable_active_migration`	False[a]	True
`active_conn_id_limit`	2[a]	2[a]

[a] The transport parameter is not sent by the browser and the default value is assumed.

We see that both browsers advertise different values for potentially impactful transport parameters. We compare both of these client configurations later in Section 5.2.2.

5.2 Measurements

In this section, we present the results of the different measurements performed in our testing environment.

At first, we do not differentiate between client and server parameters. LSQUIC is configured to advertise the same transport parameter sets for the client and the server. We use four different transport parameter configurations along with LSQUIC's default parameters (presented in Table 2.1) for these measurements. The first and the second configurations are the two most common transport parameter sets (α and β) from

Section 5.1.1. The third configuration is the seventh most common parameter set seen in the scan performed on Oct 13, 2022. This configuration uses the following values:

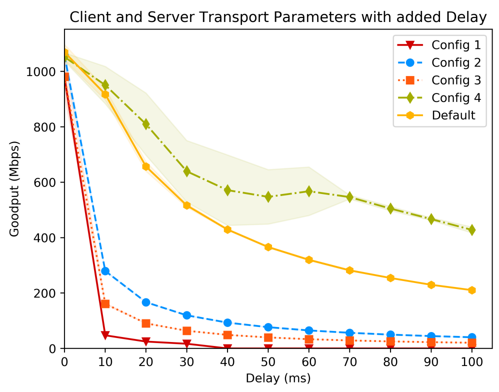
- `max_idle_timeout`: 180
- `max_udp_payload_size`: 65527
- `initial_max_data`: 68681728
- `initial_max_stream_data_bidi_local`: 524288
- `initial_max_stream_data_bidi_remote`: 524288
- `initial_max_stream_data_uni`: 524288
- `initial_max_streams_bidi`: 128
- `initial_max_streams_uni`: 3

The fourth and last configuration is the second least common parameter set seen in the scan from Oct 13, 2022. Its parameter values are:

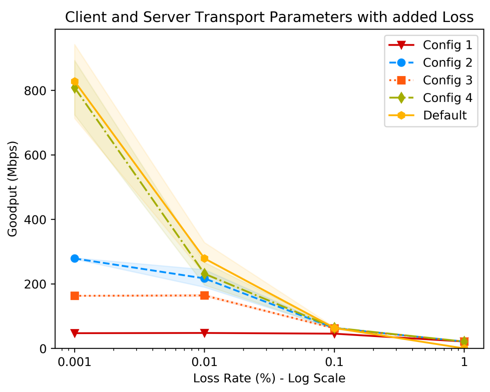
- `max_idle_timeout`: 120
- `max_udp_payload_size`: 1500
- `initial_max_data`: 4294967295
- `initial_max_stream_data_bidi_local`: 16777216
- `initial_max_stream_data_bidi_remote`: 16777216
- `initial_max_stream_data_uni`: 16777216
- `initial_max_streams_bidi`: 1024
- `initial_max_streams_uni`: 1024

We choose these last two parameter sets due to them using not commonly observed values for the flow-control-related parameters. During the measurements, we discover that LSQUIC does not support the `initial_max_data` value of the fourth configuration. This occurs because LSQUIC uses the 'unsigned' type in C to store the value of the transport parameter. We try setting the value to 34359738368, which is the retrieved value from the scan, but this value is larger than the maximum 32 bit unsigned integer value (4294967295). Therefore we decide to use the largest possible value instead.

The goodput test is performed, and we compare the average result of 10 repetitions downloading a 500 MB file under different added packet delay and loss values. The comparison under packet delay can be seen in Figure 5.1a. and the results when packet loss is present can be seen in Figure 5.1b.



(a) Goodput under delay



(b) Goodput under loss (note the x-axis does not start with 0 since it is a logarithmic scale).

Figure 5.1: Goodput achieved using the same transport parameter configurations for client and server under packet delay and loss. (The colored area surrounding each curve represents the standard deviation.)

In a perfect network environment with no loss or delay, the achieved throughput among all the configurations is comparable. The highest goodput, on average, is achieved by LSQUIC's defaults with 1069.48 Mbit/s (± 27.95 Mbit/s). The lowest goodput is seen for configuration 1 with 977.79 Mbit/s (± 22.96 Mbit/s). Configuration 3 lies within one standard deviation from configuration 1, while configurations 2 and 4 are also within the standard deviation of LSQUIC's defaults.

