Politecnico di Torino

Politecnico di Torino
Master's Degree in Computer Engineering
July 2023

Implementation of an Inside RTT monitoring technique in QUIC

Supervisors
Prof. Sisto Riccardo
Prof. Marchetto Guido

Candidate
Trovato Fabrizio

Company Tutor
Telecom Italia
Dott. Nilo Massimo

Abstract

Passive network monitoring is a vital activity for network operators that want to guarantee a good level of quality of service to the users of the network, as it allows them to observe real user data and detect performance and health issues. One of the variables that mostly impacts the effectiveness of passive network monitoring is the wire image of the protocol under analysis, which in the case of QUIC is very limited. Furthermore, the Spin Bit defined in the QUIC specification is optional and it is rarely implemented in existing QUIC libraries, making it unusable since it must be supported and agreed upon by both endpoints.

This thesis aims to present and implement the Inside RTT monitoring technique, in order to provide an alternative that can be deployed in client applications to solve the issues of adoption and cooperation that render the traditional Spin Bit unusable in most cases. The idea of this technique is to rely on a client side estimation of the RTT and to use it to unilaterally generate a square wave that allows a passive observer to measure the RTT with accuracy.

Using the Inside RTT monitoring technique, a passive observer is therefore immediately able to measure a RTT estimation provided by the client, where previously it would not be able to measure a value at all. The results obtained when testing with the Chromium browser highlight the fact that as long as the client can compute an accurate estimation, the observer can measure the actual RTT between the client and the server with a few milliseconds of error. The results also show that Inside RTT monitoring does not solve all the issues of the traditional Spin Bit, but since these problems are not introduced by the technique itself and they are not as crucial as the lack of adoption, the value of the implementation is still considerable and worth the upgrade over existing solutions.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Objectives

For network operators it has been historically important to perform passive network monitoring, which consists in capturing and analyzing real user traffic in order to understand the level of performance and health of the network. It can therefore be problematic to conduct this activity in the presence of network protocols that produce mostly encrypted traffic: this issue has become particularly relevant with the advent of HTTP/3, since QUIC is set to replace TCP as the transport layer protocol used during the exchanges between client and server. In fact, while it provides a faster and more secure mechanism for structured communications than TCP, the QUIC protocol has a very limited wire image, meaning that the amount of observable plaintext information in packets is very small. Furthermore, the protocol supports a bit in the header of the packets called Spin Bit, which is defined in the QUIC standard [1] and which can be used upon agreement and cooperation between client and server to generate a square wave that allows a passive observer to measure a Round-trip time estimation: unfortunately this bit is optional and it is not often adopted in QUIC libraries, so it is usually not included in QUIC headers. This lack of adoption makes the effectiveness of the Spin Bit questionable, as a lack of support for the Spin Bit from one of the endpoints is enough to render the technique unusable. In fact, depending on the library used by the server, a client supporting the Spin Bit might not be able to use it, as the bit might not be reflected by the server. With HTTP/3 used by 25.5% of all websites [2], the inability to perform passive network monitoring of QUIC traffic is an increasing concern

for Internet Service Providers, as even simple metrics like RTT estimations are not easy to obtain.
To address this problem, this thesis aims to provide the implementation of an Inside RTT monitoring technique that allows a passive observer to monitor the performance of the QUIC connections present in the network, by relying on an internal estimation of the RTT computed by the client application, which is then used by the client itself to generate a square wave by means of the Spin Bit in the header of QUIC packets. This mechanism aims to solve the biggest limitations of the Spin Bit by allowing the client to generate the square wave without relying on the agreement and the cooperation between itself and the server, as the only requirement is that the client supports the Inside RTT monitoring technique inside its QUIC library. In particular, in this thesis the objective is to implement the algorithm inside the QUICHE library developed by Google, which is used inside the Chromium browser.

1.2 Thesis structure

In this section I will give a brief overview of the organization of this thesis, describing the content of each chapter that composes it.
In particular:

- Chapter 2 presents the theoretic background, including the most relevant network protocols for this thesis, an introduction to the Chromium browser, and an overview of passive network monitoring techniques and tools.
- Chapter 3 describes the Inside RTT monitoring technique, presenting the algorithm with its main strengths and weaknesses. Additionally, in the same chapter I will also discuss how I implemented the technique in the QUICHE library.
- Chapter 4 describes the testing environment used to test the implementation and the tools developed to support the testing phase.
- Chapter 5 contains the description of the tests I performed and it presents the results of this thesis, with an in depth evaluation of the Inside RTT monitoring technique.
- Chapter 6 contains the conclusions of this thesis and some insights for future research.

Chapter 2

Background

In this chapter I will analyze the main network protocols involved in modern connections, the tools that I used during the course of this thesis, and I will also provide an overview of passive network monitoring.
I will start by presenting the history and the evolution of the HTTP protocol: in particular I will discuss how HTTP/3 differs from the previous major versions of HTTP.
In the second section of the chapter I will describe the QUIC protocol, with a focus on the features and the specifics that are most relevant for this thesis: I am going to discuss the wire image of the protocol and its headers, then I will end the section by presenting the Spin Bit.
In the third section I will present the Chromium open-source browser, by describing its build environment and its integration of the QUIC protocol.
Finally, in the fourth and last section of the chapter I will present some network monitoring techniques and tools that are relevant to this thesis: the focal point will be passive network monitoring.

2.1 HTTP/3

The Hypertext Transfer Protocol (HTTP) was originally released in the 90s, and since then it has been the foundation of data exchange on the World Wide Web. It is a *client-server* protocol, and as such we have two parties involved: a user agent - usually a Web browser - that initiates requests, and a server that receives the requests, handles them and then provides responses [3].
During the last three decades multiple versions of HTTP have been released,

each bringing substantial improvements in terms of performance and efficiency. However, HTTP/3 is the first major version that changes the transport layer used by the protocol: while all the previous versions rely on TCP, HTTP/3 replaces it with a combination of UDP and QUIC. The goal is to solve Head of Line Blocking [4], using the per-stream retransmission and flow control mechanisms provided by QUIC. Additionally, by combining QUIC with TLS 1.3, HTTP/3 guarantees a higher level of security than previous HTTP versions, as unlike TCP the QUIC protocol provides encryption at the transport layer by combining the TLS and the QUIC handshake into a single operation that also helps reducing latency.
As of today, HTTP/3 is used by 25.5% of all websites [2] and it is supported by major browsers including Chrome, Edge, Firefox and Safari [5].

2.2 QUIC

QUIC is a UDP-based transport layer protocol, which was designed in an effort to replace TCP with a faster and more secure mechanism for structured communication. Originally introduced by Google, it was then standardized in RFC900 [1] and it served as the base for HTTP/3.
A QUIC **connection** is always negotiated through a chryptographic handshake, and it is identified by a set of associated Connection IDs: these IDs allow to deliver QUIC traffic to the right endpoint without the need to setup a new connection even when the UDP or the IP coordinates change, like in the case of NAT rebinding; by using Connection IDs, the protocol also enables port reuse. The initial Connection ID is transmitted in the Source Connection ID field of the header, while additional ones are negotiated using specific frames.
To avoid amplification attacks via source address spoofing, QUIC performs client validation by verifying that the user-agent can receive traffic at its claimed source address. During this phase, client tokens are also provided, allowing to reduce the time required by future validations.
On connection migration, QUIC performs path validation by sending a path challenge that must be solved at the client's claimed source address: once the validation phase is over, new Connection IDs will be negotiated in both directions, to avoid correlation of flows. For this reason the Connection IDs should never be reused, and internal congestion and RTT metrics of the protocol should also be reset.

