UCLouvain

École polytechnique de Louvain

Measuring Internet Performances with QUIC

Author: **Félix Gaudin**
Supervisor: **Olivier Bonaventure**
Readers: **François Michel, Cristel Pelsser**
Academic year 2022–2023
Master [120] in Computer Science

Abstract

For many years internet utilization has increased. And people are having more and more bandwidth capacity, but they still have a lot of latency. There exist solutions to handle this latency but they are not enough applied. At the same time, internet protocols are evolving and new transport protocols are being created. This is the case for the QUIC protocol.

So in this work, we propose tools to point out the latency issue. The first one is a set of statistics that can be derived from a QUIC connection. The second tool measures latency for different scenarios where for example we take a fixed sending rate beyond user capacity. And the third tool is a website that measures latency in working conditions and so knows the real impact of the latency increase for people. With those tools, we were able to quantify the impact of the augmentation of latency inside the internet and know the cause of this.

Acknowledgements

I want to sincerely thank my promoter, Professor Bonaventure, for letting me do this topic, for challenging me and for his precious feedback. I also want to thank Maxime Piraux and François Michel who helped me and gave me feedback all along this work. And in general, I want to thank all the INL team who helped me.

Then I want to thank Cristoph, Will, Simon, and the Network Quality team for their help in implementing the RPM methodology inside LibreSpeed, but also for all the discussions we had about improving the latency.

After I want to thank all the people who made a test or who shared the link to the test. I didn't believe there would be so many results.

Finally, I want to thank my friends and my family for their mental support all along this work. I particularly want to thank Adé for supporting me and for her advice on writing emails and this work. Last but not least, I want to thank Chipie, my cat, for accompanying me with her purrs during the nights of work.

Contents

Chapter 1

Introduction

Since its beginning, the internet has evolved a lot. It became an important tool for the daily life. And so people want to always have access to their resources as fast as possible. For decades, the user capacity increased to having now gigabit connections [22].

But on the other side, the capacity is no longer a bottleneck but another one comes. The latency is becoming a bottleneck because of network queueing. People are sometimes thinking those latencies are normal but there exist solutions to avoid such problems [40, 25, 13]. So we need to have tools to detect where the solutions can be applied.

They are already a lot of tools to diagnose an internet connection but only a few of them are looking at buffering delay [44, 39].

At the same time, internet protocols are evolving and new protocols are emerging to improve other aspects, like security or privacy. This is the case for the new QUIC protocol [27]. But with this new protocol, there is no tool to measure its performance.

In this work, we focus on detecting network queuing, and so know where the argumentation of the latency is coming from, and this using the QUIC protocol. We first introduce the state of the art for network protocols and the different metrics used for diagnostics. Then we define a bunch of statistics that we can take from a QUIC connection. After that, we define two tools. The first is a Command Line Interface (CLI) tool that can perform fixed-rate connections. The second is a Graphic User Interface (GUI) tool that measures latency under working conditions using the RPM methodology [39]. Then after defining those tools, we present a validation process for them. Finally, we analyze the results from a live experiment.

Chapter 2

Background

In this chapter, we describe the background knowledge that is useful to understand our work. We start by introducing some internet protocols. Because our work is on measuring internet performance, it's very important to have an understanding of the lower levels. Then we describe how the performances are made. From early testing in the ARPANET to actual websites. And finally, we describe the new metrics that we will use in our work.

2.1 Internet Protocols

In this section, we describe several protocols on top of the Internet Protocol (IP).

As mentioned in the standardization document [3], The Internet Protocol is designed for use in interconnected systems of packet-switched computer communication networks. As illustrated in table 2.1, the IP is on the third layer of the OSI model [17].

The IP has two versions: IPv4 and IPv6. The most useful fields in both headers are the `source address` and the `destination address` because those fields are used to know where a packet should go and where the answer must go.

2.1.1 User Datagram Protocol

The User Datagram Protocol (or UDP) was introduced in 1980 [2]. This protocol allows users to exchange datagrams between two computers connected via IP. As illustrated in table 2.1 The UDP protocol is used on top of the IP protocol.

A datagram is a type of organization for a network packet that contains the network layer address of the destination host, its network layer address and the information to be sent [10]. In the case of UDP, the network layer address is the port on the different hosts. Those fields are stored in a header.

UDP provides an unreliable transfer. This means that datagrams may be lost during a connection and the data inside those may be lost. This can be a benefit when you don't need all data, for example when streaming a video, if there is a frame missing you may not notice it.

But also UDP is connectionless. And since it doesn't need to establish a connection before sending data, thus this can reduce the time taken.

4	UDP/TCP
3	IP
2	Datalink
1	Physical

Table 2.1: TCP/UDP in OSI layers

2.1.2 Transmission Control Protocol

The Transmission Control Protocol or TCP is today the most commonly used transport protocol [15, 4, 30]. As UDP the protocol is on top of the IP layer, this is illustrated in the table 2.1.

The TCP header, like the UDP header, includes source and destination ports. The difference is that TCP introduces reliable transfer. To do so, TCP introduces sequence numbers for data and acknowledgment. Before it's connection-based, TCP must establish a handshake. This cost one Round-Trip Time. But after that, the client and the server can handle losses.

Congestion Control Algorithms

However, TCP cannot know what is the internet capacity of a host. So it uses a congestion control algorithm to shape the traffic to try to guess this capacity. They are several types of algorithms: loss-based, delay-based and hybrid [45].

loss-based Those algorithms focus on loss events. So they are reacting when buffers are full and packets are dropped. We can call those algorithms as *reactive*. So with those algorithms, we expect to use a huge amount of the available bandwidth because they have to saturate it to get losses. And so we expect them to increase latency because the network queue will be filled.
Examples: New Reno [21], Cubic [41]

delay-based Those algorithms are focussing on the delay. They are preventing from having congestion by looking at the RTT. We can call those algorithms as *proactive*. Unlike loss-based we don't expect the delay-based algorithms to use a

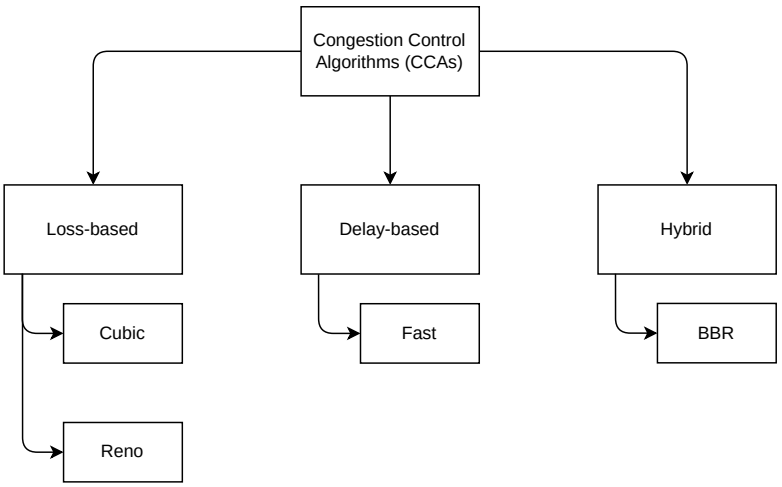


Figure 2.1: Diffrenct CCA families

huge amount of the bandwidth. But we expect them to have a small and stable latency.
Examples: Fast [28], Vegas [12]

Hybrid As the name suggests, those are both loss and delay based. For those we expect them to use a significant amount of the bandwidth but also avoiding to have a huge latency increase. Examples: BBR [14]

2.1.3 HyperText Transfer Protocol

The HyperText Transfer Protocol (HTTP) is a protocol that allows users to fetch resources located in another host. Those resources can go from simple HTML documents to large images/videos/etc. In this section, we describe the evolution of this protocol. We based on the mdn documentation[1], the CNP3 ebook [10] and the different standardization document (RFC).

The protocol uses TCP as an underlying layer to have connection reliability. The client has to send a request to a server to get the page content. They are several types of requests called *HTTP Methods* [19], for example making a `GET` request, you will get information from the server.

HTTP/1.0

The `HTTP/1.0` protocol is a *text* protocol. This means that the queries are written in ASCII and so are human-writable/readable. The main type of resource loaded

[1] `https://developer.mozilla.org/en-US/docs/Web/HTTP/`

by this protocol is HTML pages. But you can have other resources on a page such as images. And the protocol can fetch them doing another request on another connection.

However, `HTTP/1.0` has never been standardized. There exists a document about best practices [38] but, this is not a standardization.

HTTP/1.1

To finally have a first standardized document, `HTTP/1.1` was created [20]. This version brings some innovation.

When fetching multiple resources on the same server, having to establish each time a new connection will add non-negligible time to make the connection establishment. So to avoid that, `HTTP/1.1` is reusing TCP connections. Another time saving the *pipelining* principle. The idea is to load the resources one after the other.

Another interesting thing with `HTTP/1.1` is the ability to cache information. The server has to put a header indicating that the resource is cacheable and for how long. This improvement allows browsers to store information to avoid re-fetching it to the server.

Finally, a last improvement is the ability to compress information before sending it. This reduces the number of packets needed to process a website.

For years, `HTTP/1.1` has evolved with small improvements. For example, the addition of a security layer (TLS) or with the WebSocket/push from the server. In the 2010s, Google developed a protocol called `SPDY`. This protocol will evolve into `HTTP/2`.

HTTP/2

This version brings some major changes. First, the protocol uses binary messages, so not human-readable. But this increases performances. And also brings multiplexing. This allows clients to make parallel requests on the same connection, and so to reduce the global time used. The different modes are illustrated in figure 2.2, where `HTTP/1.0` uses no pipeline principle, `HTTP/1.1` uses pipelining and `HTTP/2` uses multiplexing.

2.1.4 Quick UDP Internet Connections

In 2012, Google proposed a new protocol called Quick UDP Internet Connections (QUIC) [42]. The idea is to use a UDP stack and add the transport protocol on top of it.

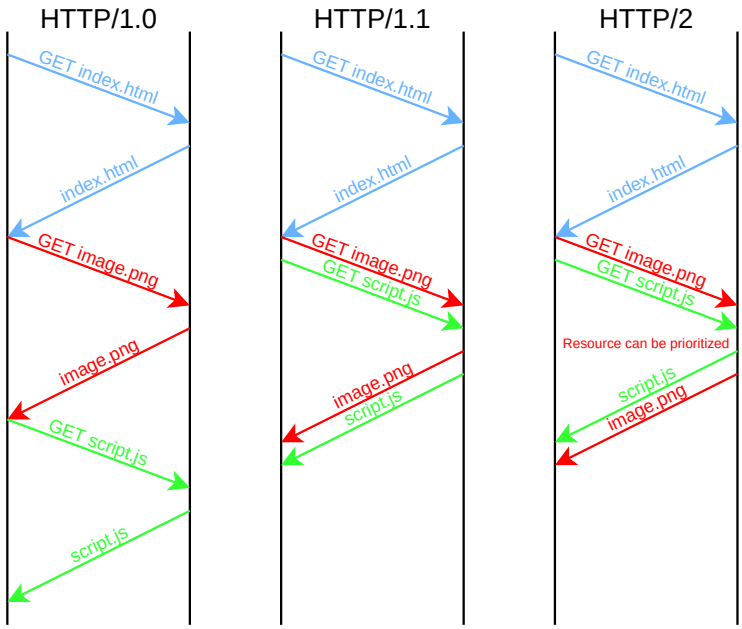


Figure 2.2: HTTP mode to fetch resources

QUIC principles

As mentioned, the QUIC protocol is on top of UDP. It was officially standardized in 2021 [27]. It brings several functionalities. The main one is the use of streams. This avoids losing time when a resource is fully loaded but the previous resource is delayed, so since `TCP` works on one stream, the data loaded must wait for the previous data to be loaded.

Also `QUIC` uses monotonically increasing packet numbers. In `TCP` we can have ambiguity for estimating the RTT. This is illustrated in figure 2.3, if we send a packet and reach the Retransmission Time-Out (RTO), we send the packet again with the same sequence number. After, when we receive the acknowledgment. We can't be sure from which packet sent does the acknowledgment came.

Another important principle is that almost all fields are encrypted. This is a good point about privacy but also from a measurement perspective. In TCP they are web caches that can alternate the performances. With QUIC this is not possible so we can be more confident about our measurement.

But while in TCP cryptographic handshake takes a considerable time (2-RTT), even more, if the application is also using TLS for security (3-RTT). In QUIC the full connection establishment is reduced. The protocol can do the TLS negotiation while doing the QUIC handshake. Furthermore, if a client was connected to a server before, he can reconnect using a 0-RTT handshake using a token.

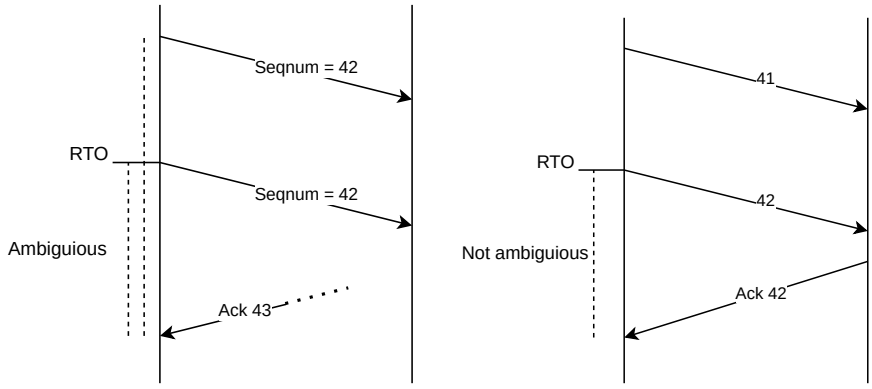


Figure 2.3: Difference between TCP acknowledgment ambiguity and QUIC solution

Qlog

Debugging an internet protocol is not an easy task. The classic approach is to use packet capture. For QUIC, Robin Marx developed an approach called `qlog` [34].

The `qlog` is a file derived from a QUIC connection that contains information about what happened. The file is in JSON format. First, we have the global information, for example, which version of QUIC is used, what implementation of QUIC is used, etc. Then we have the connection events.