When network delay is introduced, configurations 1, 2 and 3 experience a significant decrease in the goodput of the connection when compared to configuration 4 and LSQUIC's defaults. For example, with only 10 ms of added delay, configuration 1 sees a goodput decrease of more than 95%; configuration 2 decreases over 70%; and configuration 3 experiences a reduction of more than 80%. Increasing delay leads configurations 1, 2 and 3 to slowly and continuously decrease the goodput of the connection (note that with a delay of 40 ms and beyond, the download with configuration 1 timeouts). In contrast, configuration 4 and LSQUIC's default do not suffer such an impact on the throughput with the initial addition of network delay. Configuration 4 performs similarly to LSQUIC's defaults (lying within the standard deviation) with 10 ms of added delay. As the delay increases, we see that configuration 4 has a higher goodput average across the board, but this comes with a relatively high standard deviation (up to added delay values of 70 ms). The standard deviation of configuration 4 reaches a standard deviation of ± 127.18 Mbit/s with 40 ms delay, which is relatively high when compared

to the other configurations. It is especially noticeable considering that for the other configurations, the standard deviation is not even visible or barely visible at most in Figure 5.1a.
The packet loss measurement, depicted in Figure 5.1b, has a fixed delay of 10 ms to amplify the effects of packet loss and make it more noticeable. At a low probability of 0.001% packet loss, results similar to those in Figure 5.1a with 10 ms delay are observed due to the low chance of packets being dropped. However, a noticeable difference can be seen in the standard deviation of the measurements for LSQUIC's default configuration. Increasing the probability of packets being lost provoke a decreased throughput on configuration 4 and the LSQUIC default configuration's connections. At 0.01% packet loss, there is no difference significantly larger than the standard deviation between configuration 2, configuration 4 and LSQUIC's defaults. For even higher loss values, the goodput of all configurations decreases to relatively low and similar values, with the default configuration even timing out at 1% loss[1]. This behavior is expected due to the underlying congestion control algorithm; CUBIC. CUBIC makes the assumption, that packet loss signalizes congestion in the network [24], and thus the sending rates should get decreased in our case.

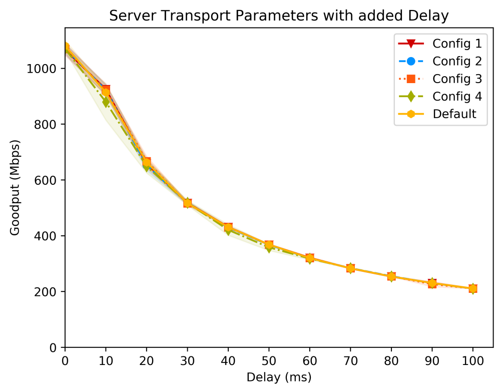
The results seen in Section 5.2 raise our interest in explaining the subpar performance of configurations 1, 2 and 3. Furthermore, it opens the question of why configuration 4 outperforms LSQUIC's default configuration when there is added delay in the network as seen in Figure 5.1a. To answer these questions, we proceed to differentiate between server and client parameters and perform measurements for them individually in Section 5.2.1 and Section 5.2.2 respectively. It is also adequate to make this differentiation since in actual QUIC connections, clients and servers tend to advertise different transport parameter sets as shown in Section 5.1.

5.2.1 Server Parameters

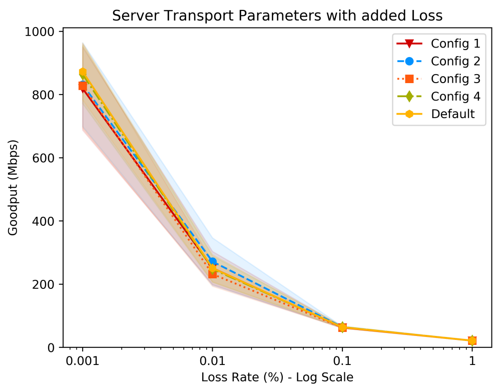
To measure the impact of the server's advertised transport parameters, we decide to fix the client's transport parameters to LSQUIC's defaults and try out different parameters for the server. We begin by using the same five configurations as in the previous measurements, but only set the configuration's values on the servers. Again we perform

[1] When a repetition timeouts in a measurement, the test is marked as failed and the remaining repetitions get aborted. The measurement with LSQUIC's default transport parameter configuration with 1% packet loss timeouts in the 8th repetition, until that point the default configuration was averaging a goodput of around 21 Mbit/s, the same as all the other configurations.

10 repetitions downloading a 500 MB file under added packet delay and packet loss (see Section 5.2.1).



(a) Goodput under delay



(b) Goodput under loss (note the x-axis does not start with 0 since it is a logarithmic scale).

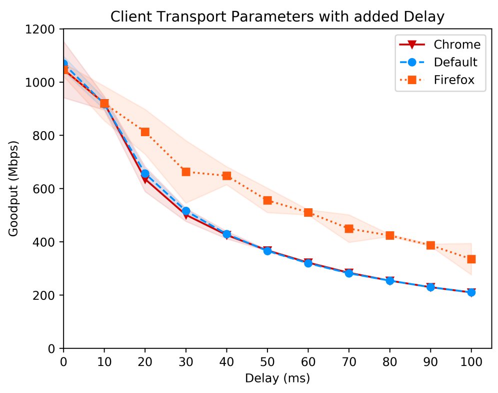
Figure 5.2: Goodput achieved with different transport parameter combinations for the server and LSQUIC's default parameters for the client under packet delay and loss. (The colored area surrounding each curve represents the standard deviation.)