QUIC relies on the stream abstraction, which is an ordered byte stream that undergoes a flow control mechanism, allowing receivers to limit the amount of data sent to their buffers, both per-stream and globally. As previously mentioned, the per-stream mechanism works better than TCP when paired with recent HTTP versions, allowing substantial performance improvements and independent retransmission. Streams generate frames that are then squeezed into **packets**, used by QUIC endpoints to communicate; packets are carried in UDP datagrams, and a single UDP datagram can contain one or more packets, as long as they belong to the same connection.
When it receives a QUIC packet, the endpoint must bind it to a connection by checking its Destination Connection ID in the header. As specified in RFC 9001 [6], QUIC packets have confidentiality and integrity thanks to the TLS 1.3 security layer, so the endpoint must then decrypt the packet, enqueue it and if needed - depending on the type of the frame - send an ACK frame.
The QUIC protocol implements a flow control mechanism that allows endpoints to control the amount of data sent to their buffers: during the handshake the endpoints can negotiate a limit of bytes that can be sent in a stream, or a maximum number of concurrent streams that a peer can open towards them. While this mechanism allows an endpoint to avoid being overwhelmed, it might impact performance as the network could work below peak capacity. Furthermore, re-negotiating the limit too often could cause significant overhead so it is not recommended. In case of loss, packets are not retransmitted as a whole, but instead new packets carrying exclusively the lost frames are generated and sent; these packets take a higher priority than new data, as retransmission is always prioritized.
A final note should be made about the flexibility provided by the QUIC protocol, when compared to TCP: while TCP implementations are tied to the operating system and the kernel, QUIC libraries are included into applications, allowing for easier development in the user space.

2.2.1 Headers

QUIC packets can have either a Long Header or a Short Header. To identify the two, it is possible to look at the most significant bit of the header, formerly known as the Header Form: if its value is 1 it is a Long Header, if its value is 0 it is a Short Header.
Packets with the Long Header are sent prior to the 1-RTT key establishment,

meaning before the TLS key exchange occurred: other than Initial packets and Handshake packets used to setup the QUIC connection, during this phase it is possible to exchange early data by using 0-RTT data packets, meaning packets protected exclusively with pre-shared TLS keys. This allows to reduce the connection setup delay, but it is important to keep in mind that data shared with 0-RTT packets is not protected against replay attacks, so it could be accepted or refused by other endpoints depending on their configuration. Instead, packets with the Short header are 1-RTT encrypted, meaning that they are sent after the QUIC version and the TLS keys have been negotiated.
In the next paragraphs I am going to discuss both headers and their fields.

Long Header

Used by Initial, Handshake, 0-RTT and Retry packets, its role is mainly to support the QUIC version negotiation between client and server. A packet with a Long Header will include the following fields:

Table 2.1: QUIC Long Header.

Name	**Length (bit)**
Header Form	1
Fixed Bit	1
Long Packet Type	2
Type-Specific Bits	4
Version	32
Destination Connection ID Length	8
Destination Connection ID	0..160
Source Connection ID Length	8
Source Connection ID	0..160
Type-Specific payload	..

Short Header

Used exclusively by packets of 1-RTT type, so as explained in this subsection only by packets exchanged after QUIC version and TLS keys have been negotiated. A packet with a Short Header will include the following fields:

Table 2.2: QUIC Short Header.

Name	**Length (bit)**
Header Form	1
Fixed Bit	1
Spin Bit	1
Reserved Bits	2
Key Phase	1
Packet Number length	2
Destination Connection ID	0..160
Packet Number	8..32
Packet payload	8..

The Packet Number is encrypted with the header key so unlike TCP it is not in clear. It is important to notice how Short Headers are the only ones that include the optional cleartext **Spin Bit**, which can be used for latency monitoring; the other Reserved Bits are instead protected with the header key.

2.2.2 Encryption and wire image

The main role of TLS 1.3 is to support QUIC during the handshake phase, in particular:

- Client and server exchange Initial packets, protected with keys derived from the Destination Connection ID of the client and a salt. This key is not secret.
- While exchanging Initial packets, client and server also exchange TLS Hello messages. If pre-shared TLS keys are available, 0-RTT packets can be used to exchange early data with no replay protection, reducing the setup time of the connection.
- Once the 1-RTT handshake is completed, the TLS key exchange will be over and QUIC will be able to guarantee confidentiality and integrity of application data.
- The headers of the packets are also protected, in particular Reserved Bits and Packet Number are always encrypted.

While the cooperation between QUIC and TLS 1.3 allows to reduce the overhead associated with the handshake phase and it increases both security and privacy, it also has major implications for Internet Service Providers and network operators: unlike TCP, the headers are not in cleartext, so it is not possible for a passive observer to evaluate network performance and latency by means of sequence numbers. Furthermore, acknowledgments are carried inside the encrypted payload.
With such a reduced wire image, the QUIC protocol gives network operators a limited amount of available data to analyze. In particular:

- QUIC version, Destination Connection ID and Fixed Bit are always observable. It should be noted that, if the implementation guidelines are respected, it is not possible to perform traffic correlation using Connection IDs as they should always be reset after each connection migration and they should never be reused.

- Source Connection ID, Header Type and Tokens are observable for packets that use the Long Header, so exclusively for the ones exchanged before the 1-RTT key establishment.

- The Spin Bit is observable for packets that use the Short Header, so packets that are 1-RTT encrypted. The Spin Bit was purposely introduced to allow latency measurements so it is in cleartext by design, however its use is optional and it should be agreed upon by both the client and the server.

- If a network operator acting as a passive observer along the path between two endpoints can recognize a handshake, it can recognize who is the client and who is the server. Additionally, the observer can measure the handshake RTT by looking at Hello packets and QUIC handshake messages.

This means that the amount of available information is small, especially after the handshake is completed and for packets that use the Short Header. In fact, a network operator acting as a passive observer cannot analyze:

- The payload or any part of the header that is encrypted, depending on the header type. This also applies to packets encrypted with the initial secret, unless we can derive the key from the Destination Connection ID, as previously explained.

- Acknowledgments, since they travel inside encrypted QUIC payloads and there is no way to distinguish them from the rest of the QUIC traffic.
- A connection teardown, since QUIC does not have an exposed teardown mechanism. The only option for a passive observer is to associate the end of a traffic flow with the end of the connection.

2.2.3 Spin Bit

The Spin Bit is an optional bit included in the Short Header of QUIC packets, that allows an on-path observer to monitor the network latency of a QUIC connection, by estimating the RTT. As briefly discussed in the previous subsections, it is unencrypted by design and its up to each endpoint to unilaterally decide whether it should be enabled or not.
In fact, the Spin Bit requires the cooperation of the client and the server:

- Both endpoints must maintain a Spin value for each connection.
- Both endpoints must remember the highest Packet Number seen for each connection.
- Whenever it receives a 1-RTT packet with a higher Packet Number than the highest seen yet on that same connection, the **server** reflects the Spin Bit: it updates its Spin value to the Spin Bit of the received packet, and it uses it to set the Spin Bit of outgoing 1-RTT packets on that same connection.
- Whenever it receives a 1-RTT packet with a higher Packet Number than the highest seen yet on that same connection, the **client** flips the Spin Bit: it updates its Spin value to the opposite of the of Spin Bit of the received packet, and it uses it to set the Spin Bit of outgoing 1-RTT packets on that same connection.

Thanks to this mechanism the client and the server generate a **square wave** that allows a passive observer to measure latency: as described in RFC 9312 [7] and as shown in Figure 2.1, each Spin period of two Spin toggles corresponds to the end-to-end RTT, and the observer can compute it by calculating the difference between the edges of the Spin period. This kind of RTT measurement is valuable because it can give to an on-path observer an estimation of the application layer delay and the transport layer delay

experienced by the application, so it serves as a valuable metric to monitor the performance of QUIC traffic.