They are several events but they are not all mandatory. In the draft describing the different events [35], there are three different levels of importance: *Core*, *Base* and *Extra*. The *Core* events are the mandatory events, and the *Base* are the events that should be in a `qlog` file. And finally, the *Extra* events can also be added but are only useful for low levels.

HTTP/3

In 2022 a new version of `HTTP` was officialized, version 3 [9]. This version is used on top of the QUIC protocol. This version benefits from all QUIC features, above all the latency improvements with the streams and the lower connection establishment. As `HTTP/2` this new version was quickly adopted. Following w3 tech, in October 2022, 25.9% of websites use `HTTP3` (`https://w3techs.com/technologies/details/ce-http3`). Also, most major browsers are using it (`https://caniuse.com/http3`).

Since the browser can't know in advance whether a website supports `HTTP3`. So to know this, the client has to send a normal request in `HTTP/2`. Then, the server will respond with a header `alt-svc` indicating the support of `HTTP/3`.

2.2 Performance Testing

In this section, we will describe various performance testing. We will first describe the early ones, then discuss the metrics we are using. And finally, list several tests available when we were writing this document.

2.2.1 Well-known metrics

Throughput

The throughput is defined as the number of data a host can send during a period. The classic way to define it is using *bits per second*. This metric can be different depending on what we measure. For example, measuring the throughput of the link will be different from measuring the throughput of TCP. This is because TCP headers and retransmission add overhead.

To avoid that, we generally speak about goodput to define the number of *useful* data a host can send during a period. The utility of the data depends on the situation, when downloading a file from a website the time taken will depend on the goodput of the `HTTP` connection.

Latency

The latency is the time taken to perform a Round-Trip, so the time taken to reach the other host and the time for the acknowledgment to come back. The latency is usually measured in milliseconds (ms).

In today's life, latency becomes very important. When doing videoconference or video streaming or gaming. The users want their data to be sent as fast as possible and so the latency matters more than throughput [18].

Losses

Using a large network like the internet, losses can happen. They can come from diverse reasons. For example due to a faulty physical layer or traffic policies to limit the flow. For losses, we usually speak about the percentage of losses.

2.2.2 Early testing

ARPANET performances

With the first version of the internet came the firsts performances measurement. Several people tested the performances of the ARPANET [1, 29]. At the time, the throughput reaches up to 40 Kbps. The latency was also measured, but the main focus at that time was to have a reliable transfer.

First tools

The first open measurement tool was the `ping` util. The `ping` was developed by Mike Muuss [37] in 1983. The util sends an `ICMP` packets with an `ECHO` message, then the other host replies with another `ECHO` message. With that, the client can compute the round-trip time (RTT) but mostly, can know if the other host is reachable.

A few years later, inspired by the `ping` util, Van Jacobson created a tool called `traceroute`. The `traceroute` util uses also `ICMP` packets but the idea is to limit the Time-To-Live (TTL) value. When a host receives a packet with a TTL set to zero, the host will respond with an `ICMP` indicating an error. With the error response, the client can get the other host with the source address on the error packet and also compute the RTT. By gradually increasing the TTL, the client can trace the route taken by the packets.

Iperf

The `ping` and `traceroute` tools don't measure throughput. One of the first is `Iperf` [24]. This is a Command Line Interface (CLI) that generate TCP or UDP streams between two host. The tool comes with a lot of tuning parameters to test several scenarios.

2.2.3 Internet tools

Nowadays, with the spread of the internet, almost everyone has an internet connection. Leading people without a strong networking background to use it. As they pay for their internet access, those people would be very interested to know their internet capacity.

Tools like `Iperf` are not attractive for experimented users since it's a Command Line Interface (CLI) tool and it requires to have two hosts. A better solution could be to have a Graphic User Interface (GUI). In the following section, we describe several GUI tools which are available on web browsers. A more exhaustive list is available on the annexe A.1.

Ookla

The most known tool today is the site by Ookla[2]. This tool measures the ping then the download bandwidth and finally the upload bandwidth. During our work, the website added the measurement of the ping during bandwidth probing tests. The tool is also available in various native apps for devices (Android, IOS, Windows,

[2] `https://speedtest.net`

Mac, Chrome, AppleTV and CLI). They also propose a very large number (around 18,000) of test servers around the world. Finally, the tool uses only `HTTP/1.1` connections. They are doing that to avoid the multiplexing which comes with `HTTP/2`.

To measure idle latency, the tool is doing several *hello* requests. Those requests are small and so is the response (42 bytes). Then for download and upload, the tool opens a WebSocket conversation to do *ping/pong* messages to measure latency during bandwidth probes. The bandwidth estimation is done by using multiple connections.

CloudFlare

Another website is the one made by Cloudflare[3]. This test is taking more time than Ookla's but it takes more scenarios in the count.

For bandwidth probing, it uses multiple iterations. First uses 10 *probes* of 100kB, then 8 *probes* of 1MB, then 6 *probes* of 10MB and finally 4 *probes* of 25MB. By the *probes* we mean a download or an upload in the function of the way we want to measure. To measure latency under working conditions the tool is using several large downloads/uploads while doing small uploads/downloads. Finally, the tool measures the loss rate by using WebRTC[4] which allows the use of streaming and so this allows the tool to know the packet lost.

LibreSpeed

All those measurement tools are great. But their implementation is not available. The most popular Open-Source speedtest is LibreSpeed[5]. The goal used by this website is to have a minimalist implementation. The front-end is a JavaScript worker that makes XML requests. The back-end part was originally written in PHP but now there exists a NodeJS and a Go version.

In this section, we summarise the implementation choices of LibreSpeed which are provided in the *doc* document inside the repository[6].

Meta-data To know more about the user. LibreSpeed uses the `ipinfo` API[7]. With that, the tool can estimate the relative distance between the server and the host. The goal with that is to have confidence in the results. A test point far away will cause higher latency.

[3] `https://speed.cloudflare.com`
[4] `https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API`
[5] `https://github.com/librespeed/speedtest`
[6] `https://github.com/librespeed/speedtest/blob/master/doc.md`
[7] `https://ipinfo.io`

Measuring latency The tool is measuring latency and jitter. To do so, the client is requesting a resource that returns an empty `HTTP` response (only headers). There is a parameter that indicates the number of pings to make to measure latency, the `count_ping`. By default, this parameter is set to 10. To get the round-trip value, the util store the initial timestamp where the request was made and compares it to the timestamp when the response happened. If allowed, the util uses performance API from the browser.

Measuring capacity To measure user capacity, LibreSpeed downloads or uploads a large file containing random bytes. But since the congestion control is done at the TCP/QUIC layer, the measured capacity may be lower than expected. So the client is making several `HTTP` requests in parallel. By default, it uses 6 requests for download and 3 requests for upload.

The interesting things about those requests are first, to use a randomize argument inside the request. This randomness is to avoid internet caches, which can alternate the results. Then, the data is generated such that the compression will not be efficient. And to enforce those, the back-end disables cache and compression.

Unfortunately, the XML request librairie only provides the real data exchanged. So it's not possible to accurately measure the throughput but we can measure the goodput. To deal with that there is a parameter called `overheadCompensationFactor` which tries to estimate the throughput. By default, this parameter is set to 1.06. This value comes from empirical testing.

Point of testing LibreSpeed website has several servers for making measurement requests. But unlike the number of servers that have Ookla, LibreSpeed only has 30 points of testing. Most of them are located in Europe, but there is a point of test in each continent.

By looking at the points of the test dataset[8], we can know that most of them are using `HTTP/1.1` or `HTTP/2` but only one supports `HTTP/3`: Singapore (Salvatore Cahyo).

Measurement Lab

A last interesting tool is the Network Diagnostic Tool (NDT) proposed by the Measurement Lab[9]. The difference with other websites is first that the test is done under 10s. And then it uses WebSocket to make all measurements.

It receives several small messages for ping probes. Then receives several large messages for download probing and finally sends large messages for upload probing.

[8] `https://librespeed.org/backend-servers/servers.php`

[9] `https://speed.measurementlab.net`

The huge difference is that the client itself doesn't look at metrics. The server replies to messages with connection info containing CCA information, connection information and finally a bunch of TCP_info information.

Furthermore, the Measurement Lab data is freely available [36] using Google Cloud Storage and can be used using Google BigQuery.

TCP_INFO As mentioned above, the NDT test uses TCP_info to have information on the metrics. TCP_info is a bunch of statistics about a TCP connection. When using a TCP socket a user can extract those statistics. For example in Linux people can use the socket option `TCP_INFO`[10].

But unfortunately, those statistics are not standardized, so there exist several versions of TCP_info [11] [12] [13] [14]. For example, the M-Lab extends the one from the Linux kernel and is showing those statistics from the NDT with their open dataset[15].

2.3 New Metrics

2.3.1 Bufferbloat

The bufferbloat is when a router is buffering too much data. And so this leads to a latency increase. The phenomena were already known in 1985 [5] but researchers start focussing on it in the 2000s [8].

A tool to measure it is `flent`[16] [23]. This tool uses *Realtime Response Under Load* (RRUL) [44]. The method uses four concurrent TCP connections in each direction while running UDP/ICMP latency measurement.

Responsiveness Under Working Conditions

In 2021, Apple proposes a new metric called RPM for Round-trip Per Minute [39]. The goal of this metric is to evaluate the latency under working conditions, and so the impact of bufferbloat.

[10] https://man7.org/linux/man-pages/man7/tcp.7.html

[11] `https://github.com/torvalds/linux/blob/5bfc75d92efd494db37f5c4c173d3639d4772966/include/uapi/linux/tcp.h#L214`

[12] `https://raw.githubusercontent.com/openbsd/src/ced6d44d3b6b8e0b34987d39fdb7319e7ac81cb1/sys/netinet/tcp.h`

[13] `https://github.com/apple-opensource/xnu/blob/eb45a4f3d6bc33c958fcbf4ea7388da13ae63a2e/bsd/netinet/tcp.h`

[14] `https://learn.microsoft.com/en-us/windows/win32/api/mstcpip/ns-mstcpip-tcp_info_v1`

[15] `https://www.measurementlab.net/tests/tcp-info/`

[16] `https://flent.org/`

The paper proposes an algorithm that creates traffic on the network. And while having a loaded network, evaluates the responsiveness. The benefit of this metric is that the more RPM you get, the better you have.

The algorithm work as followed: on one side we measure latency doing several types of ping (TCP, TLS, HTTP and HTTPS) with a defined rate (by default 100 probes per second), and on the other side, at each interval (by default each second), we add a load-generating connection. And then we check if the goodput has saturated. If so, compute the responsiveness at the current interval. And then with multiple responsiveness probes, if they are stable we can finish the test.

2.3.2 Spurious retransmission

A spurious retransmission happens when the sender reaches the Retransmission Time-Out (RTO) and then it receives the waited acknowledgment [7, 31]. This type of notification indicates a wrong value for RTO. There are also spurious retransmissions on QUIC [26]. To take this to a metric, we consider the percentage of spurious retransmission among the losses.

Summary

We saw that there are a lot of different tools to measure user performance. Most of them focus on bandwidth capacity and idle latency but only some of them are starting to use working latency. And all those tools are using a network stack that uses TCP.

Chapter 3

QUIC_INFO

To better understand some QUIC behaviours. We need to output some information about the connections. As mentioned before TCP have tcp_info but since this is not standardised, there exist several versions of it.

So in this chapter, we will first define a set of fields for QUIC to have useful statistics as TCP_info. Then we compare it to some TCP_info versions. And finally, describe how we implement a prototype version and discuss future implementation.

3.1 Defining fields

In this section, we describe all the fields of QUIC_info. Most of them are inspired by some TCP_info implementations, particularly from the version used by the Linux Kernel. Also, some fields for TCP_info are there for debugging purposes, we choose to focus more on metrics which are more relevant to our work than debugging.

3.1.1 Fixed Parameters

RTO The Retransmission Time-Out is the time in milliseconds before sending a packet again.

ATO The Acknowledgement Time-Out is the time in milliseconds before sending an acknowledgement again.

Snd-MSS The Sender Maximum-Segment-Size is the maximum size in bytes used by the sender.

Rcv-MSS The Receiver Maximum-Segment-Size is the maximum size in bytes used by the receiver.

3.1.2 Timing

Last-data-sent This is the timestamp (in ms) of the last packet sent that contains a data frame.

Last-ack-sent This is the timestamp (in ms) of the last packet sent that contains an acknowledgement frame.

Last-data-recv This is the timestamp (in ms) of the last packet received that contains a data frame.

Last-ack-recv This is the timestamp (in ms) of the last packet received that contains an acknowledgement frame.

Min-rtt This is the lowest time in milliseconds taken to perform a Round-Trip.

Smooth-rtt This is the smoothed time in milliseconds taken to perform a Round-Trip.

Var-rtt This is the variance of all Round-Trip Times.

Max-rtt This is the highest time in milliseconds taken to perform a Round-Trip.

3.1.3 Data exchanged

Bytes-sent This is the total number of bytes sent on top of the UDP layer. This includes retransmission.

Bytes-recv This is the total number of bytes received on top of the UDP layer.

Frames-sent This is the total number of QUIC frames sent.

Frames-recv This is the total number of QUIC frames received.

Data-frames-sent This is the total number of QUIC data frames sent.

Data-frames-recv This is the total number of QUIC data frames received.

Dgrams-sent This is the number of UDP datagrams sent.

Dgrams-recv This is the number of UDP datagrams received.

Data-dgrams-sent This is the number of UDP datagrams sent that contain a data frame.

Data-dgrams-recv This is the number of UDP datagrams received that contain a data frame.

Data-sent This is the number of bytes sent in the QUIC frames without including retransmissions.

Data-recv This is the number of bytes received in the QUIC frames without including retransmissions.

3.1.4 Others

Lost This is the number of lost events.

Retrans This is the number of frames retransmitted.

Pacing-rate This is the current Pacing Rate seen (in bytes per second).

Max-pacing-rate This is the highest Pacing Rate used during the transfer (in bytes per second).

Delivery-rate This is the global estimation of sending rate (in bytes per second).

Cwnd This is the current size of the congestion window in bytes.