When comparing the parameter configurations in a network with packet delay, it is seen in Figure 5.2a, that the goodput of the different configurations is almost identical. The curves of the different configurations are the same as the curve of the LSQUIC default configuration in Figure 5.1a.
Similar behavior is observed in Figure 5.2b when packet loss is added to the connection. A likelihood of 0.001% packet loss produces a minimal difference between the goodput achieved with the different transport parameter combinations. This minimal difference is within the standard deviation, which is shown as the colored area surrounding the curves. Higher loss values result in a reduced standard deviation and with 1% loss, all configurations end up converging to the same 21 Mbit/s goodput mark.

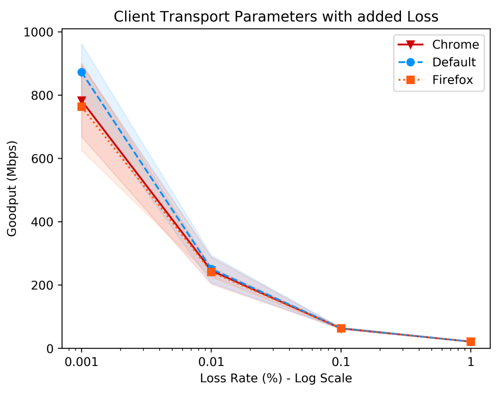
The findings observed in Section 5.2.1 reveal a negligible difference in the connection's goodput between LSQUIC's standard parameters and configurations 1, 2, 3 and 4 on the server. From these findings, we can deduce that the differences in the goodput, seen in Section 5.2, between the measured configurations must be caused by the transport parameters used by the QUIC client and not the parameters advertised by the server. The client's parameters impact is analyzed in more detail in Section 5.2.2.

5.2.2 Client Parameters

In this section, the impact of the client's advertised transport parameters is presented. QUIC clients' transport parameter values tend to differ from those advertised by QUIC servers. This can be observed when comparing the values advertised by actual clients (Chrome and Firefox browsers in Section 5.1.2) to the values advertised by the servers in the Internet-wide scans from Section 5.1.1. Therefore, the transport parameters configurations used in the previous sections (configurations 1, 2, 3 and 4 defined at the start of Section 5.2) are not used for the coming measurements. Instead, the transport parameter values advertised by the Chrome and Firefox browsers are used. The LSQUIC's default client transport parameter set is also included in the measurements for comparison purposes. To evaluate the effect of QUIC client's transport parameters, the server's transport parameters are fixed to use LSQUIC's default server values. Once again we perform a 500 MB file download introducing packet delay and packet loss into the network. 10 repetitions are done for each data point. We plot the results of these measurements in Section 5.2.2.



(a) Goodput under delay



(b) Goodput under loss (note the x-axis does not start with 0 since it is a logarithmic scale).

Figure 5.3: Goodput achieved with different transport parameter combinations for the client using LSQUIC's default parameters for the server under packet delay and loss. (The colored area surrounding each curve represents the standard deviation.)

In Figure 5.3a, we can observe that for 0 and 10 ms of delay, the difference between Firefox, Chrome and the default LSQUIC configuration is negligible. Chrome's configuration achieves a very similar goodput to that of LSQUIC's default client for higher delay values. This is expected since both configurations share many values for potentially performance-relevant parameters. The `initial_max_data` and `initial_max_stream_data_bidi_local` transport parameters have the same values in both the Chrome configuration and the LSQUIC's default configuration. In contrast, the configuration used

by the browser has a better performance than the other two configurations when we introduce delay values of 20 ms and beyond. For those delay values, the Firefox configuration averages around 170 Mbit/s more than the other configurations. The Firefox configuration also has a higher standard deviation than the other configurations (depicted as the area surrounding the curves). Interestingly, the Firefox browser advertises significantly higher values than the other clients analyzed. For the texttttinitial_max_ data and `initial_max_stream_data_bidi_local`, Firefox uses values that are 1.6 and 2 times larger respectively.
Introducing packet loss into the connection does not reflect a significant difference between the three measured client configurations as seen in Figure 5.3b. We observe that the higher the loss rate, the less the difference between the configurations; this finding is consistent with all the previous measurements with added packet delay (see Figure 5.1b and Figure 5.2b).