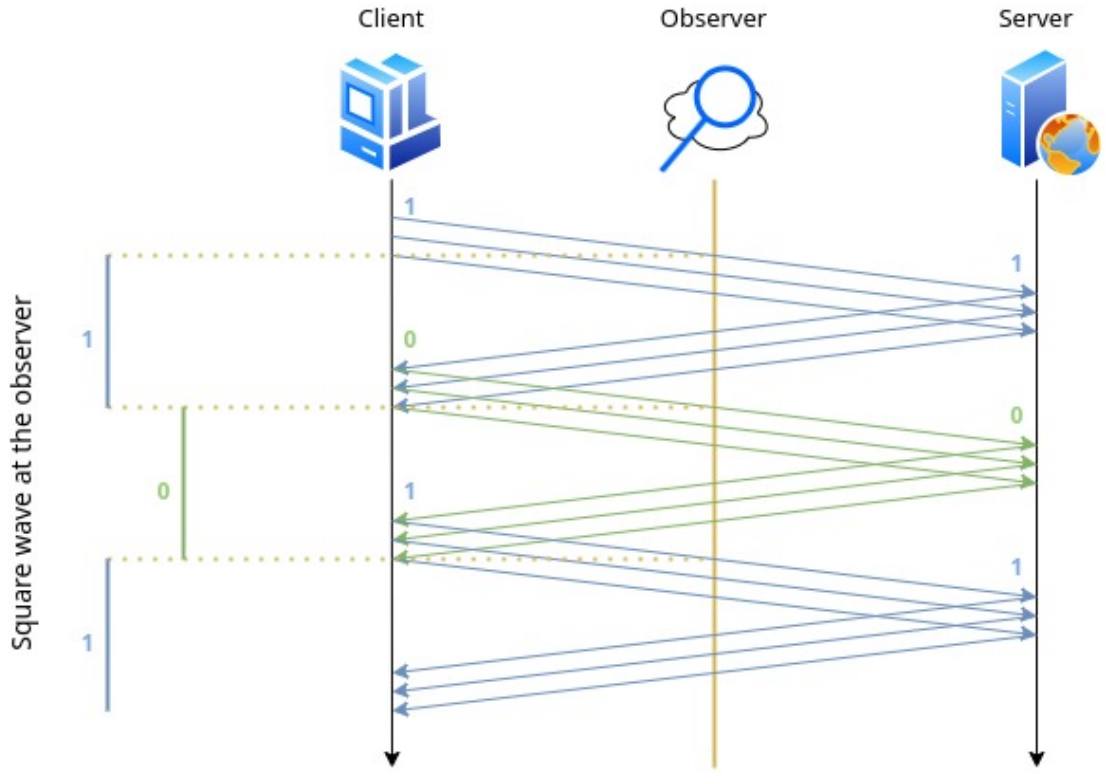

Figure 2.1: Representation of the Spin Bit algorithm.

Caveats

Packet reordering can cause spurious edges detection at the observer, negatively impacting its ability to compute the RTT. Furthermore, the loss of an edge might also afflict the measurements. For this reason it is common to use heuristics to determine the impact of these problems and reject bad RTT samples. Additionally, as application layer delay is also measured by the Spin Bit, if it is larger than the network delay by a significant amount, the observer might not be able to reliably estimate the RTT.
Finally the biggest issue with the Spin Bit resides in its optional nature: as this bit is mostly relevant for network operators, there is no tangible incentive for the developers of client-side and server-side QUIC libraries to implement it. As an example, currently neither Chromium nor Firefox support the Spin

Bit.
Telecom Italia started developing some Spin Bit implementations, including a version for the quic-go library [8] and one for the QUICHE library found in Chromium, developed in collaboration with Politecnico di Torino [9]. However, this does not solve the problem of the **lack of adoption** of the Spin Bit: as each side can unilaterally decide whether the Spin Bit will be used or not, a client-side implementation does not guarantee that the server will reflect the Spin Bit, effectively preventing the generation of the square wave. For the same reason, a server-side library that supports the Spin Bit cannot use it unless the client will flip it; an example of QUIC server-side library that implements the Spin Bit is the *lsquic* library [10], found in the open-source Web server OpenLiteSpeed.

Other explicit flow measurements

Other possible techniques to measure network performance have been described in recent years [11] [12], and they might be particularly valuable in QUIC where we can leverage two extra Reserved Bits in the Short Header.
In particular the **Delay Bit** is designed to overcome the limitations of the Spin Bit: it is set once per RTT period on a packet called Delay sample, and by tracking it and recording the timestamps, a passive observer can measure the RTT without the issue of spurious edges.
Other alternate marking techniques can instead leverage Reserved Bits to measure loss, like in the case of the Q Bit and the L Bit, or congestion, like in the case of the E Bit.

2.3 Chromium

Chromium is a free and open-source Web browser, mainly developed and maintained by Google. The Chromium codebase is used by some of the most popular browsers in the world, including Chrome, Edge and Samsung Internet. About 49% of the 35 millions lines of code in the project are written in C++, while the rest is mainly split between HTML, Javascript and C [13]. All of Chromium's code is versioned using Git and it is publicly available [14].
The checkouts are managed by a package of scripts called *depot tools*, which also includes the scripts used to fetch and update the code and its dependencies. The build system is composed of a combination of *gn* and *ninja*: the

first produces a set of build files, that are then compiled by the second.

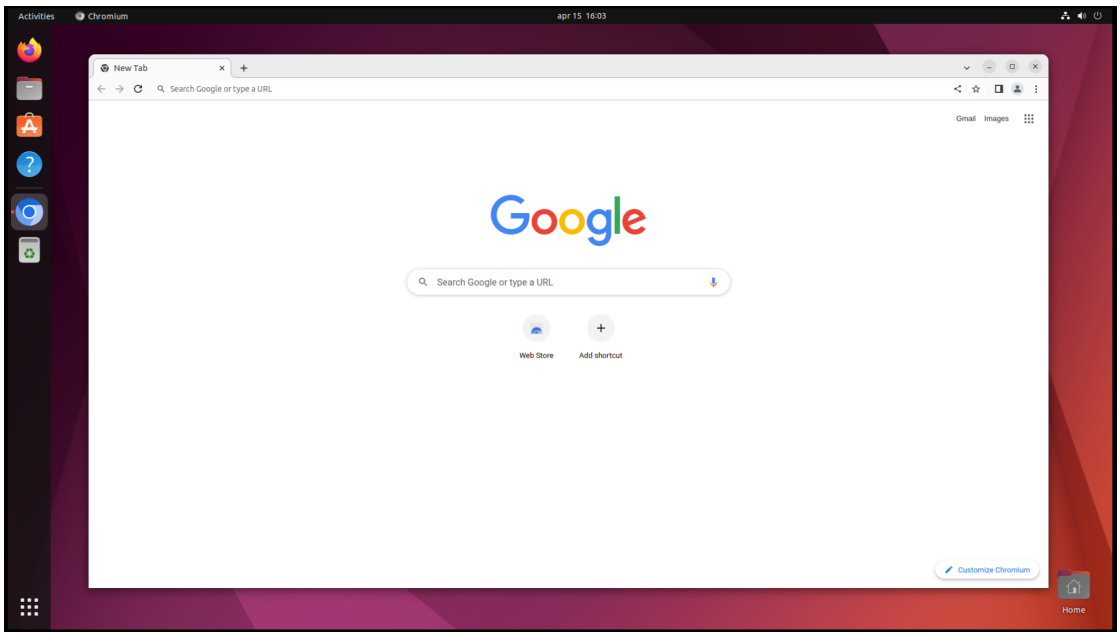

Figure 2.2: Chromium running in Ubuntu 22.04 LTS.

When building Chromium, developers can target a variety of operating systems, including Android, Linux and Windows; *gn* make this convenient as it allows to set different build options in different checkouts.

2.3.1 QUIC support

QUIC has been supported in Chromium since the very beginning, as both projects are mainly developed by Google. In particular Google develops the **QUICHE** [15] library, which is the network library used in all versions of Chromium and Google's servers to deal with QUIC, HTTP/2 and HTTP/3; the library is entirely written in C++.
The components of the library that are most relevant for this thesis are located in the *quiche/quic/core* subdirectory, in particular:

- *quic_connection.cc*: the QuicConnection entity contains the code for QUIC connection functionalities, including the processing of packets, and the code that sends and receives them. Additionally, QuicConnection handles path migrations, Connection IDs and the connection statistics,

by relying on the RttStats class. Finally, QuicConnection has a variety of time related functions that rely on the QuicClock class.

- *quic_packet_creator.cc*: the QuicPacketCreator entity contains the code to create packets from streams on behalf of QuicConnection. This includes filling the payload and the header of the packets, and then returning them to QuicConnection so that they can be serialized and sent.

- *quic_framer.cc*: the QuicFramer entity contains the code that sets the bits in the header, based on the values received by QuicPacketCreator.