Next-send-pkt-nbr This is the number of the next packet that will be sent.

Next-recv-pkt-nbr This is the number of the next packet that will be received.

Duration This is the total time elapsed.

Spurious This is the number of spurious retransmission.

3.2 Comparison with TCP_info

In this section, we compare different implementations of TCP_info. We based on the following implementations: Linux Kernel[1], OpenBSD[2], Apple[3] and Windows[4].

Some TCP_info fields are not relevant for a QUIC implementation. For example, the TCP FSM status is not applicable for QUIC. Another example is anything

[1] `https://github.com/torvalds/linux/blob/5bfc75d92efd494db37f5c4c173d3639d4772966/include/uapi/linux/tcp.h#L214`

[2] `https://raw.githubusercontent.com/openbsd/src/ced6d44d3b6b8e0b34987d39fdb7319e7ac81cb1/sys/netinet/tcp.h`

[3] `https://github.com/apple-opensource/xnu/blob/eb45a4f3d6bc33c958fcbf4ea7388da13ae63a2e/bsd/netinet/tcp.h`

[4] `https://learn.microsoft.com/en-us/windows/win32/api/mstcpip/ns-mstcpip-tcp_info_v1`

linked to the `sacked`. However, we add fields to QUIC_info that are not applicable for TCP. An example is the number of frames sent.

To see with more detail the mapping between TCP_info implementations and QUIC_info, we describe it in annexe B.

Linux Kernel The majority of QUIC_info fields are inspired by the TCP_info implementation inside the Linux Kernel, and so, look very similar. We choose this implementation as the main basis because Linux is the most used OS for servers [6].

Since there is no official documentation about the fields aside from the names described in the header file `tcp.h`. In order to understand the fields, we based on the unofficial documentation provided by M-LAB for their speedtest (`https://www.measurementlab.net/tests/tcp-info/`).

OpenBSD Kernel The OpenBSD TCP_info is derived from the one in the Linux kernel. However, the fields were copied from an older version of the Linux Kernel than the one we used. So it adds some fields but also has a lack of the latest added.

Apple The Apple implementation seems also derived from the Linux Kernel. But the main differences are fields for particular access points (cellular/Wi-Fi/wired), for example, the number of bytes sent using Wi-Fi or the number received using cellular. In our experience, we are only using one access at the same time, so we choose not to add a such thing to our fields. Also, in Apple products, it's simple to map an interface with an interface since this is the same system. But with Linux or Windows, this will be much more complicated. Furthermore, this idea is not scalable if a new type of access point is created.

The Apple TCP_info is very complete. There is a particular focus on interfaces used for a connection (cellular, Wi-Fi and Wired). For implementation reasons, we choose to not add this idea to QUIC_info, this will be discussed in the implementation section.

Windows The Windows implementation is the one with the lowest information.

3.3 Implementation

After defining fields and metrics, we want to have a tool to extract them from a QUIC connection. We will first describe our implementation using `qlog` and then discuss future implementation.

3.3.1 Qlog implementation

Since `qlog` of the connection contains a lot of information about our fields, we choose to use those files to get the statistics. Also, the `qlog` is implemented in almost every QUIC implementation so we believe we can have a tool that covers all QUIC implementation. But there are some attention points.

Qlog version

First, the `qlog` is being standardised but the draft is still evolving. Thus we can have different `qlog` versions depending on the QUIC implementation. For example, `picoquic` implements `qlog` version `draft-00` while `aioquic` implementation uses version `draft-03`.

Luckily, the differences are not too impacting. When making a code for a version, it's not too difficult to adapt the code. For example from `draft-00` and `draft-03` the main difference is the format of the events. In the first one, we have a 4-Tuple while in the second this was changed to a JSON object. We didn't spend much time on making a fully adaptative code that can change code by matching `qlog` version but we believe this can be possible.

Qlog format

The next difficult thing about the `qlog` is the format. The original version proposes a `JSON` format. But for performance reasons, there are other formats such as `ND-JSON` or `JSON-SEQ`. For example, `picoquic` uses simple `JSON` while `quic-go` uses `ND-JSON` format.

As for the `qlog` version. We highly think it's possible to handle multiple formats with the same tool. But to simplify this, we made a converter from `ND-JSON` to `JSON` format.

Qlog events

The last thing to take care of is events. Some `qlog` events or fields in an event are optional. So if an implementation is not using some events, the QUIC_info can have a lack of information.

Only connection summary

The implementation we made with `qlog` is working well. But only to make a summary of a connection. The `qlog` are written to a file so we can't extract QUIC_info on an active connection. We will have a gap between what is going on and what we get.

3.3.2 Futur implementation

To avoid the issue with incomplete `qlog` and having the ability to get QUIC_info information within a connection. Our idea is to implement it inside a QUIC implementation.

The `picoquic` implement already has a statistics tool like TCP_info called `quicperf`[5]. But we want to have a standardized version so that from a client we could ask for the information, receive it and know the format despite the QUIC implementation.

[5] `https://github.com/private-octopus/picoquic/blob/master/doc/quicperf.md`

Chapter 4

NesQUIC

Our first tool is called NesQUIC it was originally developed by François Michel and Maxime Piraux[1]. This is a client-server test using `picoquic`[2]. This tool performs large transfers using the QUIC protocol in both ways, upload and download. It runs several transfers by varying the CCAs but also by fixing the sending rate.

4.1 Methodology

With this tool, we want to compare different behavior of QUIC during a connection. First, we compare congestion control algorithms to know their behavior with QUIC. And then we want to know how the network behaves when we choose a fixed rate that can be higher than the user capacity.

Testing process

The client sends first a message containing the test information formatted as a JSON file. The fields are shown in the table 4.1. The first field is the `testname` which is *bulk* or *limited_transfer*.

In the first case, for the *bulk* test, the client has to put another parameter to indicate the congestion control algorithm used by the sender (`cc_alg`). Note that when speaking about *reno* this means the New Reno CCA. In the other case of a *limited_transfer* test, the client adds bytes per second field (`bps`). The sender will limit itself to this sending rate.

The other fields are common to the two test types. The client has to specify the `way`. If the way is *download*, then the server is the sender. And if the way is *upload*, the client will be the sender.

[1] `https://github.com/francoismichel/nesquic`

[2] `https://github.com/private-octopus/picoquic`

After that, the client has to indicate the number of bytes that will be transferred (`size`) and the maximum duration of the test (`max_duration_s`). And last but not least, indicate an interval duration (`interval_duration_s`). At each interval, the client will log the number of stream data sent/received, the number of QUIC data sent/received, the number of datagrams sent/received and finally the real elapsed time.

Argument	Value
testname	bulk/limited
cc_alg	cubic/bbr/reno/fast
bps	number of bytes per second
way	upload/download
size	number of bytes transferred
max_duration_s	maximum duration in seconds
interval_duration_s	time in seconds between two intervals

Table 4.1: Arguments sent by NesQUIC client

4.2 Implementation

4.2.1 Server hosting

To host the NesQUIC server, we choose to make a `systemd` service. This solution avoids using a container or a virtual environment that can add overhead.

4.2.2 NesQUIC client

To test our tool in the wild, the idea is to have a simple client to make it the most available for people. So we make a Python client that launches the NesQUIC client. The idea is to compare several setups with only one client.

Tests set

First, our client performs bulk tests with four different congestion control algorithms: BBR [14], New Reno [21], Cubic [41] and Fast [28]. With that, we have two loss-based (New Reno and Cubic), one delay-based (Fast) and one hybrid (BBR). All those are the ones implemented inside `picoquic`.

Then for the limited test, the client will first estimate the bandwidth by taking the highest estimated bandwidth of the three bulk tests with the same congestion control algorithm (by default BBR). After that, we take several percentages of the estimated bandwidth and use them for fixed bandwidth tests.

We probe six steps: 30%, 50%, 80%, 90%, 100% and 110%. With this, we can see the impact of a low bandwidth on several metrics (i.e. latency and losses) and also what happens when we take a higher bandwidth than what we have. However, we know that the bandwidth estimation is not perfectly accurate but the main goal is mostly to know the magnitude of the user bandwidth.

Log the test

To collect data from each transfer we have several solutions. We can record the dump of the transfer with tools such as `Wireshark` or `tcpdump` and store the private key from `picoquic`. However, this solution produces very large files since it stores the packet's payload. And also, we don't have information about the internal state (e.g. the variation of the congestion window).

Another option is to collect `qlog` files. As explained in chapter 3, with those files we can derive a lot of metrics.

The `picoquic` implementation provides a last option, this is called `perflog`. The idea is the same as QUIC_info or TCP_info, this is a bunch of metrics about the connection.

So we choose to keep the `qlog` and the `perflog` for the client. But since the server can also generate `qlog`. We generate them on both sides. And after a test, we can retrieve the server `qlog` by matching the QUIC connection ID.

Test in the wild

To test our tool in the wild, we made a docker image to launch all tests needed [3]. And since the container will be in an unsupervised environment we have several things to take care of.

First, we have to collect meta-data about the machine. We are logging the network interfaces, and the IP/hostname/relative location using the website `https://ipinfo.io`. With this website, we are also estimating the distance (in km) from the server. We are also logging the IP source and destination, so with that, we have the 4-tuple used by the internet to decide paths. We also logged the CPU model used by the host which can influence the sending rate. Finally, we are logging the maximum size of the receiving window (`rmem_max`).

Then we have to pay more attention to the received buffer size. So we check the value inside the `/proc/sys/net/core/rmem_max` file. However, Docker isolates this part of the kernel so we can't see the value. Thus, we decide to use the `host` mode for networking (`https://docs.docker.com/network/host/`). This mode allows us to look at the value. The QUIC-Go wiki recommends using a 2.5 MB

[3] `https://hub.docker.com/r/felixgaudin/nesquic`

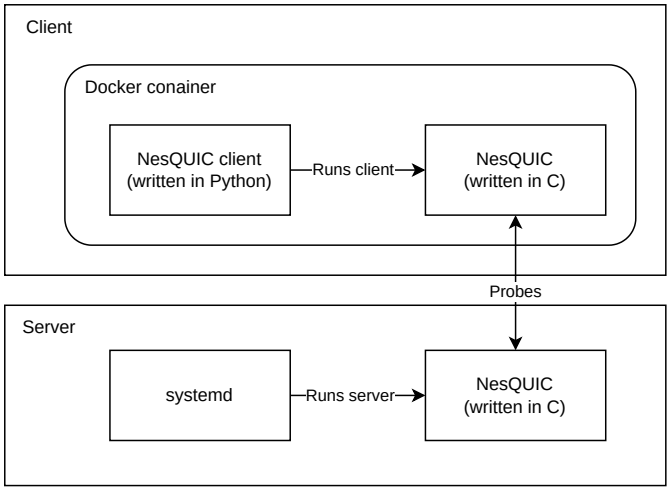


Figure 4.1: Architecture of NesQUIC

buffer to avoid being limited by this buffer for high-bandwidth connections. So we won't let a host perform the test if the size is below this value.

Finally, to collect client logs we want the transfer to be as fast as possible. One way of doing that is to compress the data sent. For uploading the logs to the server, we are using the `request` Python librairie[4]. This allows compression but the best way to compress qlog is to use CBOR compression [11, 33].

The complete architecture of the NesQUIC client is described in figure 4.1. In section 5.3 we will describe more about how the results server is made.

[4] `https://requests.readthedocs.io/en/latest/`

Chapter 5

LibreSpeed

Librespeed is an open-source project to have a speedtest from a browser. For improving the tool, we have two goals. The first one is to implement the RPM methodology inside the LibreSpeed client. The second one is that we want to use the QUIC protocol to make the tests.

5.1 Methodology

5.1.1 Adding responsiveness under working conditions

The RPM methodology [39] requires making a bulk transfer while sending ping probes. Luckily, LibreSpeed already has a way to do those types of probes.

5.1.2 Using QUIC

The next thing we want is to measure the responsiveness under working conditions using the QUIC protocol. For browsers, there are two ways of using QUIC. The first is by using `WebTransport` and the second to use the `HTTP/3` protocol. Since the `WebTransport` API is not available for several browsers[1], we choose to use `HTTP/3` protocol. This protocol is supported by several browsers[2].

For having a `HTTP/3` transfer between the client and the server. We choose to put a reverse proxy that supports `HTTP/3`. Then inside the server, the reverse proxy can call the backend using `HTTP/1.1` or `HTTP/2` connections on localhost. The localhost transfer is considered negligible in terms of performance because the bottleneck should be in the network which will have lower capacity and highest latency. This is illustrated in fig 5.1.

[1] `https://developer.mozilla.org/en-US/docs/Web/API/WebTransport`
[2] `https://caniuse.com/http3`

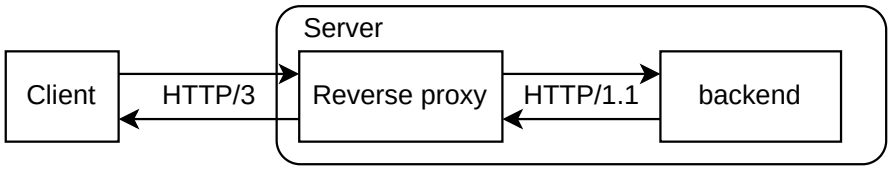


Figure 5.1: Architecture using a `HTTP/3` capable proxy

Also as QUIC_info uses `qlog` files. We want a reverse proxy capable of generating `qlog`.

Handling multiplexing

Most speedtest websites are using `HTTP/1.1` connections for estimating bandwidth because they are not using multiplexing. However, `HTTP/3` is using it. The drawback with multiplexing is that it's reusing connections for resources in the same location. And if we have one connection, the congestion control algorithm of this connection can choose to not use all available bandwidth depending on the implementation.

To avoid that, we choose to do all probes on wildcard links. This means that for each connection we use another URL that indicates the same server, for example, `con42.speedtest.com` will be on the same server as `con23.speedtest.com`. With that, the browser is thinking that those are new connections and so creates new ones. The disadvantage of this idea is that each time the client has to resolve the new URL using DNS queries.