Building on the results showing the improved performance of the Firefox browser's configuration in a network with increased delay values, we now turn our attention to analyzing the potential client parameters that contribute to this increase in the goodput.
The connections we measure are file downloads initiated by the client, therefore the QUIC streams are initiated by the client as well. This lets us discard parameters such as `initial_max_stream_data_bidi_remote`, and `initial_max_streams_bidi` since they only affect peer-initiated streams. We can also disregard the `initial_max_stream_data_uni` and the `initial_max_streams_uni` transport parameters since in our case, the data in our download is transferred in client bidirectional streams (this is checked by inspecting the traffic of several measurements) and not uni-directional streams. This leaves the client's `initial_max_data` and `initial_max_stream_data_bidi_local` parameters in question. These client transport parameters can be relevant to the performance of our connections because the `initial_max_data` parameter limits the maximum amount of data to be transmitted in an entire connection; and the `initial_max_stream_data_bidi_local` parameter advertises the flow control limit for locally initiated bidirectional streams. All mentioned transport parameters are previously described in Section 2.2.1. We now isolate the client's `initial_max_data` and `initial_max_stream_data_bidi_local` transport parameters and investigate their impact on the connection's performance.

To measure the impact of the `initial_max_data` parameter, we fix all other client and server transport parameters to the default values provided by LSQUIC. We perform 20 repetitions of the goodput test for each of the values we test and we set 30 ms delay in the network. The added delay simulates the conditions of the previous measurements (see Figure 5.1a and Figure 5.3a) where we observed a performance difference. We use a combination of values advertised by the analyzed QUIC clients and arbitrary values (some of which were advertised by servers) to create a range of values to allow comparison. For example, we notice that LSQUIC's `initial_max_data` value for the client is exactly 10 times the value advertised by the server. Therefore we also consider the values in between those two advertised values.

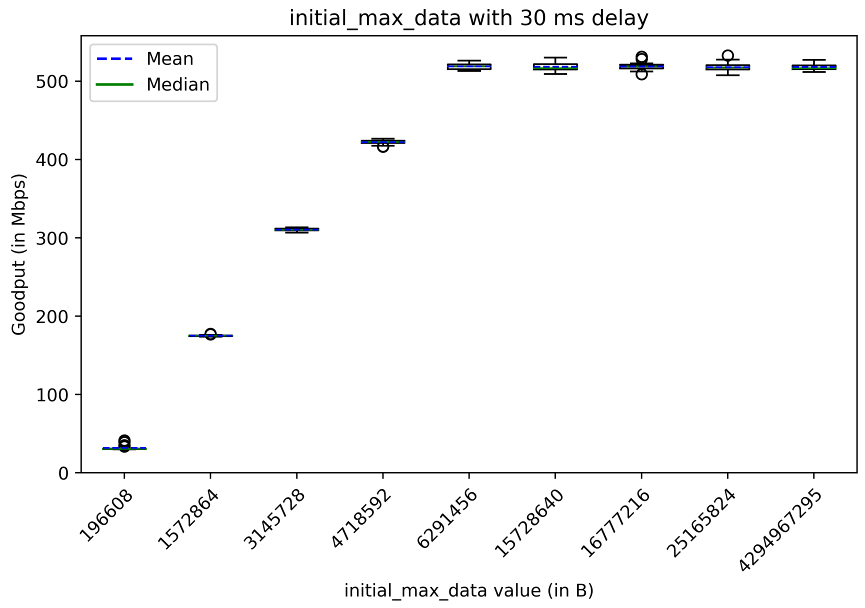
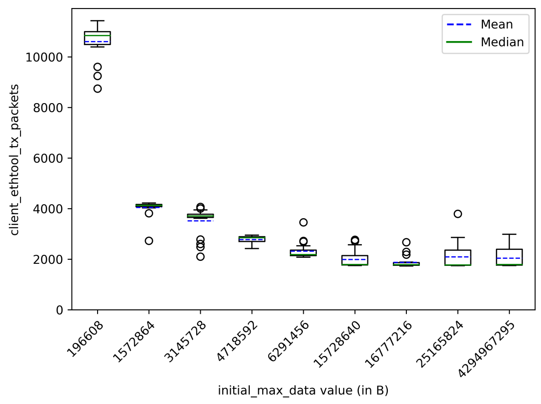


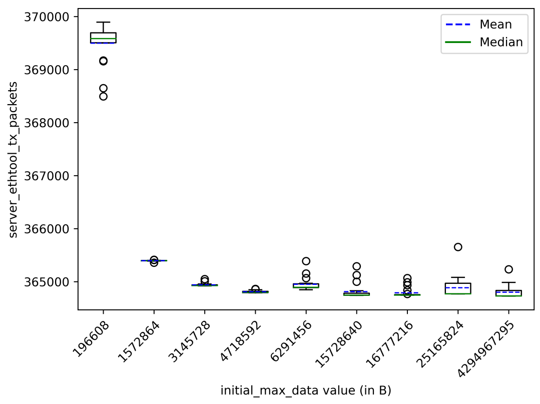
Figure 5.4: Goodput with different `initial_max_data` values with 30 ms delay. All other client and server transport parameters are fixed.