- *quic_types.h*: the QuicTypes entity contains the declaration of the QUIC header. In particular, the QuicPacketHeaderTypeFlags enumerated type contains the definition of the header bits I described in this chapter, like the Header Form, the Fixed Bit and the Reserved Bits.

While it supports all the core features of the QUIC protocol, the QUICHE library does not support the optional Spin Bit.

Testing QUIC in Chromium

Chromium provides sample implementations of both a **QUIC Client** and a QUIC Server in its codebase. To use them it is necessary to create a new checkout of the source code and then build the two components.
For the scope of this thesis, I am going to focus mostly on the QUIC Client, since it allows to easily generate requests and it can be controlled from the command line or with a bash script: this is valuable to perform automated tests with no user interaction. Furthermore, the QUIC Client has a number of useful flags:

- *--num_requests*: repeat the same request multiple times during the same QUIC connection.

- *--disable_certificate_verification*: ignore certificate verification, which is instead very strict when using Chromium to generate QUIC traffic, as it requires a certificate issued by a default trusted CA, without the option to trust custom CAs.

- *--quiet*: quietly output debug information, without printing the content of the responses.

The QUIC Client is lightweight and faster to compile than the Chromium browser, giving more flexibility to developers. Finally, the component has access to low level library logs and debug messages. For the sake of clarity, I present an example:

```
$ ./out/Checkout/quic_client --host=127.0.0.1 --port=6121
↪ --disable_certificate_verification https://www.example.org
```

This command will start the QUIC Client and it will make it interact with a QUIC server deployed locally on the developer's machine, even if it has no valid certificate. Notice that the QUIC server can be any server with QUIC support, and not necessarily the QUIC Server found in Chromium.

2.4 Network monitoring

For Internet Service Providers and network operators, monitoring the network is extremely valuable: not only it allows them to have an overview of the general performance and health level of the infrastructure, but it also enables the detection and the resolution of faults and configuration issues.
In particular, **passive network monitoring** is a technique that consists in pulling the traffic produced by the users of the network and analyze it, in order to understand the performance and the status of the whole network, or of a specific segment of it. While active network monitoring relies on artificial traffic injection to predict network behaviour, the passive method relies on real user-generated data: this allows network operators to detect issues that directly impact end-users, and to have an indication of the quality of the user experience, which is particularly relevant for ISPs that want to provide the highest quality of service possible to their customers.
The effectiveness of passive network monitoring is strongly affected by the wire image of the protocol under analysis: as I previously explained in this chapter, QUIC headers have a limited amount of cleartext fields, and the protocol only allows to compute estimations of the RTT by means of the Spin Bit. For this reason, a network operator that wants to monitor QUIC traffic should carefully pick a tool that enables advanced analysis of the protocol, with the ability to register flips in the Spin Bit.

2.4.1 Spindump

Spindump is an open-source UNIX utility developed by Ericsson Research [16], which can be used to perform passive in-network monitoring on a variety of protocols, including QUIC.
In particular, Spindump is able to report Spin Bit values and flips and associate them with a given timestamp and Connection ID. To start a Spindump capture, it is enough to launch the utility from the CLI; while doing so, it is possible to specify a series of options, and it is also possible to include pcap filters. If we want to capture QUIC traffic and report the Spin Bit we can enter the following command:

```
$ spindump udp and port 443 --report-spins
```

The output can be displayed in the terminal or it can be saved as text, in the form of a JSON file.

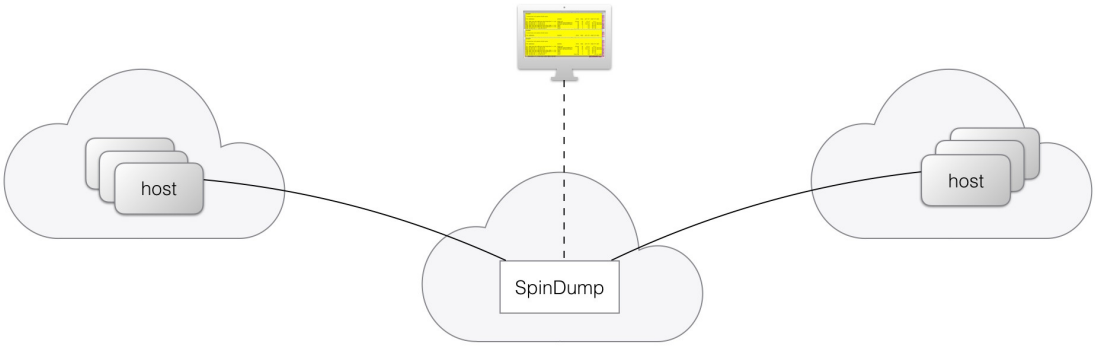

Figure 2.3: Example of Spindump usage from Ericsson's documentation.

2.4.2 Probes on user devices

As previously said, the wire image of network protocols has a large impact on passive network monitoring. Furthermore, explicit flow measurements like the Spin Bit need endpoints' cooperation and they work best when the network operator has visibility in both the directions of the traffic, which might not always be the case. Monitoring all the network is also not likely to scale well: as the number of connected devices and services in today's Internet increases, a passive observer might find itself dealing with huge amounts of traffic and an ever-increasing number of connections.

To work around these issues, Telecom Italia and Huawei propose to enable probes on user devices [17], in order to increase the precision of the measurements, while also reducing the number of connections per-probe. In particular, the proposal suggests that user devices could be convenient places to deploy explicit performance observers by means of applications that willingly measure and transmit performance data to a certain Internet Service Provider. Additionally, user devices could capture data at the source to produce more accurate measurements that exclude client-side application delay, and they are guaranteed to have visibility of the traffic in both directions. The above cited proposal also explains that, when dealing with protocols that support explicit measurements, the cooperation between user devices and network probes could be very useful to **monitor the performance of the network** and to locate faults, which is particularly relevant for this thesis: as the QUIC protocol can leverage the Spin Bit, user applications could use it to provide valuable information in support of passive network monitoring.
It is important to notice that to preserve users' privacy, the packet marking mechanism should be explicitly enabled by the user.

Chapter 3

Inside RTT monitoring

In Chapter 2 I explained the value of network monitoring and I presented some techniques that allow network operators to perform it. However, I also stated that solutions like the Spin Bit might end up being ineffective, as they require cooperation between the client and the server; this problem is further amplified by the lack of adoption of the Spin Bit in existing QUIC implementations. Overall, monitoring the performance of the QUIC protocol is a complex task and network operators would benefit from having better metrics at their disposal, hence I presented the proposal from Telecom Italia and Huawei of using user devices as active elements of network monitoring in an effort to improve explicit flow measurement techniques [17].
For this reason, in this thesis I developed an Inside RTT monitoring technique that allows network operators to monitor the performance of QUIC traffic without using the reflection mechanism found in the Spin Bit. The technique has been proposed and patented by Telecom Italia [18]: the idea is to leverage a **RTT estimation available on the client's device** called Inside RTT, in order to unilaterally generate a square wave that a passive on-path observer can capture to compute the RTT on its end. While this technique is not strictly tied to measurements on user devices, using them as probes guarantees the ability to observe and take advantage of the Inside RTT, effectively allowing to observe a RTT estimation without server cooperation. To implement this technique, I modified the QUICHE library used in Chromium, so that the Web browser could use the already existing Inside RTT of the library to mark the Spin Bit of outgoing packets. All a passive observer would need to do in this scenario, is to capture the QUIC traffic of the connection going from the client to the server, and measure the

duration of the Spin period.

3.1 Algorithm

The Inside RTT monitoring technique enables passive latency monitoring for the duration of a QUIC connection, **without requiring cooperation** between the client and the server [18]. As it uses the Spin Bit header field found in the Short Header of QUIC packets, the algorithm starts after the completion of the handshake, so packets that use the Long Header - and that therefore do not include the Spin Bit - are excluded from this mechanism. As previously explained, the Spin Bit found in the header of QUIC packets is in cleartext by design: a passive on-path observer can therefore capture the timestamp of the first packet with a given Spin Bit and the timestamp of the first packet with the inverted Spin Bit, and use them compute the duration of the Inside RTT interval.