5.2 Implementation

In the implementation of the RPM methodology inside LibreSpeed, we faced several challenges:

RPM back-end matching LibreSpeed back-end

The first challenge is to remain the backend unchanged. This will allow us to use the modified version of the test using existing endpoints. The RPM algorithm back-end requirement uses equivalent routes as LibreSpeed. The `garbage` route for sending or receiving data in LibreSpeed is translated to a `large_download_url` and `upload_url` in RPM. And the `empty` route is translated to `small_download_url`.

RPM Routes	LibreSpeed equivalent
large_download_url	garbage (GET)
upload_url	garbage (POST)
small_download_url	empty

Table 5.1: Translation of RPM routes to LibreSpeed backend

Estimate the overhead

The LibreSpeed code measures the goodput of the user. But users want to measure their throughput. In the LibreSpeed documentation (`https://github.com/librespeed/speedtest/blob/master/doc.md`), the authors explain that they multiply the goodput with a parameter called `overheadCompensationFactor`. This factor is set by default to 1.06.

But since we don't have any dataset for measuring the overhead compensation factor. We decide to set this factor to one so that we measure the goodput. We believe that with data collection we should be able to compute a new factor for QUIC.

Usage of QUIC

As explained in section 5.1.2, we want to use a reverse proxy that allows a client to use `HTTP/3` connections. To do it we explore several solutions. The first one was using a patch provided by Cloudflare `Quiche` implementation into `Nginx` proxy. But `Nginx` has also a solution provided by them called `nginx-quic`. Another solution was `Caddy` written in *Go*.

As for NesQUIC, we want to output `qlog` about connection to analyze them. But none of them can output such files. We choose to modify `Caddy` because it seems easier to modify because it uses the `quic-go` implementation (`https://github.com/quic-go/quic-go`).

Follow RPM updates

The RPM is a draft so during our experiment the draft evolved several times. Since the changes were not too impacting and we had contact with the authors. We decide to follow the updates. By doing this we benefited from the better version of the algorithm. For example, the evolution to draft-02 lowers the time taken by the test. And also we were able to give feedback to the authors so that they can improve the draft.

Modify RPM parameters

The RPM has several parameters, one of them is the *Maximum responsiveness probes per second* (MPS). In the draft, they indicate the default value to 100. By testing on browsers, we found that this number of probes was too high. The browser couldn't measure bandwidth because it was too busy doing responsiveness probes. By testing, we found that 10 MPS was a good compromise.

5.3 Coordinator

For both experiments, we generate a lot of data. We want to retrieve them for analysis. But both tools are also generating a lot of load during the probes and if two tests are running at the same time, one can affect the other. So we want to only have one test at a time.

General idea

As explained in the figure 5.2, the client has to ask for a token to the *coordinator* server. The *coordinator* will ensure that at most one test is running. Then the client can perform the test (NesQUIC or LibreSpeed). And finally, it only has to say to the server that he finishes the test. The *coordinator* also implements a timeout mechanism to avoid deadlock if a client requests a test and never releases the lock.

Implementation

For implementation, we made the coordinator with a `Flask` server. The server uses a first route for giving a token (`/get-id`). He has first to check if another test is running. If the previous test has timed out or there is no test running, the server generates a new token. He has also to ensure the token was not used by a previous test. We want this because we want to be able to link a test with the results, and the best way we found is to use a such token. To avoid collision as much as possible, we are using a 20 characters token made of ASCII letters and digits. This made 62^{20} ($\approx 7 \cdot 10^{25}$) possibilities so we have a great margin for the number of tests.

The next route is for releasing the lock (`/release-lock/<test_id>`). The client has to provide his test token to ensure this is the current test. For NesQUIC, there is a particular route `/upload/<test_id>`. The route is different because the client must upload the results of the test.

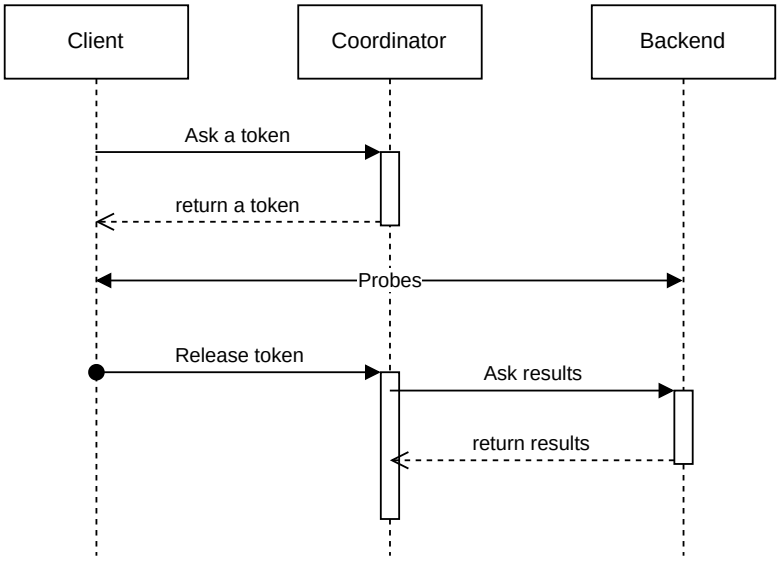


Figure 5.2: Diagram of how the client interacts with the coordinator and the backend

Finally, the server has a route for NesQUIC that allows the client to retrieve `qlog` files from the server (`/download/<file>`). The server `qlog` filename can be derived from the client filename using the connection id.

route	Value	Arguments
/get-id	get a token	type of the test
/release-lock/<test_id>	release the lock of the test	the token (for LibreSpeed the test results)
/download/<file>	download qlog from the server	the server qlog file
/upload/<test_id>	upload qlog file for NesQUIC (releases the lock)	the token

Table 5.2: coordinator routes

LibreSpeed special case

Since browsers don't allow us to log a lot of information, the client can only upload the test summary from it. But on the server side, we can collect more. As mentioned before, we use `Caddy` for the proxy, and it can generate `qlog` files. So when releasing the lock for a LibreSpeed test, the *coordinator* will take all `qlog` generated files and put them in the result folder for the given token.

However, it's very difficult to avoid a user to fetch the front-end page. And this fetch could generate `qlog` since it may use `HTTP/3`. Thus we will have to sort all the results files to avoid them. Our solution is to put the front-end on another server and make probes calls to the back-end server.

The last thing to collect is to know if the client chooses to make `HTTP/3` connec-

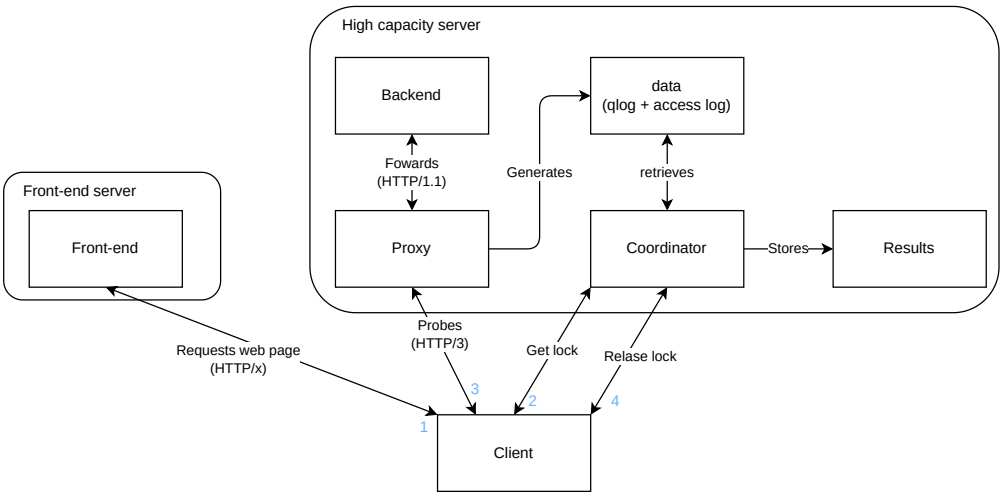


Figure 5.3: Architecture of LibreSpeed with the coordinator server

tions. Even with `Alt-Svc` headers, the browser still can choose to continue using `HTTP/2` connections. On the front-end it's impossible to know which protocol a request used. The debugging tool gives this information, but it requires experimented users to use it and we want the test to be automatized. So we find a better way to do this. A `Caddy` option allows the proxy to log access including the URI called, the protocol used and some client information. So when releasing the lock, we can go into those logs and store them.

The figure 5.3 illustrates all those small specific behaviors. And we can see that it still follows the abstract specification of figure 5.2.

Chapter 6

Validation of the tools

In this chapter, we analyze the behavior of our tools in a controlled environment. This allows us to validate our tools. Since we know the network parameters and topology, we can measure the difference between what we expect and what we measure.

6.1 Experimental setup

We took 3 Intel NUC NUC5PPYH that have an Intel Pentium n3700. And connected them using ethernet with a capacity of 1Gbps. One is used as the router, one as the client and the last one as the server. All devices are using Ubuntu 20.04.6 LTS with kernel version 5.15.0-72-generic. This setup is described in figure 6.1.

On the router, we use the `tc(8)` utility to limit bandwidth and latency. We use the following commands with different combinations on both interfaces. To vary the combinations we use a Token Bucket Filter[1] with a buffer of 50kB, the rate we want and a limited latency of the value we want. The latency is limiting the number of times a packet can be in the TBF. So we add a NetEm filter to add

[1] `https://man7.org/linux/man-pages/man8/tc-tbf.8.html`

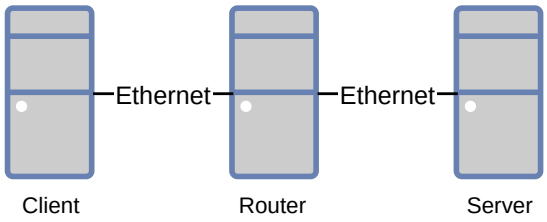


Figure 6.1: Architecture of the experimental setup

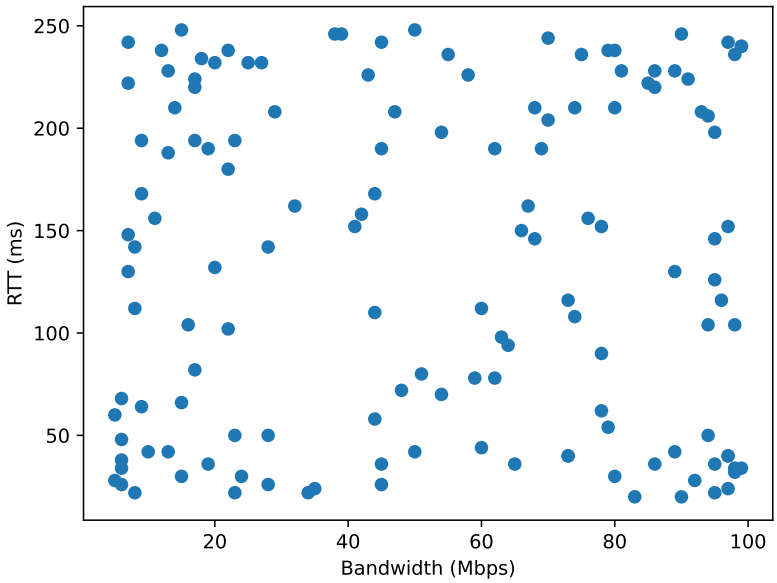


Figure 6.2: Test set by varying bandwidth and latency using WSP

virtual latency.

```
tc qdisc add dev {interface} root handle 1: tbf \
    rate {bandwidth}mbit burst 50k latency {latency}ms
tc qdisc add dev {interface} parent 1: netem delay {latency}ms
```

Also, the router is used as a DNS server. For LibreSpeed we need a DNS revolver to get all subdomains. So we use `dnsmasq` on the router to handle this.

6.1.1 Test set

We use an experimental testing methodology which is using the WSP algorithm [43][2] to generate test scenarios:

We choose the bounds by looking at performances without limitation. For NesQUIC we get around 180Mbps and 5ms while for LibreSpeed we get around 200Mbps and 10ms. So we chose a range from 5 to 100Mbps for bandwidth. And from 5 to 125ms for latency. But since the limitation is done on both interfaces, the latency is doubled, and so from 10 to 250ms.

[2] `https://docs.rs/wsp/latest/wsp/`

6.1.2 Automate tests

For NesQUIC since the test can be launched using CLI this is very easy to do. The router put the traffic controller and then launches the tests on the client using `ssh`.

However, for LibreSpeed this is a bit more complicated since the user has to click on the button to start the test. There are several solutions for this, but another constraint we have is that the browser must be able to use `HTTP/3` connections. To do this, we use the `browserless` Docker image[3] that launches a Chromium instance, which can use `HTTP/3`, and that can be controlled by puppeteer[4] using a WebSocket.

6.2 NesQUIC

For the validation of the NesQUIC tool, we first want to know if the *bulk* tests are good. So to know if `picoquic` is implementing the different CCA as expected. Also, for the *limited* we need BBR to be good at estimating the bandwidth to run the test on fixed bandwidth after. Then, we want to know if by taking a higher bandwidth (110% of the measured bandwidth), the connection will be saturated.

6.2.1 Bulk tests

In figure 6.3 we look at how the different CCAs are using the available bandwidth. The lines are the Cumulative Probability Distribution (CDF) of the bandwidth utilization for the different CCAs. The *bandwidth utilization* is the percentage of the available bandwidth taken by the CCAs.

We first can see that the upload and the download are looking very similar. It's not surprising because the limitations are symmetric, the devices are the same and the QUIC implementation is the same. So for the next analysis, we'll only consider upload since the download will be the same.

We can also see that Cubic and BBR are almost always taking a huge percentage of the available bandwidth and New Reno (reno in the figure) is a bit lower but it's taking however a significant amount of bandwidth. Those are normal since they are loss-based or hybrid. In the case of a delay-based CCA, we can see that FAST is taking a small amount of the bandwidth in general.

Then we look at the RTTs observed. The figure 6.4 shows us multiple metrics for each CCA. As figure 6.3, the figures are showing the CDF, but here we look at different latency metrics. We choose to take the mean, the median and two high percentiles (80% and 90%).