In Figure 5.4, we confirm that the `initial_max_data` transport parameter value advertised by the client affects the performance of the connection. We observe, that higher `initial_max_data` values produce a higher goodput in the connection. Although it should be noted, that for values greater than 6 291 456 B (6.29 MB), there is no longer an increase in the goodput. Even using the highest possible value for the parameter does not increase the goodput significantly. The goodput obtained with the `initial_max_data` values of the Chrome browser, the LSQUIC's default client, the value of set α from Section 5.1.1, the Firefox browser; and the maximum possible value is very similar. Values lower than 6.29 MB present a decrease in the goodput of the connection. Note that this value is not the exact threshold from which the goodput starts dropping considerably. We see that the `initial_max_data` advertised by set α from Section 5.1.1 and the value advertised by LSQUIC's default server have a very significant effect on the connection; lowering the goodput by more than half in the case of the LSQUIC's server and by reducing the throughput by more than 90% in the case of set β from Section 5.1.1.

We look into the network interface's statistics generated with Ethtool [23] to find out if there is a difference in the packets sent and received by the client and server when different `initial_max_data` values are advertised by the client. The number of server-sent packets is assumed to be the same as the number of received packets by the client and vice-versa. This assumption is done since in some isolated cases, there may exist a difference of at most one single packet. Due to this assumption, we only plot and analyze the number of packets sent by the client and the number sent by the server in Section 5.2.2.



(a) Client's transmitted packets.



(b) Server's transmitted packets (note that the y-axis does not start with 0 to help contrast the observed results)

Figure 5.5: Comparison of the number of packets sent by the client and the server interface for different values of the client's `initial_max_data` with a fixed delay of 30 ms.

In Figure 5.5a we observe that the values that achieve lower goodput in Figure 5.4 tend to transmit more packages. Advertising 196 608 B, results in the client sending substantially more packets than other values; it can send up to 5 times as many packets. The client can send more than 10 000 packets when advertising 196 608 B as the `initial_max_data`, but it send roughlt between 2000 and 4000 packets. Less noticeably, it can be observed that the higher the `initial_max_data`, the lower the number of packets sent by the client. Nonetheless, this affirmation is only applicable up to a certain point. Similar to Figure 5.4, the threshold for the point from which further increasing the transport parameter's value does not affect performance could be defined again as 6.29 MB.

Due to the relatively high number of packets sent by the server during a download, the relative change in packets transmitted by the server is lower when compared to the numbers seen with the client. Nevertheless, a significantly higher number of packets are sent, in this case by the server, when the used value for the `initial_max_data` is equal to 196 608 B. As seen in Figure 5.5b, the server can send upwards of 5000 packets when advertising the previously mentioned `initial_max_data` value. It could be argued that a correlation between the size of the advertised parameter's value and the number of server-sent packets exists, although again due to the smaller relative difference, the affirmation is weaker in this case.

In the next measurement, we evaluate the impact of the `initial_max_stream_data_bidi_local` transport parameter. For this measurement, we fix all other LSQUIC transport parameters to their default values and perform 20 repetitions of the goodput test with 30 ms of added delay for each different parameter value. The values selected for the transport parameter consist of a similar range of values as the one used for Figure 5.4 and Section 5.2.2.