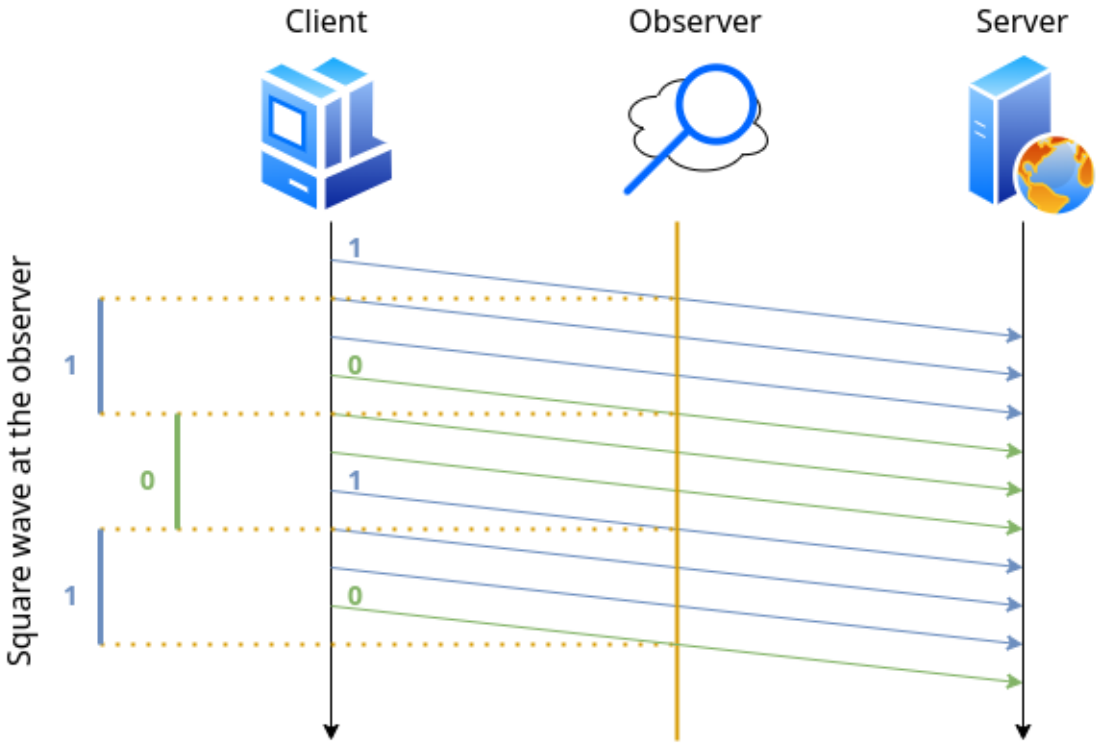

Figure 3.1: Generation of the square wave at the observer.

In the case of the Inside RTT monitoring technique, this is possible

because as shown in Figure 3.1 the client generates a **square wave** by sending packets with the same Spin Bit value for the duration of an Inside RTT interval, which lasts for an amount of time equivalent to the latest RTT estimation computed by the client application. This means that when the client application uses this algorithm, **a passive observer will be able to measure the Inside RTT** computed by the client application without ever needing the server to cooperate, and simply by looking at the QUIC traffic generated by the client and travelling towards the server.
The ability to unilaterally generate the square wave without server reflection is extremely valuable to address the lack of adoption of the traditional Spin Bit in QUIC libraries, as Inside RTT monitoring can be implemented only on one of the endpoints, with no required agreement between client and server: a network operator that is determined to monitor QUIC performance, could develop its own client, distribute it to its users and immediately start capturing RTT estimations. Furthermore, while it was not developed during the course of this thesis, a server side implementation would allow the same type of measurements.
In order to use the algorithm, the client should declare and update a series of values for each connection:

- *Spin Bit value*: the internal value of the Spin Bit, used to mark outgoing packets with the Short Header. It should be initialized to 0.
- *Inside RTT interval*: the end time of the marking interval, during which the client marks outgoing packets with the same Spin Bit value. It should be initialized to 0.
- *Latest RTT*: the latest value of the RTT, as computed by the client application.
- *Current time*: the current time must be obtained by means of a clock placed in the client application. The value of current time and the value of the Inside RTT interval should be comparable, while the value of the current time and the value of the latest RTT should be summable.

If all these values are available at the client, then the algorithm works as follows:

1. Immediately before sending a packet with the Short Header, compare the current time to the Inside RTT interval: if we are past the Inside RTT interval, flip the Spin Bit value.

2. Whenever the Spin Bit value is flipped, compute a new Inside RTT interval as the sum of the current time and the latest RTT. If the Spin Bit value is unchanged, do not update the Inside RTT interval.

3. Set the value of the Spin Bit in the Short Header of the outgoing packets to the same value of the Spin Bit value internally maintained by the client application.

4. Repeat the previous steps for the duration of the connection.

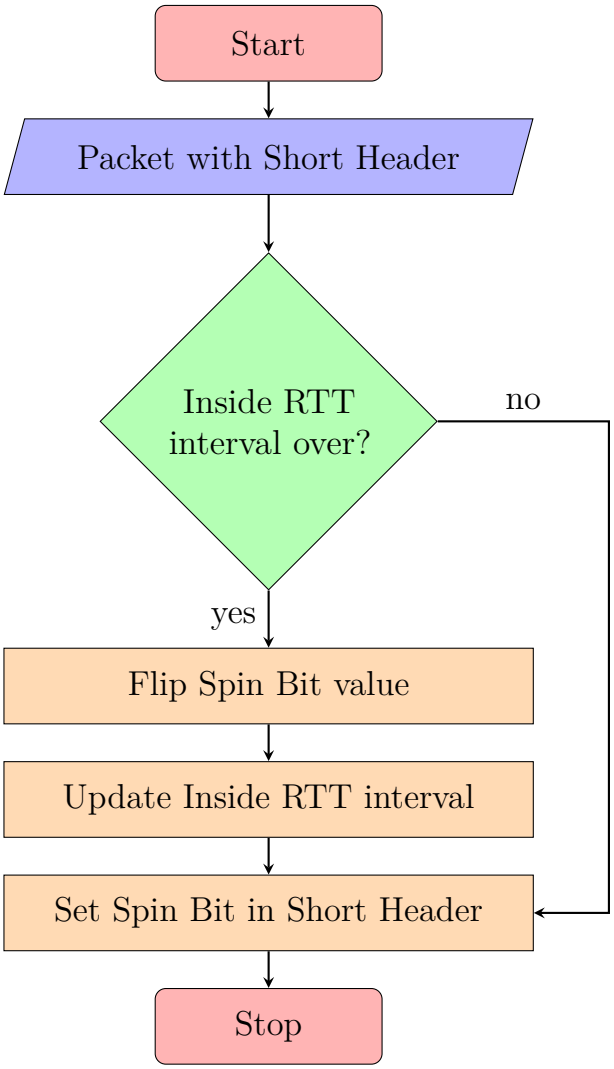

Figure 3.2: The Inside RTT monitoring algorithm.

It should be noted that, since the Inside RTT interval is initialized to 0, the first outgoing packet with a Short Header should always flip the Spin Bit value, and it should also always update the Inside RTT interval.
A final consideration should be made about the risk of connection correlation: in Chapter 2 I explained how the QUIC protocol takes the necessary measures to preserve user privacy by avoiding the reuse of Connection IDs, but also by resetting RTT estimations. For this reason, a privacy conscious implementation of the Inside RTT monitoring technique should **reset the values** it uses for its computations, including the Spin Bit value and the Inside RTT interval, on each change of Destination Connection ID.