[3] `https://www.browserless.io/docs/docker`

[4] `https://pptr.dev`

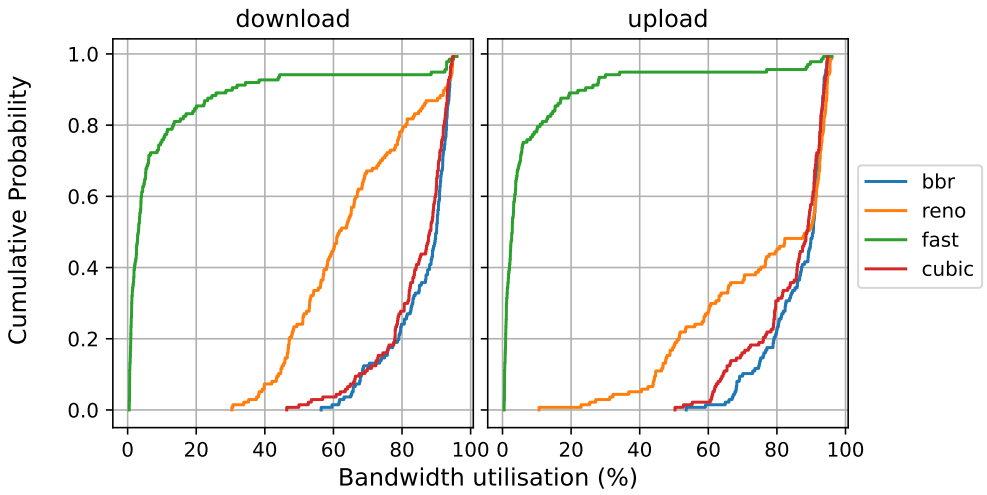


Figure 6.3: CDF of the bandwidth utilization for the different congestion control algorithms for the *bulk* tests

We can see that the loss-based algorithms (Cubic and New Reno) are the ones with the highest values. This is normal because they are only reacting to losses that are coming from filled buffers. Conversely, the delay-based algorithm (Fast) has the lowest results. Furthermore, the different metrics are very close. This indicates that the latency didn't vary much during the transfers. Finally, The hybrid CCA is having the same problems as the loss-based ones but with a lower scale. This is expected because the hybrid is taken into account both losses and latency.

6.2.2 Limited

For limited tests, as mentioned before, we first estimate the bandwidth using bulk BBR probes. As shown in figure 6.3, BBR is close to reality so we can use it to estimate the bandwidth. But then we want to know if the custom CCA, limiting to a fixed bit rate, is close to the expected values.

The figure 6.5a shows the bandwidth accuracy of the different probes. The *bandwidth accuracy* is the percentage of real bandwidth used by the different fixed bandwidth tests. The color of the points is related to the test done. For example, the red points are from the tests taking 90% of the estimated bandwidth. We can see that in general, we are a bit below the target. The main issue can be that taking 110% of the measured bandwidth isn't enough.

Then for *normal* CCA, the latency can impact the behavior. In the limited rate CCA we don't look at it so it shouldn't impact the bandwidth. The figure 6.5b is

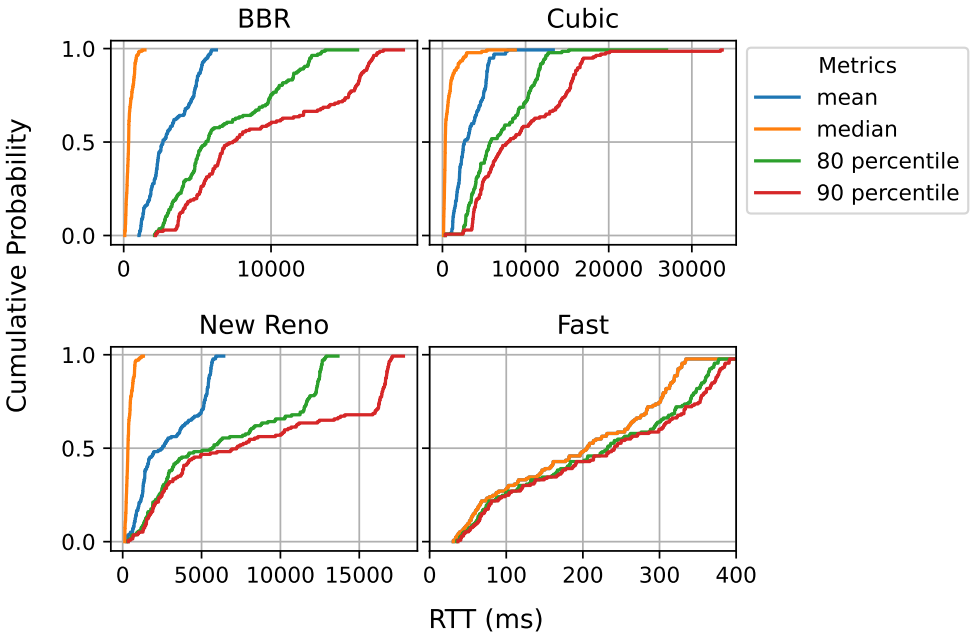
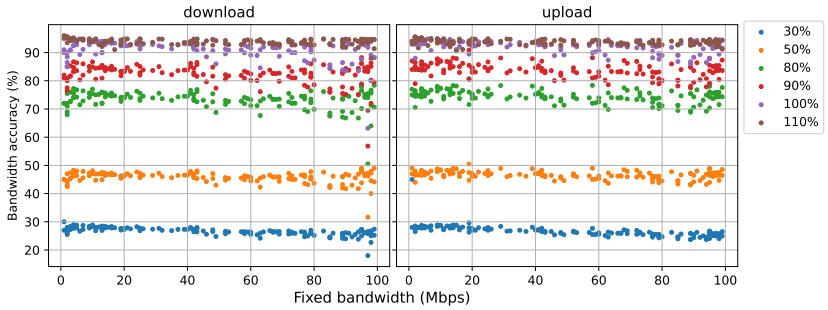
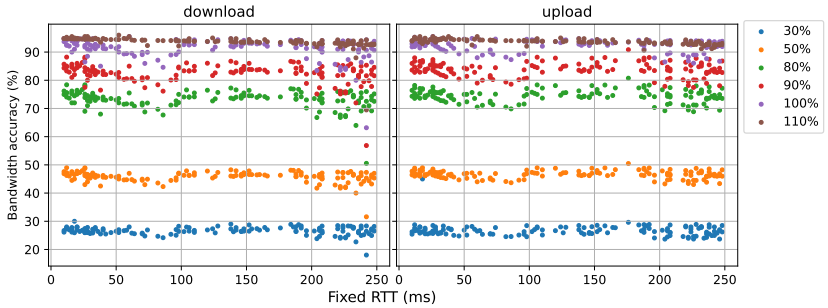


Figure 6.4: CDF of the latency for the different congestion control algorithms for the *bulk* tests



(a) Varying the fixed bandwidth



(b) Varying the fixed RTT

Figure 6.5: Percentage of the used bandwidth for *limited tests*, taking X% of the estimated bandwidth

showing that the latency doesn't influence the bandwidth taken. The graph is the same as 6.5b, except we are varying the fixed RTT instead of the bandwidth.

Finally, we want to know if taking 110% of the measured bandwidth will lead to saturation. The figure 6.6 is showing that. The different subplots are the CDF of the measured RTT for several metrics and for the different tests taking each time a particular percentage of the available bandwidth.

We can see that for tests taking less bandwidth than measured, the graph are looking the same. The median is close to the high percentiles and the most extreme value is around 400ms. Thus we can see that those are not varying a lot.

However for the tests taking the measured bandwidth or even more, the latency increase by a lot. All metrics are forming almost vertical lines which indicates that the test didn't vary from one to another. So the bottleneck doesn't come from the fixed latency but from the buffer size.

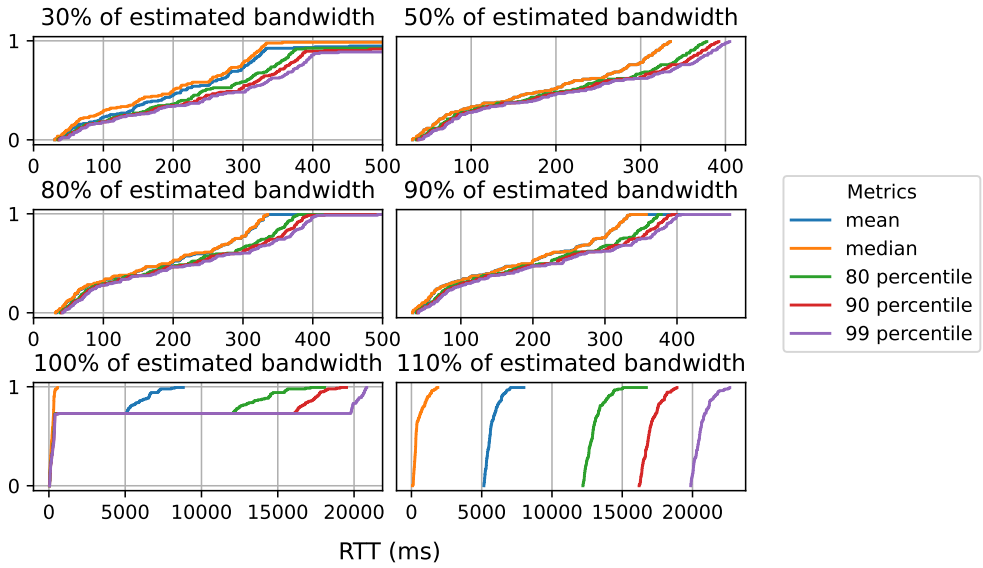


Figure 6.6: CDF of different RTT metrics for *limited tests* taking a fixed percentage of measured bandwidth

6.3 LibreSpeed

6.3.1 Estimating bandwidth

The first thing we want to know is if the bandwidth is well-measured. The figure 6.7 shows us that. The x-axis represents the bandwidth we fixed using `tc` and the y-axis, the bandwidth measured by LibreSpeed. The *expected line* (in red) is the ideal case where the fixed bandwidth is equal to the measured bandwidth. The points are the measured bandwidth, so if a point is below the *expected line* then the point is underestimated. And above the *expected line*, the point will be overestimated. Finally, the color of the points is the fixed latency, the more the points tend to be yellow (bright) the more the latency is high.

In both ways, the points are forming a line. We can also see that those lines have the same origin as the red line but are growing slower. We can explain this by the fact that in section 5.2, we set the overhead compensation factor to 1 to only measure goodput. So we can confirm that we need to measure an overhead compensation factor.

However, in the uploading test (on the right), they are some values that do not follow the straight line. We can see that there are tests with high RTT. This can be explained by the fact that those are tests with a high RTT, greater than 100ms. As explained in section 5.2, we are doing 10 latency probes per second so a latency probe every 100ms. If the client has a RTT below 100ms the responsiveness probes

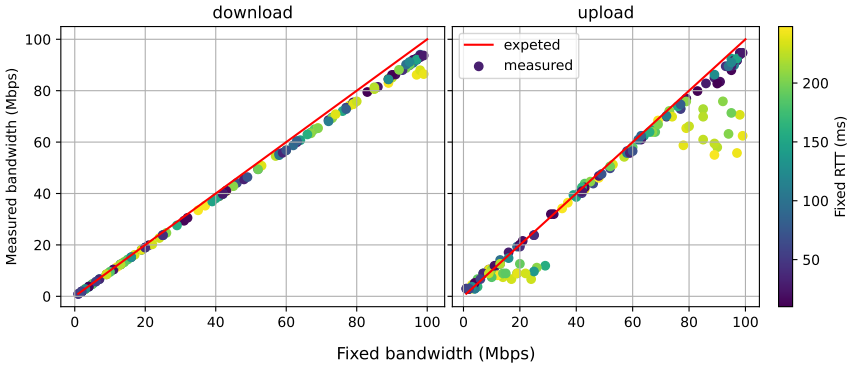


Figure 6.7: Comparison between fixed and measured bandwidth

are arriving before sending another one and since the packet is small this doesn't impact much the bandwidth. But in those extreme cases, the client is doing new probes even before the previous ones have arrived. And so, it fills the router buffer with latency probes. Thus the probes are taking a more significant part of the available bandwidth, and so the measured bandwidth is less than it should be. We had discussions with the RPM authors to maybe use an adaptative number of probes[5].

6.3.2 Computing overhead

To correct the measured bandwidth we have to multiply by an overhead compensation factor. To do that, we take all ratios between the expected and measured bandwidth and take the median of it. We first took the mean but since the mean is more impacted by outliers, we prefer to use the median.

For those tests, we get the value 1.051 for the compensation factor. We trust this value because it's close to the factor used by LibreSpeed for TCP which is 1.06 and we expected that QUIC overhead won't be too much different than TCP. The figure 6.8 shows the same measurement as figure 6.7 but by multiplying the overhead compensation factor by the measured bandwidth. We can see that the line formed by measurement points follows the red line, and so this means that the factor is well chosen.

[5] `https://github.com/network-quality/community/wiki/Meeting-Notes#march-1-2023-830am` on the discussion about the `draft-ietf-ippm-responsiveness`

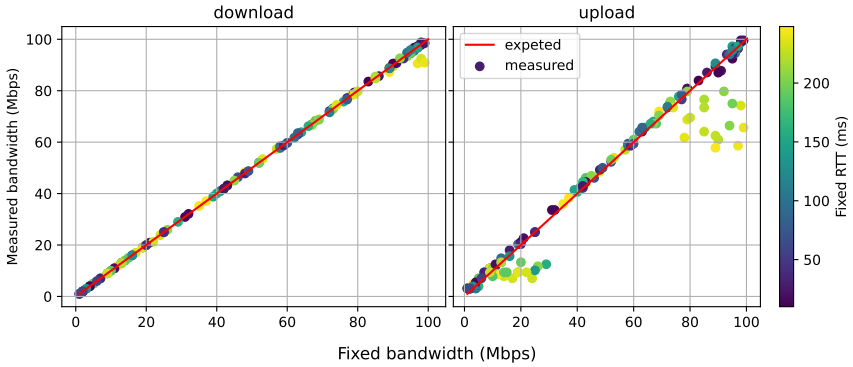


Figure 6.8: Comparaison between expected and measured bandwidth after correction

6.3.3 Estimating idle latency

Our tool focuses on measuring the latency, so it's very important to measure it accurately. The figure 6.9 shows the difference between the latency we fixed and the measured idle latency. As for figure 6.7, the x-axis is the fixed value by `tc` and the y-axis is the value measured by LibreSpeed, with the difference that we are no longer measuring bandwidth but latency. In the same idea as before, the red line is the perfect case where we measure exactly what we fixed.