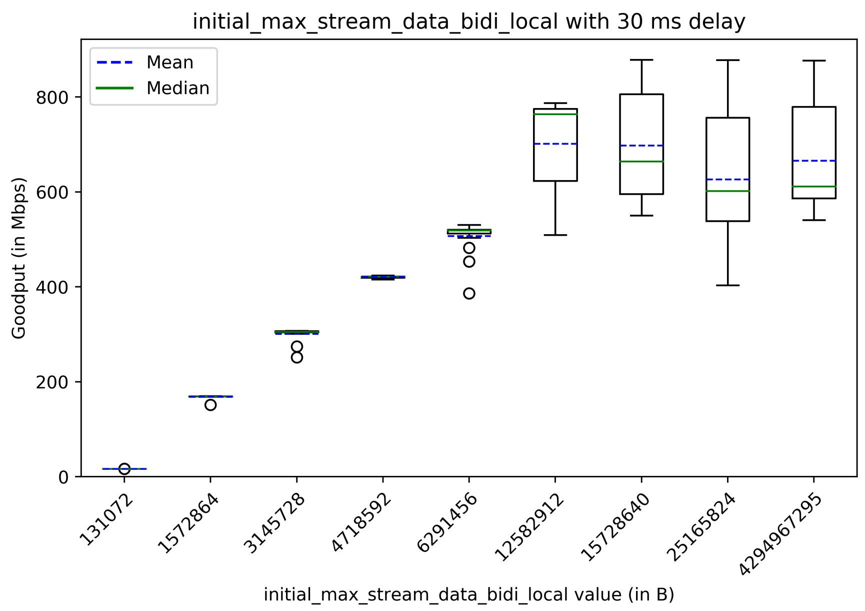
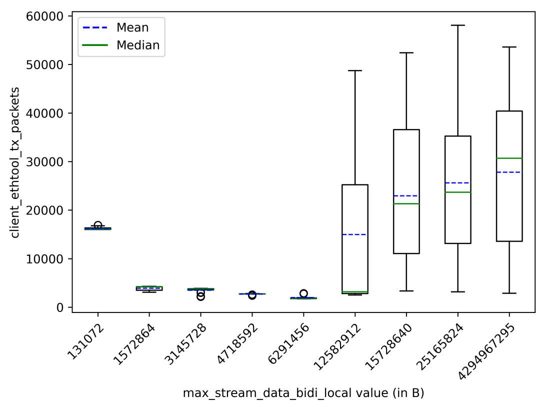


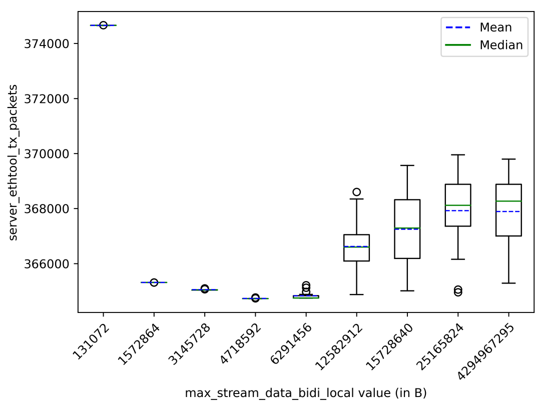
Figure 5.6: Goodput with different `initial_max_stream_data_bidi_local` values with 30 ms delay. All other client and server transport parameters are fixed.

In Figure 5.6 it can be seen, that the client's advertised `initial_max_stream_data_bidi_local` value affects the connection's goodput in the presence of packet delay. We can see, that using values lower than 12 582 912 B (12.58 MB) decrease the the goodput significantly. It is observed that for greater parameter values, the goodput's average does not vary much, but the observed goodput tends to vary more for each repetition. The subpar performance offered by using the `initial_max_stream_data_` `bidi_local` values used by set α and β from Section 5.1.1 further explain the results seen in Figure 5.1a. The difference between setting the parameter to 6.29 MB and 12.58 MB results in a goodput difference of almost 200 Mbit/s. This finding explains the difference seen between the goodput achieved by the Chrome and Firefox browser configurations in Figure 5.3a.

To further understand this finding, we analyze the interface statistics that are generated during the measurements in Section 5.2.2. The number of server-sent packets is assumed to be the same as the number of received packets by the client and vice-versa. This assumption is done since at most, a difference of one single packet between the two counters may exist in some cases.



(a) Client's transmitted packets.



(b) Server's transmitted packets (note that the y-axis does not start with 0 to help contrast the observed results)

Figure 5.7: Comparison of the number of packets sent by the client and the server interface for different values of the client's `initial_max_stream_data_bidi_local` with a fixed delay of 30 ms.

The client's number of sent packets varies greatly depending on the value of the client's advertised `initial_max_stream_data_bidi_local` transport parameter as seen in Figure 5.7a. In contrast to the `initial_max_data` transport parameter, `initial_max_stream_data_bidi_local` values that produce a higher goodput, tend to cause the client to send more packets than values that achieve a lower goodput (i.e., values lower than 12.58 MB). However this does not apply for all `initial_max_stream_data_bidi_local` values; when setting the parameter to 131.72 kB, we observe a low goodput of 16.81 Mbit/s but roughly the same number of packets sent as when setting the parameter to 12.58 MB (which causes a goodput of 701.33 Mbit/s). The number of packets sent by the server (seen in Figure 5.7b) resembles the number of packets sent by the client depending on the value of `initial_max_stream_data_bidi_local`. Parameter values higher than 6.26 MB make the connection achieve a higher goodput but also cause the server to send more packets. Again the relatively small value of 131.72 kB is an exception to this.

Chapter 6

Conclusion

Here we summarize and discuss the effect of QUIC's transport parameters on the protocol's performance. Furthermore, we present an outlook on future research possibilities.

6.1 Summary

The main questions for our research were as follows:

- Which transport parameters are used by QUIC deployments on the Internet?
- Do these parameters have an effect in the first place?
- If they do, how do they affect the performance?