3.1.1 Caveats

In order to work, the algorithm requires that the client maintains an internal estimation of the RTT, which might not always be the case. Furthermore, the algorithm uses the Spin Bit field of the Short Header, so it can only be implemented in a QUIC implementation that explicitly declares it in its list of supported headers.
The value of the Inside RTT interval is the sum of the current time and the latest RTT, therefore it requires a clock at the client, and it can only be updated when the client application can provide a first initial RTT estimation. This means that in some instances the start of the algorithm might depend on the client itself, rather than being strictly tied to when the first packet with the Short Header is sent. It should be noted that clients can easily compute the first RTT measurements during the handshake phase, so this should not be a big problem.
One additional issue to consider, is that just like in the case of the Spin Bit algorithm, we are measuring both transport layer delay and application layer delay: this includes delayed acknowledgments and the time necessary to generate a response. Moreover, when performing Inside RTT monitoring, we are strictly relying on the RTT estimation **computed by the client**, so if this computation is not accurate there is no way for the algorithm to correct this and the observer will be able to compute exclusively the RTT as perceived by the client application.
Like in the case of the Spin Bit algorithm, the observer should take into account the problems introduced by flow control performed by the client application, as well as spurious edge detection caused by packet reordering and inaccurate RTT estimation caused by packet loss. RFC 9312 [7] states

that the performance of **a client that only sends small amounts of data periodically is particularly hard to monitor** with the Spin Bit, which instead work best with continuous traffic flows: this also applies to the Inside RTT monitoring technique, and the RFC recommends the use of heuristics or filters to improve the accuracy of the measurements. The traffic flow could also be impacted by client-side caching, which might reduce the amount of overall packets in transit, further impacting explicit flow measurements.
Finally, while it does not require cooperation and adoption by both endpoints, the success of the Inside RTT monitoring technique is still tied to the willingness of the users of the network to use applications that include this mechanism.

3.2 Implementation

While the QUICHE library does not implement the traditional Spin Bit, I previously mentioned the existing implementation of it that was developed internally at Telecom Italia, in collaboration with Politecnico di Torino [9]. This implementation includes Spin Bit support in the Short Header by declaring it in the QuicTypes entity, as well as the QuicFramer code that allows to set the value of the Spin Bit, and the logic of the traditional Spin Bit algorithm described in Chapter 2. My choice was to use this code as the starting point of my implementation, so that Spin Bit support could be readily available and I could focus on the Inside RTT algorithm instead.
Additionally, in the QUICHE library the QuicConnection entity includes the RttStats class, whose role is to declare and to update the Round Trip Time statistics associated to the connection; the entity also includes a clock used for time based functions. The combination of these factors made the library the ideal target for the implementation of the Inside RTT monitoring technique.
At the start of my thesis, I rebased the existing Spin Bit code to make it compatible with the most recent version of QUICHE, and I produced a patch that could easily be applied to the library to enable Spin Bit support. From there I started implementing the variables and the logic required by the Inside RTT monitoring technique, which I will further describe in the rest of this chapter. The full implementation is publicly available [19], alongside a patch for easy portability into a Chromium checkout.

3.2.1 Variables

The algorithm requires a **RTT estimation** and a **clock**, so at first glance the QuicConnection entity looks like the obvious choice for the declaration of the rest of the needed variables and for the development of the logic. Unfortunately, initial testing proved that this component is not close enough to the code that fills the headers of the packets, which leads to unreliable results and losses in precision. For this reason, I instead used **QuicPacketCreator** as the entity that contains most of the code related to Inside RTT monitoring, as it is the component that creates the packets on behalf of QuicConnection.
Since this entity does not have access to a clock, I declared a new variable and initialized it on entity creation, using the clock available in QuicConnection. Furthermore, QuicPacketCreator does not have access to the RttStats class, so I also declared a variable to contain the latest RTT, which is updated by QuicConnection in the following cases:

- Whenever a stream is sent from QuicConnection to QuicPacketCreator.
- Whenever RttStats might compute a new RTT, so after processing acknowledgments and on path validation.

An update occurs only if the new latest RTT value is different from the one already available in QuicPacketCreator, as updates from QuicConnection to QuicPacketCreator might occur more frequently than actual updates of the estimation in RttStats. The code that declares and initializes the variables is available below:

```
// Latest RTT received from the Connection.
QuicTime::Delta latest_rtt_ = QuicTime::Delta::Zero();

// Clock used to compute the Inside RTT interval.
const QuicClock* clock_;
```

Finally, to fully support Inside RTT monitoring, I declared two additional variables: one contains the Spin Bit value, and the other contains the Inside RTT interval.

```
// Spin Bit value maintained internally by the endpoint.
bool current_spin_bit_ = false;

// Interval used for Inside RTT marking.
// It is the sum of the current time and the latest RTT.
QuicTime inside_rtt_interval_ = QuicTime::Zero();
```

It is important to notice that the Inside RTT interval is of type QuicTime, which is the same type returned by the clock when it returns the current time, so the two can be compared. The latest RTT object is of type Delta and it can be summed to QuicTime objects, since it represents the difference between two points in time.
For all the above listed variables, I created setter and getter methods, and I followed the naming and code formatting conventions used in the rest of the QUICHE library.

3.2.2 Logic

In the QuicPacketCreator entity, we find the `FillPacketHeader` function: as suggested by the name, this function fills the headers of the outgoing packets, and it is also the place where the Spin Bit is set. In particular, the below code sets the Spin Bit in the Short Header to the Spin Bit value maintained by the endpoint:

```
void QuicPacketCreator::FillPacketHeader(QuicPacketHeader* header)
 ↪ {
    ...
    if (!HasIetfLongHeader()) {
        MaybeFlipSpinBit();
        header->spin_bit = current_spin_bit_;
        return;
    }
    ...
}
```

Right above the line of code that sets the Spin Bit we find a call to the function that implements the logic of the Internal Spin Bit algorithm:

```
void QuicPacketCreator::MaybeFlipSpinBit() {
    QuicTime current_time = clock_->Now();
    if (current_time >= inside_rtt_interval_)
     {
        if (!latest_rtt_.IsZero())
         {
            inside_rtt_interval_ = current_time + latest_rtt_;
            current_spin_bit_ = !current_spin_bit_;
         }
     }
}
```

As described in the previous section, right before filling the Short Header we compare the current time to the Inside RTT interval, and if we are past it we trigger the code that flips the Spin Bit value and that updates the Inside RTT interval by using the current time and the latest RTT. It is worth mentioning that this operation will be performed only if a valid RTT measurement is available, to handle the corner case in which a Short Header might be sent before the client computed a RTT estimation.
I previously explained that the packet creator does not have access to the RttStats class, so to update the latest RTT in the QuicPacketCreator, the QuicConnection entity can leverage the following function:

```
void QuicPacketCreator::MaybeUpdateLatestRtt(QuicTime::Delta
↪  latest_rtt) {
  if (!latest_rtt.IsZero() && latest_rtt != latest_rtt_) {
      latest_rtt_ = latest_rtt;
  }
}
```

The above function is invoked inside the functions `OnAckFrameStart` and `OnMultiPortPathProbingSuccess` of QuicConnection, as they might trigger RTT updates. It is also invoked in the function `ConsumeData` of QuicPacketCreator, as I modified its signature so that, whenever it consumes a stream, the packet creator also receives the latest RTT measurement from QuicConnection.

Resetting the Spin Bit value and Inside RTT metrics

While other implementations of the Spin Bit do not take into account the issue of traffic correlations, I decided to implement this privacy measure in this Inside RTT monitoring implementation, in an effort to comply with the standard [1] and to preserver users' privacy.
For this reason, on each change of Destination Connection ID the metrics associated to the Inside RTT are reset using a dedicated function:

```
void QuicPacketCreator::ResetSpinBit() {
    current_spin_bit_ = false;
    inside_rtt_interval_ = QuicTime::Zero();
    latest_rtt_ = QuicTime::Delta::Zero();
}
```

In particular, the function is invoked inside the `SetServerConnectionId` function of QuicPacketCreator.

Chapter 4

Testing environment and tools

In this chapter I will present the organization of the laboratory environment in which I conducted the tests for this thesis. I had at my disposal two Linux machines running Ubuntu, physically located inside Telecom Italia's laboratories and connected to each other with a 1 Gbps link: one played both the roles of the client and the observer, while the other acted as a Web server with QUIC support. Both machines had a desktop environment installed, and they were remotely accessible through a management LAN.
In the first section of the chapter I will describe how I deployed the QUIC server, and I will also introduce the tool I used to simulate delay, packet losses and packet reordering between the two machines.
Subsequently, in the second section I will list the client applications I used and how I configured them.
Finally, in the third section I will discuss how I performed the captures and I will present the tools I developed to support Inside RTT monitoring at the observer.