We can see that the measured values are forming a straight line which seems parallel line to the red line. This indicates that the browser adds a small constant latency overhead. In our tests, this is around 10ms which was the value measured when not limiting latency. So since the difference is constant and is linked to the device/browser used by the client, we don't have to compensate for.

6.3.4 Latency increasing

After looking at idle latency, we are now taking a look at the loaded latency. The figure 6.10 shows that the latency doesn't increase a lot when we have a loaded network. This is an expected result because the router should be able to support the rate of the sender.

However, we see some results with high latency increase. Those are results with low fixed latency and a la low bandwidth and so a low Bandwidth-Delay Product (BDP). This phenomenon is better shown with the figure 6.11. On the x-axis, we indicate the BDP which is the available bandwidth times the RTT and on the y-axis we indicate the factor of increase. We can see that the latency increases when the BDP is lower than 50kB. This is normal behavior because the BDP is lower than the router congestion window size (which is 50kB) [46].

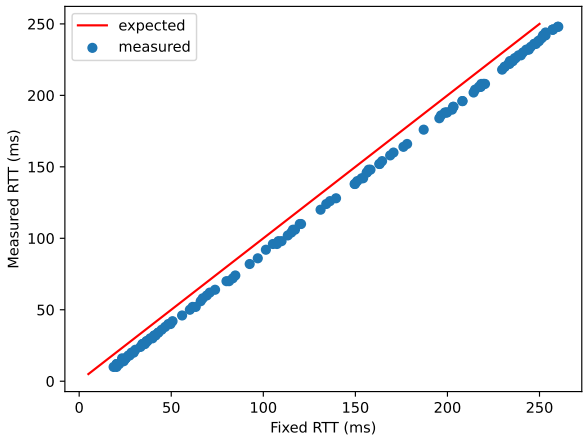


Figure 6.9: Factor of latency increase with bandwidth

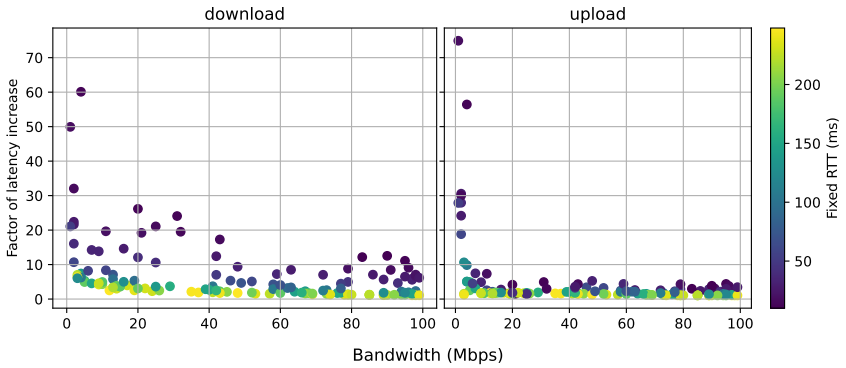


Figure 6.10: Comparaison between fixed and measured bandwidth

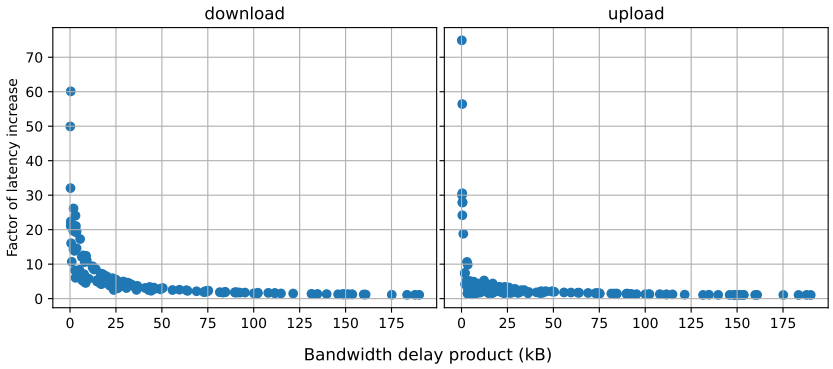
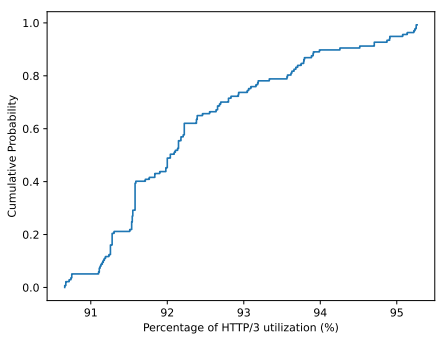
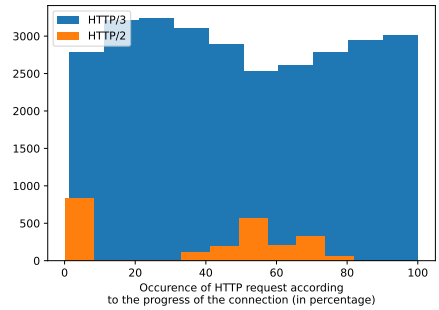


Figure 6.11: Comparaison between latency increase and bandwidth-delay product



(a) CDF of `HTTP/3` usage



(b) Distribution of `HTTP/2` and `HTTP/3` requests according to the progress of the connection

Figure 6.12: Usage of `HTTP/3` in LibreSpeed connections

6.3.5 QUIC usage

Finally, we want to know if the solution with the proxy explained in section 5.2 allows us to use `HTTP/3` connections. In figure 6.12a we are showing the CDF of the usage. For computing the percentage of `HTTP/3` utilization we are taking the number of such connections among the others. We can see that we have a huge utilization of `HTTP/3`. At the lowest, we had 90% which is already significant.

In the figure 6.12b we focus on when the `HTTP/2` connections appear. We can see that we have a spike at the start of the test then we have others in the middle of the test. Besides that, we can see that `HTTP/3` connections are more used all along the connections.

Summary

We saw that the different CCAs are acting as expected with our controlled environment and that BBR is using a huge amount of bandwidth for most cases. Using it we saw that for the fixed rate CCA, we were close to what we expected. And saw that by taking a rate above the actual capacity we introduced a lot of latency.

Then with LibreSpeed, we were able to spot a limitation of the RPM draft. But also to compute an overhead compensation factor that is close to the factor used for a stack using TCP.

Chapter 7

Live experiment

The controlled environment is good to validate our tools, but this doesn't reflect the internet behavior. So we deployed both tests in the wild. The LibreSpeed test is simple to launch, even for non-IT people.

Sharing the test

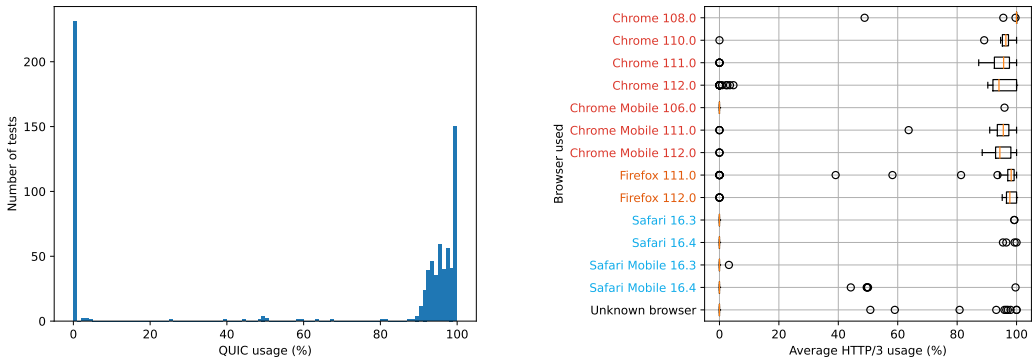
To have a consequent number of tests, we had to share the tool with as many people as possible. We share the results on several media: Twitter, Linkedin, etc. The authors of the RPM draft and other researchers also share the tool on those media. And finally, there was a mention of the tool in an IETF blog post[1].

Global results

We get a lot of results, about 750 tests done. Those tests have various locations, but most of which are in Belgium or close to the server. This is because of the author's location.

For deployment, we host the LibreSpeed backend as mentioned in section 5.3. This means using two servers. The frontend is less important so we deployed it in an OVH VPS which contains other services. The backend is deployed in a dedicated OVH VPS. This server uses Ubuntu 22.04 (LTS) with kernel 5.15.0-67-generic. It also has 500Mbps of bandwidth (download and upload), two virtual cores and 4Go of RAM. It's located in Gravelines (France).

[1] `https://www.ietf.org/blog/banishing-bufferbloat/`



(a) Distribution of QUIC usage

(b) Usage of QUIC by browsers

Figure 7.1: Quic usage

7.1 Browsers and QUIC usage

Since we want the test to use QUIC to better understand his behavior and have better logs. We can see in figure 7.1a that the QUIC usage is very on/off. We see that bins are located in the corner of the graph. This means either the browser uses no `HTTP/3` connections, or it uses almost only QUIC.

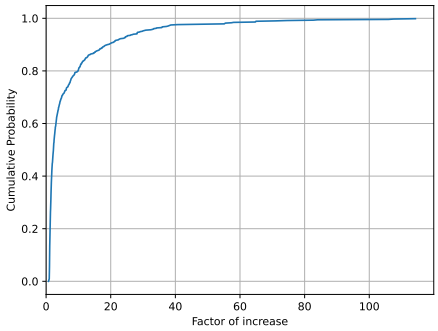
By looking at browser usage in figure 7.1b, we can enforce the idea of on/off usage of QUIC. We can also see that the latest version of Chrome and Firefox tend to use QUIC every time. This is because QUIC is activated by default in those while on older versions or Safari, you must activate an experimental feature[2]. We had also unknown browsers. Those are embedded browsers in some apps (Linkedin, Instagram, Twitter, etc).

7.2 Latency increase

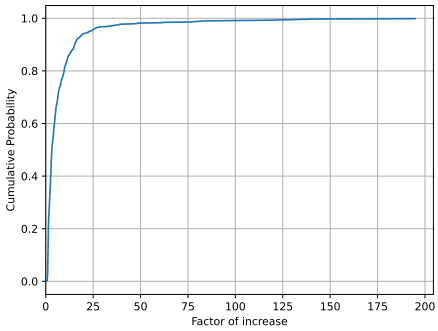
Since people have various internet subscriptions and are using different ways to connect (ethernet, Wi-Fi, cellular, etc), this is not very useful to aggregate global metrics, like the average bandwidth/latency, because the variance will be very high.

Instead, we aggregate relative metrics. In the figures 7.2a and 7.2b we are using the *factor of increase* which is the ratio between unloaded and loaded latency. If we have an unloaded latency of R_U and a loaded latency of R_L, both expressed in terms of RPM, the *factor of increase* is R_U/R_L. This ratio indicates to us how much the latency increase. We first thought about taking the difference but this will be proportional to the values. For example, if we have $R_U = 50$ RPM and $R_L = 5$

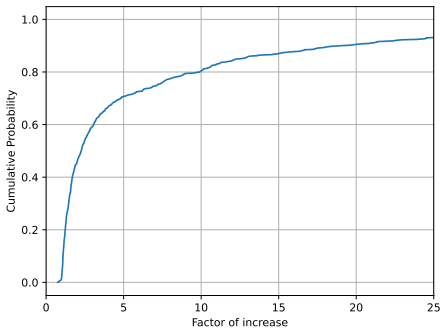
[2] `https://developer.apple.com/forums/thread/660516`



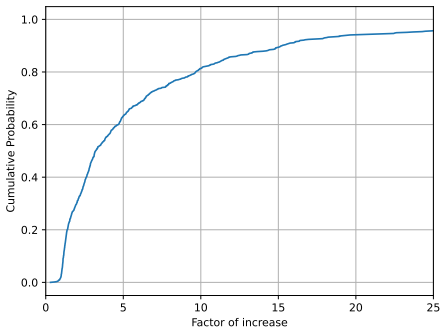
(a) CDF of the ratio between unloaded and loaded latency for download



(b) CDF of the ratio between unloaded and loaded latency for upload



(c) CDF of the ratio between unloaded and loaded latency for download zooming on factors below 25



(d) CDF of the difference between unloaded and loaded latency for upload zooming on factors below 25

Figure 7.2: Multiple CDF of relative latency measurement

RPM, the difference will be the same as for $R_U = 1050$ RPM and $R_L = 1005$ RPM. But in the first case, we have an increase of 10 while in the second case, the latency increase is less noticeable.

In figures 7.2c and 7.2d, we show the distribution of the *factor of increase* for download and upload as a CDF but with zooming on factors below 25 to have a better view. We can see that they both have a median increasing factor of around 3. And we can also see that for 30% of the tests, the latency increase by a factor bigger than 5. The global form looks more like an exponential distribution, most of the values are *small* and there are some outliers.