At first, we differentiate between transport parameters advertised by clients and servers. We recognize 148 different server transport parameter sets in the latest Internet-wide scan and we find out that for some parameters many different values are being advertised while for other parameters, a large majority of QUIC servers advertised the default values defined for the protocol. This finding is consistent with the findings from Zirngibl et al. [4]. We show that the Firefox and Chrome browsers also advertise different values for the transport parameters. These different values found for client and server transport parameters are used to perform our measurements.

The measurements performed allows us to show the effect of transport parameters in different network environments. In perfect network conditions, we notice a negligible effect of client and server transport parameters on the performance. We identify that for high packet loss rates, it is probably the congestion control algorithm in use (CUBIC) that limits the throughput in the connection. Furthermore, when adding packet delay into the network we observe different behavior. On the one hand, the transport parameters advertised by the server do not have a significant effect on the goodput. On the other hand, the parameters advertised by the client can significantly impact the performance of the connection.

It is also shown, that the Firefox browser's transport parameter configuration achieves higher goodput than the Chrome configuration in connections that experience high packet delay. We distinguish which specific client transport parameters cause the impact on the performance. We try out different values for these parameters and also see that the number of packets sent varies when using different transport parameter values.

The results of our work and the seen effects of certain transport parameters are obtained by using the LSQUIC implementation. Therefore, it must be noted that using another QUIC implementation may lead to different results.

6.2 Future Work

The effect of transport parameters on QUIC's performance still has a lot of potential for future studies, especially considering that protocol extensions can add more transport parameters in the future.

We believe that the findings shown in this work can bootstrap further work on the topic. As the QUIC protocol continues to evolve and be adopted more widely, it might be of interest to find transport parameter values that can optimize connections given different network conditions. Areas that this work does not cover but could be interesting to research further are:

- Analyzing the effect of transport parameters on other QUIC implementations.
- Exploring effect of other parameters such as `max_ack_delay`, which LSQUIC does not allow configuring. As well as implementation-specific transport parameters like Google-specific parameters advertised by the Chrome browser.
- Comparing the advertised parameters from other Internet browsers.
- Performing similar measurements with BBR as the congestion control algorithm.

Chapter A

Appendix

A.1 List of acronyms

TCP	Transmission Control Protocol
UDP	User Datagram Protocol
ISO	International Organization for Standardization
OSI	Open Systems Interconnection
IETF	Internet Engineering Task Force
TLS	Transport Layer Security
RTT	round-trip-time
HOL	Head-of-Line
IANA	Internet Assigned Numbers Authority
LSQUIC	LiteSpeed QUIC
POS	Plain Orchestrating Service
MTU	Maximum Transmission Unit
HTTP	Hypertext Transfer Protocol
ACK	acknowledgement, a message, sent by the receiver to the sender, confirming the received data sent previously and that it is ready for the next.
TBF	Token Bucket Filter, is a classful queueing discipline available for traffic control with the tc command [25].
IFB	Intermediate Functional Block, a pseudo network interface, regularly used to redirect other interfaces to it and apply a single stack of queueing disciplines, classes and filters.

Bibliography

[1] I. O. for Standardization, “"ISO/IEC 7498-1:1994 Information technology" - Open Systems Interconnection - Basic Reference Model”, International Organization for Standardization, Tech. Rep., 1994.

[2] J. Roskind, “QUIC: Design Document and Specification Rationale”, Tech. Rep., Apr. 2012. [Online]. Available: `https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit?usp=sharing` (visited on 01/21/2023).

[3] J. Iyengar and M. Thomson, “RFC 9000: QUIC: A UDP-Based Multiplexed and Secure Transport”, Tech. Rep., May 2021. [Online]. Available: `https://www.rfc-editor.org/rfc/rfc9000.txt`.

[4] J. Zirngibl, P. Buschmann, P. Sattler, B. Jaeger, J. Aulbach, and G. Carle, “It’s over 9000: Analyzing Early QUIC Deployments with the Standardization on the Horizon”, in *Proceedings of the 21st ACM Internet Measurement Conference*, ser. IMC ’21, Virtual Event: ACM, 2021, 261–275, ISBN: 9781450391290. DOI: `10.1145/3487552.3487826`.

[5] E. Volodina and E. P. Rathgeb, “Impact of ACK Scaling Policies on QUIC Performance”, in *2021 IEEE 46th Conference on Local Computer Networks (LCN)*, 2021, pp. 41–48. DOI: `10.1109/LCN52139.2021.9524947`.

[6] M. Piraux, Q. De Coninck, and O. Bonaventure, “Observing the Evolution of QUIC Implementations”, in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, ser. EPIQ’18, Heraklion, Greece: Association for Computing Machinery, 2018, 8–14, ISBN: 9781450360821. DOI: `10.1145/3284850.3284852`.