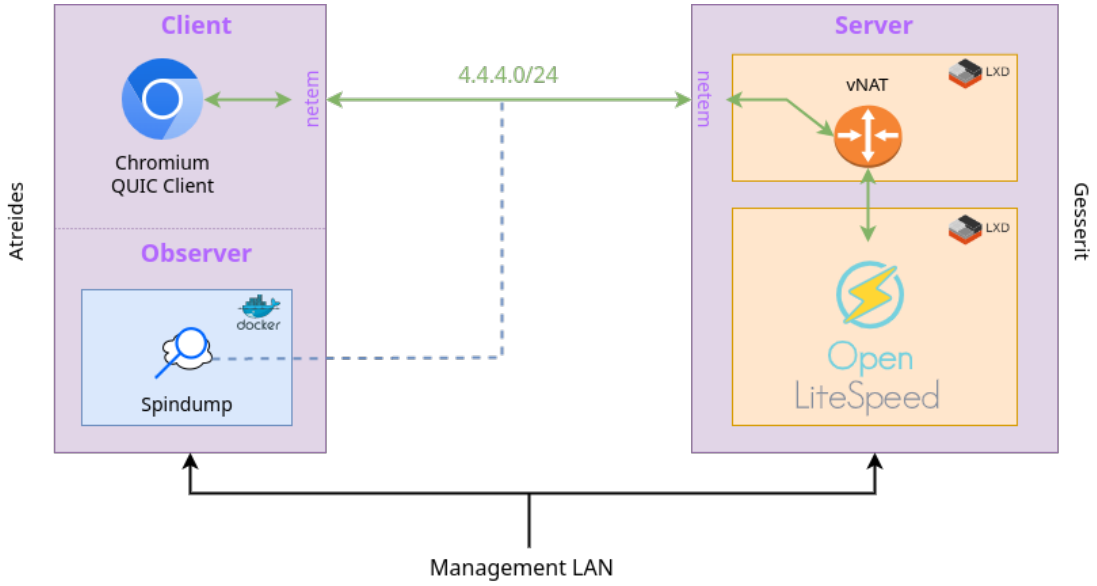

Figure 4.1: Overview of the laboratory testing environment.

4.1 Server

I deployed the server on a dedicated machine named Gesserit. I started creating a LXC container and then I installed the **OpenLiteSpeed** Web server inside it; to allow the Web server to be reachable inside the LAN, I created four virtual NAT devices with the LXD container manager, mapping the ports 80 and 443 of the LXC container to the ones of Gesserit for both UDP and TCP.
The OpenLiteSpeed Web server supports the QUIC protocol out of the box, however it is necessary to install a valid certificate on it: for this step I used OpenSSL to create a custom CA, and then I generated a new certificate that I added to the Web server.
To complete the server configuration I created two web pages:

- *Download/Upload*: the page contains a download panel with three files available, respectively of approximately 55, 120 and 460 MB of size. The page also contains an upload panel that allows users to select one or more files and upload them to the Web server using PHP.

- *Images*: the page contains a collection of six images displayed in a grid. Each image has a size between 100 kB and 3 MB, for a total of

approximately 6 MB for the whole page.

Both pages are styled with CSS using Bootstrap and it is possible to reach them using the navigation bar of the website.

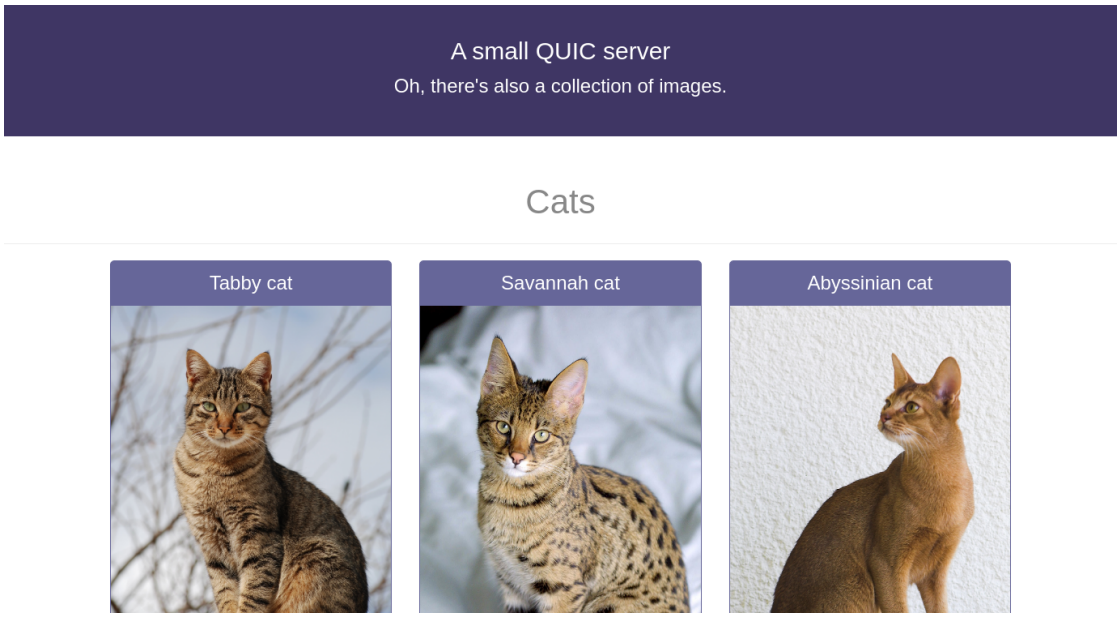

Figure 4.2: A page of the Web server.

To support the testing phase, I also installed **netem** on Gesserit: this Linux traffic control utility, allows to artificially introduce a desired amount of delay, packet loss or packet reordering using the terminal. In particular, on this machine I wanted to introduce delay to control the RTT between client and server.

4.2 Client

I installed the client applications on a machine named Atreides. In particular, on this machine I configured the Chromium build environment and I patched the checkout to include the Inside RTT monitoring code. Finally, I built both the **QUIC Client** and the **Chromium** browser to use them during the testing phase of the thesis.
As mentioned in Chapter 2, the QUIC Client can negotiate with servers ignoring certificate verification. On the other hand, the Chromium browser

has a stricter security policy and it cannot disable certificate verification, nor it can trust custom CAs when negotiating QUIC [20]. For this reason, I had to export the fingerprint of the certificate I added to the OpenLiteSpeed Web server, and I had to run Chromium with the following command:

```
$ ./out/<Checkout>/chrome \
    --user-data-dir=/tmp/chrome-profile \
    --enable-quic \
    --origin-to-force-quic-on=gesserit.local:443 \
    --host-resolver-rules='MAP gesserit.local:443 4.4.4.2:443' \
    --ignore-certificate-errors-spki-list=$(cat
    ↪  /path/to/fingerprints.txt) \
    https://gesserit.local
```

The IP address included in the command is the one of Gesserit, while `gesserit.local` is the hostname I assigned to Gesserit in the host file of Atreides.
On Atreides, I installed netem as well: in this case, the utility allowed me to introduce packet loss and packet reordering on the client network interface, to test how the Inside RTT monitoring reacts to these events.

4.3 Observer

During the tests, Atreides also acts as the observer by performing captures with **Spindump**: to facilitate the deployment, I created a Dockerfile that generates a Docker image containing Spindump and all its dependencies. When creating a container from this image, the container will run Spindump and it will capture QUIC traffic, including the Spin Bit values; the output will be the result of the capture in JSON format.
I automated the configuration of the container by creating a Docker Compose file that:

- Configures the container in host network mode, so that the container can capture the traffic on Atreides.
- Specifies the directory of Atreides to use as the destination of the output.
- Specifies the network interface of Atreides from which the container should capture.

- Sets a pcap filter to exclusively capture traffic going from Atreides to Gesserit.

By using the same Docker image and a slightly different Docker Compose configuration, a network operator could automate captures of QUIC traffic mirrored on an on-path Linux observer.

4.3.1 Developed tools

To support test execution and to analyze their results, I developed three scripts that I am now going to present. The code of the scripts and the containerized Spindump are all publicly available [21].