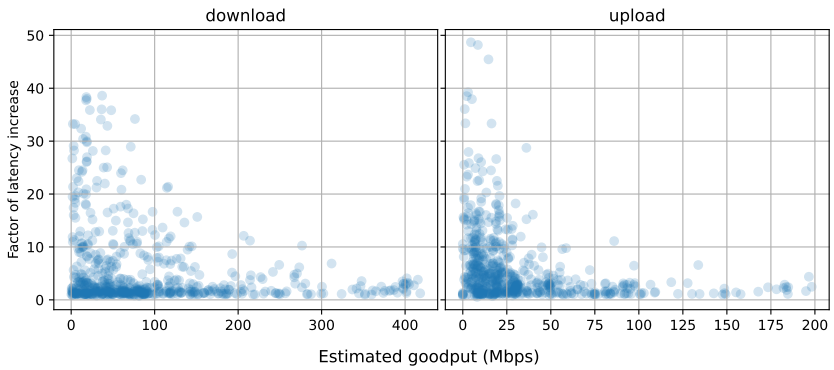


Figure 7.3: Factor of latency increase with goodput (without outlier)

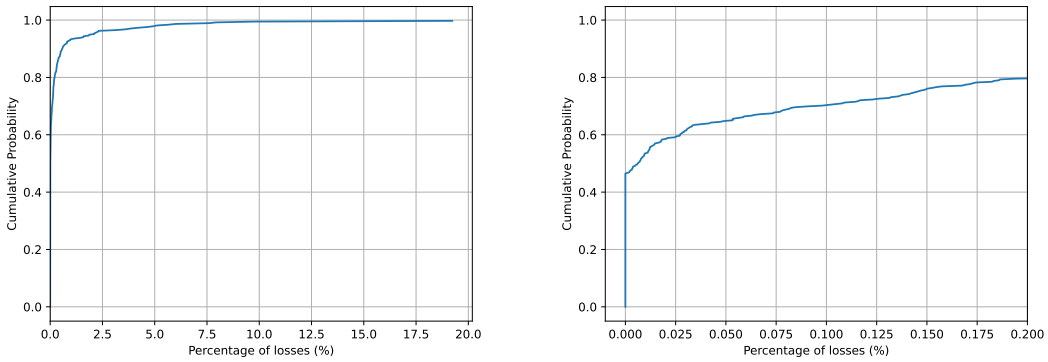
7.3 Bandwidth and latency

Since ISPs only give metrics on bandwidth we compare it to the latency increase. This is shown in figure 7.3. The x-axis indicates the measured bandwidth and the y-axis indicates the factor of latency increase. The points are drawn with an opacity of 20%, allowing us to see where the clusters are. To improve the view of the results, we removed outliers who have a Z-score above 3 [16].

We can see that higher bandwidths have fewer problems with increasing their latency. And in the other case, with lower bandwidth, under 100Mbps for download and 25Mbps for upload, we have a consequent number of tests with a high latency increase. This behavior is normal, with a better subscription for bandwidth came better latency [32].

Server bottleneck

However, the cases with high bandwidth are not relevant because the network might be saturated on the server side. For example, for one test with a client having a bandwidth of 1Gbps, we get 475.87Mbps in download and 455.55Mbps in upload. Since the server is limited to 500Mbps those values seem normal. Furthermore, if we put the estimated overhead compensation factor, we get results of 498 and 477Mbps which are close to the situation. The latency increasing factor is 1.32 for download and 0.92 for upload. Thus we can see that the client didn't saturate its connection.



(a) (b) zoomed for percentage below 0.2%

Figure 7.4: Distribution of losses

7.4 A focus on losses

Now we saw that there is a lot of latency. We will try to understand what is the cause. One possible cause can be the losses. To analyze the losses, we can focus on the percentage. However, our results showed that for most users the percentage of losses is under 1%. This is shown by figure 7.4a. The figure 7.4b zoom for values below 0.2%, and we can see that for half test the users didn't have losses. By looking at the outliers, the highest loss rate reaches 19.22%, then the next are below 10% of losses

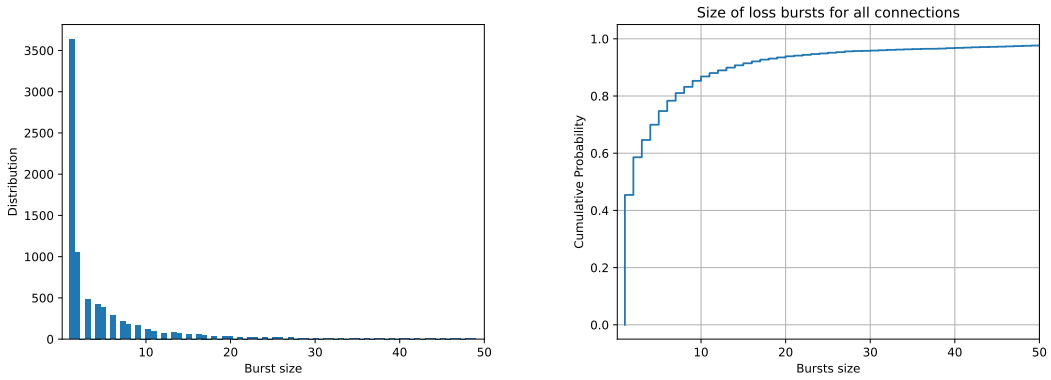
7.4.1 Burst of losses

To better understand the losses, we focus on the burst losses. By looking at the `qlog`s we can see the loss events with their timestamp and the packet number loss. So we can look at the burst's size and patterns, this is explained in the algorithm C. We define the burst size like this as the consecutive packet with a time difference lower than the addition of the RTT and the RTT variance.

Our results show that most of the bursts are burst of one packet (around 40%). For the other, more than 80% of the bursts are burst with less than 10 packets. This is explained in figure 7.5a and 7.5b, the x-axis is the size of the bursts and the y-axis indicates the distribution.

7.4.2 Spurious retransmission

The next thing we want to look at is to know if losses are causing spurious retransmissions. We define the rate as the number of spurious retransmission events



(a) PDF (b) CDF

Figure 7.5: Burst distributions both zoom for values below 50

among the number of loss events. By looking at the distribution, in figure 7.6a, we can see that this is an on/off phenomenon. By looking in detail at high spurious retransmission rates. We see that those are only for low loss rates, shown in figure 7.6b. The points are indicating for the loss rate (x-axis) and the rate of spurious retransmission among the loss events (y-axis). We use again point with an opacity of 20% to know where are the clusters. So we can expect that the spurious are only isolated events, which indicates a good global gestion of those.

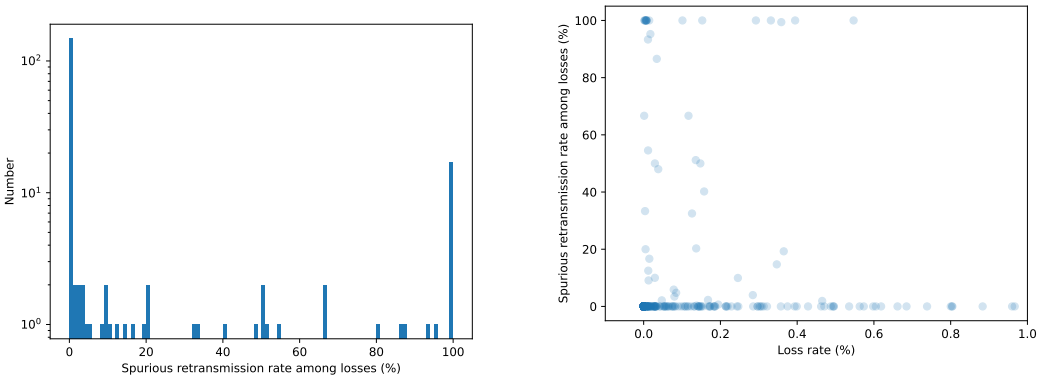
7.4.3 Influence of losses on latency increase

After detecting losses, we want to understand their effect on the latency increase. The figure 7.7 shows this by plotting points, with an opacity of 20%, who are having a particular loss rate (x-axis) and the associated factor of increase (y-axis). We can see that for download, if there are few losses the latency tends to increase. So this may mean that the Congestion Control Algorithm (CCA) is loss based. And so the algorithm waits for losses to adapt his rate. By looking at the server, we can confirm that it uses Cubic which is loss based. For the uploading way, the trend is less extreme. This is because browsers are maybe using other CCAs.

In figure 7.8 we zoom the graph to display loss rates below 1%. We can see that the tendency is still the same.

7.5 Limitation

We had the chance to collect several tests on a plane. By looking at the ratio of increase we get a maximal value of 3.2. We could thus say that the latency doesn't



(a) Spurious retransmission distribution

(b) Rate of spurious retransmission among the losses

Figure 7.6: Spurious retransmissions

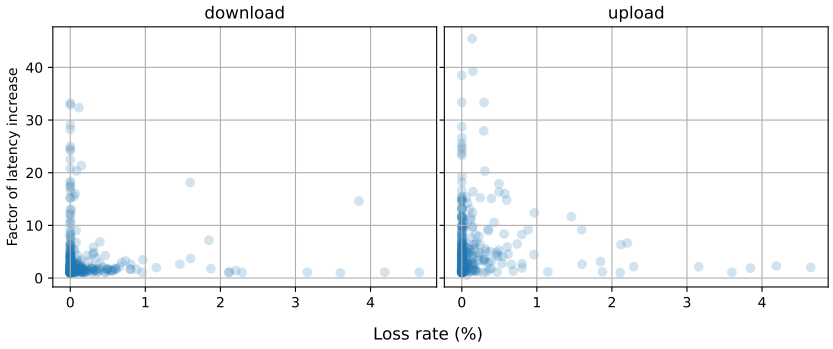


Figure 7.7: Factor of latency increase with loss rate (without outlier)

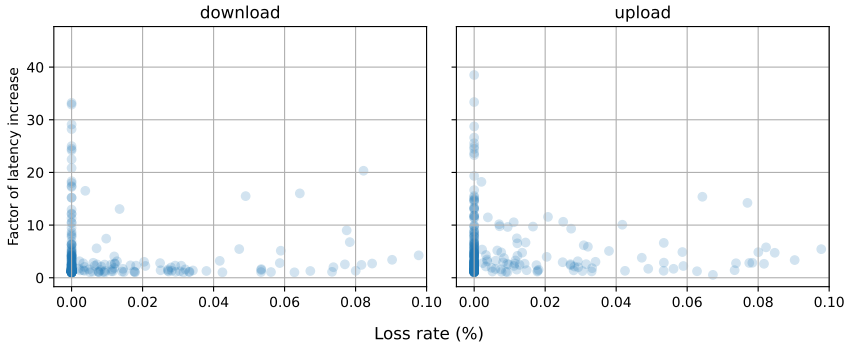


Figure 7.8: Factor of latency increase with loss rate (without outlier) by zooming for loss rate below 1%

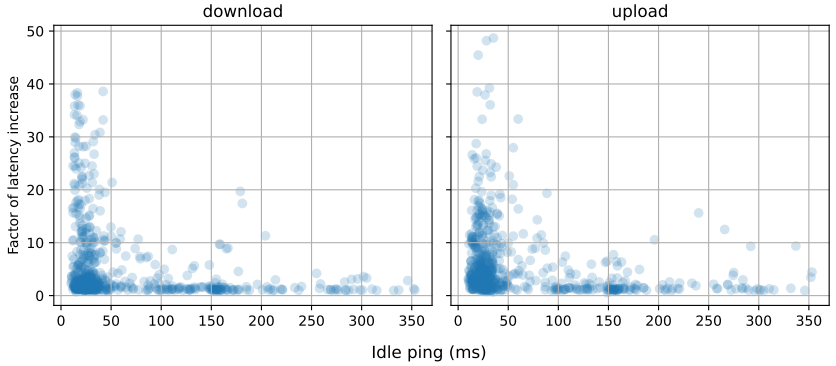


Figure 7.9: Factor of latency increase with idle latency (without outlier)

increase a lot and the test is looking good. But since the unloaded latency is about 70 RPM ($\approx$ 0.85s) which can already be seen as bad latency, the loaded latency doesn't increase by a lot, around 60 RPM for download ($\approx$ 1s) and around 30RPM for upload ($\approx$ 2s).

In figure 7.9, we plot the point having a particular idle latency (x-axis) and the associated factor of increase (y-axis) and plot them with an opacity of 20%. Also, we removed the outliers as for figure 7.3 using the z-score of 3. We can see that the most increases are coming from low idle latency and that for higher latency we have less increase.

Summary

In this chapter, we analyzed the live results from the LibreSpeed tool. We saw that browsers are using `HTTP/3` in a large proportion. Also, we saw that there is a significant increase for most of the users and that the losses are not the cause. Finally, we saw the limitation of the tool with a network having a huge idle latency.

Chapter 8

Conclusion

In this thesis, we were able to first define statistics about a QUIC connection. Then use this tool able to make a tool that can analyze the behavior of QUIC connections using different congestion control algorithms and with probes with different levels of fixed rates. Finally, we were able to make a tool usable using a web browser.

With the NesQUIC tool, we first found that BBR is very good at estimating the bandwidth, but also avoids a too-significant latency increase. Then from this estimation, we used the measured bandwidth to perform tests using a fixed-rate CCA. We saw that when using below the fixed bandwidth the latency didn't increase. But when going at the estimated rate or higher, the latency increase by a lot.

Then, with LibreSpeed, we proved a limitation on the RPM draft. But also we were able to compute an overhead compensation factor for `HTTP/3` connections which is close to the factor chosen by LibreSpeed authors for a network stack using TCP.

And finally, we saw that the latency increase is a real problem. And we were able to quantify this problem. We also found that losses may not be the cause of latency increase and even more, can give feedback to loss-based CCAs.

Contribution summary

With this work, we brought several Open-Source contributions on existing projects. The main contribution is the one made on LibreSpeed which allows the tool to follow the RPM methodology[1]. Then while working on this tool we were able to give feedback to the RPM authors about usage[2] but also the draft itself[3]. Finally,

[1] `https://github.com/librespeed/speedtest/pull/562`
[2] `https://github.com/network-quality/community/wiki/Meeting-Notes`
[3] `https://github.com/network-quality/draft-ietf-ippm-responsiveness/pull/77`

when searching for a proxy that can use `HTTP/3` connections and extract `qlog` files for them, we didn't find one. So we modify Caddy allowing us to extract such files[4].

All our contributions are freely available on GitHub.