[7] W. Eddy, “RFC 9293: Transmission Control Protocol (TCP)”, Tech. Rep., Aug. 2022. [Online]. Available: `https://www.rfc-editor.org/rfc/rfc9293.txt`.

[8] J. Postel, “RFC 768: User Datagram Protocol”, Tech. Rep., Aug. 1980. [Online]. Available: `https://www.rfc-editor.org/rfc/rfc768.txt`.

[9] M. Thomson and S. Turner, "RFC 9001: Using TLS to Secure QUIC", Tech. Rep., May 2021. [Online]. Available: `https://www.rfc-editor.org/rfc/rfc9001.txt`.

[10] "IANA: QUIC Transport Parameters", IANA, Tech. Rep., Feb. 2021. [Online]. Available: `https://www.iana.org/assignments/quic/quic.xhtml#quic-transport` (visited on 01/22/2023).

[11] M. Kühlewind and B. Trammell, "Applicability of the QUIC Transport Protocol", Internet Engineering Task Force, Internet-Draft, Sep. 2022, Work in Progress. [Online]. Available: `https://quicwg.org/ops-drafts/draft-ietf-quic-applicability.html` (visited on 01/30/2023).

[12] T. Pauly, E. Kinnear, and D. Schinazi, "RFC 9221: An Unreliable Datagram Extension to QUIC", Tech. Rep., Mar. 2022. [Online]. Available: `https://www.rfc-editor.org/rfc/rfc9221.txt`.

[13] M. Thomson, "RFC 9287: Greasing the QUIC Bit", Tech. Rep., Aug. 2022. [Online]. Available: `https://www.rfc-editor.org/rfc/rfc9287.txt`.

[14] D. Schinazi and E. Rescorla, "Internet-Draft: Compatible Version Negotiation for QUIC", Tech. Rep., Dec. 2022. [Online]. Available: `https://www.ietf.org/archive/id/draft-ietf-quic-version-negotiation-14.html#name-version-information` (visited on 01/23/2023).

[15] *LSQUIC*, LiteSpeed. [Online]. Available: `https://github.com/litespeedtech/lsquic` (visited on 01/22/2023).

[16] K. Wolsing, J. Rüth, K. Wehrle, and O. Hohlfeld, "A Performance Perspective on Web Optimized Protocol Stacks: TCP+TLS+HTTP/2 vs. QUIC", in *Proceedings of the Applied Networking Research Workshop*, ser. ANRW '19, Montreal, Quebec, Canada: Association for Computing Machinery, 2019, 1–7, ISBN: 9781450368483. DOI: `10.1145/3340301.3341123`.

[17] J. Iyengar and I. Swett, "QUIC Acknowledgement Frequency", Internet Engineering Task Force, Internet-Draft draft-ietf-quic-ack-frequency-02, Jul. 2022, Work in Progress, 13 pp. [Online]. Available: `https://datatracker.ietf.org/doc/draft-ietf-quic-ack-frequency/02/` (visited on 01/24/2023).

[18] M. Kempf, "Analysis of Performance Limitations in QUIC Implementations", Dec. 2022.

[19] *QUIC Interop Test Runner*, Chair of Network Architectures and Services. [Online]. Available: `https://gitlab.lrz.de/acn/quic-project/quic-interop-runner-dev`.

[20] *Wireshark: Network Analyzer.* [Online]. Available: `https://www.wireshark.org/`.

[21] S. Gallenmüller, D. Scholz, H. Stubbe, and G. Carle, “The Pos Framework: A Methodology and Toolchain for Reproducible Network Experiments”, in *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT ’21, Virtual Event, Germany: Association for Computing Machinery, 2021, 259–266, ISBN: 9781450390989. DOI: `10.1145/3485983.3494841`. [Online]. Available: `https://doi.org/10.1145/3485983.3494841`.

[22] M. Seemann and J. Iyengar, “Automating QUIC Interoperability Testing”, in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, ser. EPIQ ’20, Virtual Event, USA: Association for Computing Machinery, 2020, 8–13, ISBN: 9781450380478. DOI: `10.1145/3405796.3405826`.

[23] *ethtool man page.* [Online]. Available: `https://man7.org/linux/man-pages/man8/ethtool.8.html`.

[24] S. Ha, I. Rhee, and L. Xu, “CUBIC: A New TCP-Friendly High-Speed TCP Variant”, *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, 64–74, Jul. 2008, ISSN: 0163-5980. DOI: `10.1145/1400097.1400105`. [Online]. Available: `https://doi.org/10.1145/1400097.1400105`.

[25] *tbf man page.* [Online]. Available: `https://man7.org/linux/man-pages/man8/tbf.8.html`.