- QUIC-info:
  `https://github.com/FelixGaudinMasterThesis/QUIC-info`
- NesQUIC client:
  `https://github.com/FelixGaudinMasterThesis/NesQUIC-Client`
- Caddy (modified):
  `https://github.com/FelixGaudin/caddy/tree/librespeed-RPM`
- LibreSpeed (modified):
  `https://github.com/FelixGaudin/speedtest-RPM/tree/master-thesis`
- The coordinator
  `https://github.com/FelixGaudinMasterThesis/coordinator`
- All our analysis scripts:
  `https://github.com/FelixGaudinMasterThesis/analysis`

Futur works

To conclude this work, we discuss the possible future works to do. The first focus can be to implement QUIC_info inside a QUIC implementation and have a way to be able to transfer/log those statistics to have a better understanding of network quality. Then another perspective can be to have a live experiment for NesQUIC as we had for LibreSpeed. This can be harder to spread because the tool needs a bit of IT knowledge to be launched. A solution can be to embed it in an Android application. Last but not least, we should also continue to work on LibreSpeed to find a way to avoid the issue where the bandwidth is not well measured because of client latency.

[4] `https://github.com/caddyserver/caddy/pull/5428`

Bibliography

[1] Mean round-trip times in the ARPANET. RFC 619, Mar. 1974. URL https://www.rfc-editor.org/info/rfc619.

[2] User Datagram Protocol. RFC 768, Aug. 1980. URL https://www.rfc-editor.org/info/rfc768.

[3] Internet Protocol. RFC 791, Sept. 1981. URL https://www.rfc-editor.org/info/rfc791.

[4] Transmission Control Protocol. RFC 793, Sept. 1981. URL https://www.rfc-editor.org/info/rfc793.

[5] On Packet Switches With Infinite Storage. RFC 970, Dec. 1985. URL https://www.rfc-editor.org/info/rfc970.

[6] D. Alam, M. Zaman, T. Farah, R. Rahman, and M. S. Hosain. Study of the dirty copy on write, a linux kernel memory allocation vulnerability. In *2017 International Conference on Consumer Electronics and Devices (ICCED)*, pages 40–45. IEEE, 2017.

[7] M. Allman and V. Paxson. On estimating end-to-end network path properties. *ACM SIGCOMM Computer Communication Review*, 29(4):263–274, 1999.

[8] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. *ACM SIGCOMM Computer Communication Review*, 34(4):281–292, 2004.

[9] M. Bishop. HTTP/3. RFC 9114, June 2022. URL https://www.rfc-editor.org/info/rfc9114.

[10] O. Bonaventure et al. *Computer Networking: Principles, Protocols and Practice*. 2021. URL https://www.computer-networking.info.

[11] C. Bormann and P. Hoffman. Concise binary object representation (cbor). Technical report, 2013.

[12] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *Proceedings of the conference on Communications architectures, protocols and applications*, pages 24–35, 1994.

[13] B. Briscoe, K. D. Schepper, M. Bagnulo, and G. White. Low Latency, Low Loss, and Scalable Throughput (L4S) Internet Service: Architecture. RFC 9330, Jan. 2023. URL `https://www.rfc-editor.org/info/rfc9330`.

[14] N. Cardwell, Y. Cheng, S. H. Yeganeh, I. Swett, and V. Jacobson. BBR Congestion Control. Internet-Draft draft-cardwell-iccrg-bbr-congestion-control-02, Internet Engineering Task Force, Mar. 2022. URL `https://datatracker.ietf.org/doc/draft-cardwell-iccrg-bbr-congestion-control/02/`. Work in Progress.

[15] V. Cerf and R. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on communications*, 22(5):637–648, 1974.

[16] A. E. Curtis, T. A. Smith, B. A. Ziganshin, and J. A. Elefteriades. The mystery of the z-score. *Aorta*, 4(04):124–130, 2016.

[17] J. Day and H. Zimmermann. The osi reference model. *Proceedings of the IEEE*, 71(12):1334–1340, 1983. doi: 10.1109/PROC.1983.12775.

[18] G. Dion, K. De Schepper, and O. Tilmans. Focusing on latency, not throughput, to provide a better internet experience and network quality. In *IAB Workshop on Measuring Network Quality for End-Users*, 2021.

[19] R. T. Fielding and J. Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, June 2014. URL `https://www.rfc-editor.org/info/rfc7231`.

[20] R. T. Fielding, H. Nielsen, J. Mogul, J. Gettys, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2068, Jan. 1997. URL `https://www.rfc-editor.org/info/rfc2068`.

[21] A. Gurtov, T. Henderson, and S. Floyd. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782, Apr. 2004. URL `https://www.rfc-editor.org/info/rfc3782`.

[22] W. Hardaker and O. Shapira. IAB Workshop Report: Measuring Network Quality for End-Users. RFC 9318, Oct. 2022. URL `https://www.rfc-editor.org/info/rfc9318`.

[23] T. Høiland-Jørgensen, C. A. Grazia, P. Hurtig, and A. Brunstrom. Flent: The flexible network tester. In *Proceedings of the 11th EAI International Conference on Performance Evaluation Methodologies and Tools*, pages 120–125, 2017.

[24] C.-H. Hsu and U. Kremer. Iperf: A framework for automatic construction of performance prediction models. In *Workshop on Profile and Feedback-Directed Compilation (PFDC), Paris, France*. Citeseer, 1998.

[25] T. Høiland-Jørgensen, P. McKenney, dave.taht@gmail.com, J. Gettys, and E. Dumazet. The Flow Queue CoDel Packet Scheduler and Active Queue Management Algorithm. RFC 8290, Jan. 2018. URL `https://www.rfc-editor.org/info/rfc8290`.

[26] J. Iyengar and I. Swett. QUIC Loss Detection and Congestion Control. RFC 9002, May 2021. URL `https://www.rfc-editor.org/info/rfc9002`.

[27] J. Iyengar and M. Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021. URL `https://www.rfc-editor.org/info/rfc9000`.

[28] C. Jin, D. X. Wei, and S. H. Low. Fast tcp: motivation, architecture, algorithms, performance. In *IEEE INFOCOM 2004*, volume 4, pages 2490–2501. IEEE, 2004.

[29] L. Kleinrock and H. Opderbeck. Throughput in the arpanet–protocols and measurement. *Communications, IEEE Transactions on*, 25:95 – 104, 02 1977. doi: 10.1109/TCOM.1977.1093724.

[30] D. Lee, B. E. Carpenter, and N. Brownlee. Observations of udp to tcp ratio and port numbers. In *2010 Fifth International Conference on Internet Monitoring and Protection*, pages 99–104. IEEE, 2010.

[31] R. Ludwig and R. H. Katz. The eifel algorithm: making tcp robust against spurious retransmissions. *ACM SIGCOMM Computer Communication Review*, 30(1):30–36, 2000.

[32] K. MacMillan and N. Feamster. Beyond speed test: Measuring latency under load across different speed tiers.

[33] R. Marx, M. Piraux, P. Quax, and W. Lamotte. Debugging quic and http/3 with qlog and qvis. In *Proceedings of the Applied Networking Research Workshop*, pages 58–66, 2020.

[34] R. Marx, L. Niccolini, M. Seemann, and L. Pardue. Main logging schema for qlog. Internet-Draft draft-ietf-quic-qlog-main-schema-05, Internet Engineering Task Force, Feb. 2023. URL https://datatracker.ietf.org/doc/draft-ietf-quic-qlog-main-schema/05/. Work in Progress.

[35] R. Marx, L. Niccolini, M. Seemann, and L. Pardue. QUIC event definitions for qlog. Internet-Draft draft-ietf-quic-qlog-quic-events-04, Internet Engineering Task Force, Feb. 2023. URL https://datatracker.ietf.org/doc/draft-ietf-quic-qlog-quic-events/04/. Work in Progress.

[36] Measurement Lab. The M-Lab NDT data set. https://measurementlab.net/tests/ndt.

[37] M. J. Muuss. The story of the ping program. URL https://ftp.arl.army.mil/~mike/ping.html.

[38] H. Nielsen, R. T. Fielding, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945, May 1996. URL https://www.rfc-editor.org/info/rfc1945.

[39] C. Paasch, R. Meyer, S. Cheshire, and W. Hawkins. Responsiveness under Working Conditions. Internet-Draft draft-ietf-ippm-responsiveness-02, Internet Engineering Task Force, Mar. 2023. URL https://datatracker.ietf.org/doc/draft-ietf-ippm-responsiveness/02/. Work in Progress.

[40] R. Pan, P. Natarajan, F. Baker, and G. White. Proportional Integral Controller Enhanced (PIE): A Lightweight Control Scheme to Address the Bufferbloat Problem. RFC 8033, Feb. 2017. URL https://www.rfc-editor.org/info/rfc8033.

[41] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, and R. Scheffenegger. CUBIC for Fast Long-Distance Networks. RFC 8312, Feb. 2018. URL https://www.rfc-editor.org/info/rfc8312.

[42] J. Roskind. Quick udp internet connections: Multiplexed stream transport over udp. *Adresse: https://docs. google. com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit (besucht am 05. 07. 2017)*, 2012.

[43] J. Santiago, M. Claeys-Bruno, and M. Sergent. Construction of space-filling designs using wsp algorithm for high dimensional spaces. *Chemometrics and Intelligent Laboratory Systems*, 113:26–31, 2012.

[44] D. Taht. Rfc: Realtime response under load (rrul) test specification. *GMANE. September*, 2012.

[45] B. Turkovic, F. A. Kuipers, and S. Uhlig. Fifty shades of congestion control: A performance and interactions evaluation. *arXiv preprint arXiv:1903.03852*, 2019.

[46] C. Villamizar and C. Song. High performance tcp in ansnet. *ACM SIGCOMM Computer Communication Review*, 24(5):45–60, 1994.

Appendices

Appendix A

Internet speedtests

Website	Capacity	Idle latency	Working latency	Open-Source ?
librespeed.org	Yes	Yes	No	Yes
speed.measurementlab.net	Yes	Yes	No	Yes
speedtest.net	Yes	Yes	Yes	No
speed.cloudflare.com	Yes	Yes	Yes	Yes
www.waveform.com/tools/bufferbloat	Yes	Yes	Yes	No
bandwidthplace.com	Yes	Yes	No	No
speedof.me	Yes	Yes	No	No
netmeter.eu	Yes	Yes	No	No
nperf.com	Yes	Yes	No	No
speedsmart.net	Yes	Yes	No	No
testmy.net	Yes	Yes	No	No
speedcheck.org	Yes	Yes	No	No
fast.com	Yes	Yes	Yes	No
speed.googlefiber.net	Yes	Yes	No	No
wifi.wtf	Yes	Yes	No	No
www.att.com/support/speedtest	Yes	Yes	No	No
gfblip.appspot.com	No	Yes	No	Yes
speedtest.gcore.com/	Yes	Yes	No	No
openspeedtest.com	Yes	Yes	No	Yes
www.dslreports.com/speedtest?httpsok=0	Yes	Yes	Yes	No
pingtest.cse.engin.umich.edu/ping	No	Yes	No	No

Table A.1: Exhaustive list of internet speed tests

Appendix B

TCP_info vs QUIC_info

Linux Kernel	**OpenBSD**	**Apple**	**Windows**	**QUIC_info**
tcpi_rto	tcpi_rto	tcpi_rto	hardcoded	RTO
tcpi_ato	___tcpi_ato	hardcoded	hardcoded	ATO
tcpi_snd_mss	tcpi_snd_mss	tcpi_snd_mss	Mss	Snd-MSS
tcpi_rcv_mss	tcpi_rcv_mss	tcpi_rcv_mss	Mss	Rcv-MSS
tcpi_lost	___tcpi_lost	/	TimeoutEpisodes	Lost
tcpi_retrans	___tcpi_retrans	tcpi_txretransmitpackets	BytesRetrans	Retrans
tcpi_last_data_sent	tcpi_last_data_sent	/	/	Last_data_sent
tcpi_last_ack_sent	tcpi_last_ack_sent	/	/	Last_ack_sent
tcpi_last_data_recv	tcpi_last_data_recv	/	/	Last_data_recv
tcpi_last_ack_recv	tcpi_last_ack_recv	/	/	Last_ack_recv
tcpi_min_rtt	tcpi_rttmin	tcpi_rttbest	MinRttUs	Min-rtt
tcpi_rtt	tcpi_rtt	tcpi_srtt	RttUs	Smooth-rtt
tcpi_rttvar	tcpi_rttvar	tcpi_rttvar	/	Var-rtt
tcpi_pacing_rate	/	/	/	Pacing-rate
tcpi_max_pacing_rate	/	/	/	Max-pacing-rate
tcpi_delivery_rate	/	/	/	Delivery_rate
tcpi_bytes_sent	/	tcpi_txbytes	BytesOut	Bytes_sent
tcpi_bytes_received	/	tcpi_rxbytes	BytesIn	Bytes_recv
Not Applicable	Not Applicable	Not Applicable	Not Applicable	Frames-sent
Not Applicable	Not Applicable	Not Applicable	Not Applicable	Frames-recv
Not Applicable	Not Applicable	Not Applicable	Not Applicable	Data-frames-sent
Not Applicable	Not Applicable	Not Applicable	Not Applicable	Data-frame-recvs
tcpi_snd_cwnd	tcpi_snd_cwnd	tcpi_snd_cwnd	SndLimBytesCwnd	Cwnd
tcpi_segs_out	/	/	/	Dgrams-sent
tcpi_segs_in	/	/	/	Dgrams-recv
tcpi_data_segs_in	/	/	/	Data-dgrams-sent
tcpi_data_segs_out	/	/	/	Data-datagram-recv
/	/	/	/	Data-sent
/	/	/	/	Data-recv
/	tcpi_snd_nxt	tcpi_snd_nxt	/	Next-send-pkt-nbr
/	tcpi_rcv_nxt	tcpi_rcv_nxt	/	Next-recv-pkt-nbr
/	/	/	TimestampsEnabled	Duration
/	/	/	/	Spurious

Table B.1: Mapping of QUIC_info fields to the main TCP_info implementations

Appendix C

Bursts size algorithm

```
    events = get_loss_events()
    srtt = quic_info.get_srtt()
    var = quic_info.get_varrtt()
    threshold = srtt + var

    burst_loss = []
    current_burst = 0
    for i in range(len(events) - 1):
        delta_time = events[i+1].timestamps - events[i].timestamps
        if delta_time <= threshold: # in the same burst
            current_burst += 1
        else:
            burst_loss.append(current_burst)
            current_burst = 1
    burst_loss.append(current_burst)

```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**