Parser

Python script that takes captures from the Spindump container, and it produces an output file containing:

- The destination IP address and the Connection ID of the connection.
- The Spin Bit value at different timestamps.
- The lower value of the Spin Bit period, which is the time between the first and the last packet with the same Spin Bit. This is not a valid Spin Bit measurement, but it is useful to understand the shape of traffic flows.
- The upper value of the Spin Bit period, which is the time between the first packet with a given Spin Bit and the first packet with the flipped Spin Bit. This is, by definition, a Spin Bit measurement.
- The number of the Spin Bit flips that occurred during the connection.

As the Inside RTT monitoring algorithm described in Chapter 3 uses the Spin Bit field in the Short Header, this Parser is a valuable tool to **measure the duration of the Inside RTT** computed by the client application, by measuring the duration of the Spin Bit period at the observer, as shown in Figure 3.1.
Alongside captures, the script might also receive as input a QUIC Client log file or the value of the netem delay: in this case the Parser will include the

value of the Inside RTT computed by the client or the value of the delay set through netem in the output, to facilitate the comparison.

```
Destination IP address: 4.4.4.2
    Connection ID: ef2c114c65c0c94b, new connection towards 4.4.4.2
    +++ Flip number: 1
    15:59:42.241915 Spin Bit: 1
    15:59:42.252340 Spin Bit: 1
    15:59:42.253166 Spin Bit: 1
    15:59:42.253408 Spin Bit: 1
    15:59:42.253581 Spin Bit: 1
    15:59:42.254121 Spin Bit: 1
    15:59:42.254478 Spin Bit: 1
    --- Marking interval lasted at least:   15:59:42.254478 - 15:59:42.241915 = 12.563ms
    --- Marking interval lasted at most:    15:59:42.255008 - 15:59:42.241915 = 13.093ms
    ||| Internal RTT in Chromium:   13618us
    +++ Flip number: 2
    15:59:42.255008 Spin Bit: 0
    15:59:42.255371 Spin Bit: 0
    15:59:42.255732 Spin Bit: 0
    15:59:42.256220 Spin Bit: 0
```

Figure 4.3: An example of output of Parser.

Tester

Bash script that instructs the QUIC Client to fetch a given URL, the containerized Spindump to perform captures and the Parser script to generate its output, in order to perform automated tests. It produces:

- A set of QUIC Client log files, one for each fetch.
- A set of Spindump captures, obtained using disposable Docker containers.
- The output of the Parser script, obtained by passing to it the above data as input.

TTE

Time To Evaluate (TTE) is a Python script that takes a set of Parser outputs and produces aggregated tables, that allow to conveniently analyze:

- The expected RTT in ms, expressed as the average of the Inside RTT in QUIC Client log files, or as the value of the delay set in netem.
- The average **deviation** between the expected RTT and the RTT computed by the observer, both in percentage and in milliseconds.

- The average deviation between the expected RTT and the RTT computed by the observer, after **pruning** the outliers. In particular, outliers are pruned when their average deviation is larger in percentage than a given value, set by the tester.
- The number and the percentage of pruned measurements, and the total number of measurements.

By reading the tables produced by TTE, the tester can evaluate how well the Inside RTT is reflected at the observer when using QUIC Client, but also how well the Inside RTT monitoring technique allows to measure the actual RTT between client and server, when using Chromium. Furthermore, the tester can evaluate the effects of pruning, with a clear indication of the impact that this procedure has on the number of available measurements.

AVG RTT (ms)	AVG Deviation (%)	AVG Deviation (ms)	Pruned AVG Dev. (%)	Pruned AVG Dev. (ms)	Pruned (%)	Pruned (#)	Total (#)
27.744	3.646	1.011	2.436	0.676	0.680	1	146
27.028	3.528	0.953	2.654	0.717	0.710	1	140
27.323	3.469	0.948	2.820	0.770	0.670	1	150
27.469	3.803	1.045	2.083	0.572	1.330	2	150
26.738	3.138	0.839	2.203	0.589	0.710	1	140
25.287	2.460	0.622				None	142
27.415	4.229	1.159	2.146	0.588	1.400	2	143
26.402	3.233	0.854	2.291	0.605	0.750	1	133
27.302	3.958	1.081	2.053	0.560	1.360	2	147
27.785	3.493	0.971	2.003	0.557	1.340	2	149

Figure 4.4: An example of output of TTE.

Chapter 5

Evaluation

In this chapter I will examine the Inside RTT monitoring technique I implemented during the course of this thesis, by running a series of both automated and manual tests that give an indication of how well the implementation acts, and how helpful it might be to a passive observer that is trying to monitor the performance of QUIC connections.
In the first section I will present the testing methodology, by explaining for each test its goal, the client I used to conduct it, and the steps I followed during its execution.
In the second section instead, I will show the results obtained with the tests, and I will evaluate the algorithm and the implementation presented in Chapter 3.
Finally in the third an last section, I will present an overall evaluation of the Inside RTT monitoring technique implemented in this thesis, highlighting its strengths and its weaknesses.
All the tests that I am going to present have been executed in the laboratory testing environment described in Chapter 4, using the discussed tools.

5.1 Methodology

The tests I performed can be divided in two macro-categories:

- *Automated tests*: executed using the Tester script and QUIC Client, as the log files of QUIC Client include the debug messages of the QUICHE library I evaluated how accurately the Inside RTT monitoring implementation reflects the value of the Inside RTT computed by the

client application. Furthermore, these automated tests generate QUIC traffic without human interaction so they have a smaller chance of outliers and they can be easily performed in large quantities. As part of these tests, I also evaluated how the implementation behaves in the presence of packet loss and packet reordering.

- *Manual tests*: executed using Chromium, with these tests I evaluated how reliably the Inside RTT monitoring technique allows a passive observer to measure the actual RTT between two endpoints.

It is important to notice that delay and packet losses artificially introduced using the netem tool are reliably measured using the ping command.

5.1.1 Automated tests

As previously mentioned, the goal of these tests is to evaluate **how accurately the implementation can reflect the Inside RTT** computed by the client application. It is important to remember that the Inside RTT value is not computed by the algorithm, but instead it is independently computed by the client application, so as mentioned in Chapter 3 the Inside RTT monitoring technique strongly relies on the client's ability to provide an accurate measure.
Automating these tests also helps reducing the impact of human interaction, while allowing to produce a larger sample of available data with an increased level of reproducibility.
During the execution of the test the containerized Spindump will capture QUIC traffic, and at the end Parser will produce its output: this process is fully automated by the Tester script.

Downloads

This kind of test is performed by configuring the Tester script to instruct QUIC Client to download a file from the QUIC server. The test was repeated using:

- 3 different files available on the OpenLiteSpeed Web server, which have a size of approximately 55 MB, 120 MB and 460 MB, as mentioned in Chapter 4.
- 7 different delay values, introduced using netem: 5 ms, 10 ms, 15 ms, 25 ms, 50 ms, 100 ms and 500 ms.

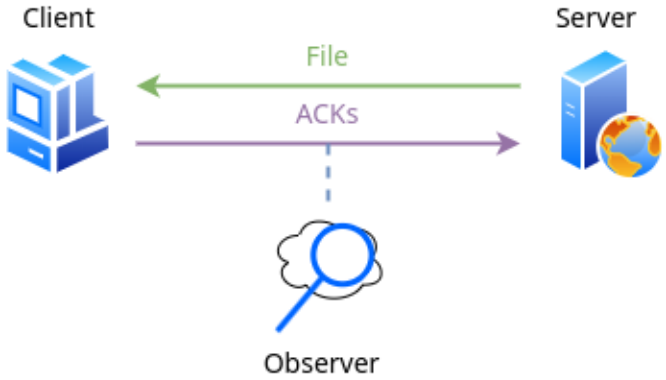

Figure 5.1: Traffic flow of a download.

In particular, I executed 10 automated tests for each combination of file and delay, for a total of 210 tests.

Images

This kind of test is performed by configuring the Tester script to instruct QUIC Client to visit the Images page of the QUIC server. Like in the previous test, I used seven different delay values and I repeated each test 10 times, for a total of 70 tests.

Packet Loss

This kind of test is performed by configuring the Tester script to instruct QUIC Client to download the 55 MB file from the QUIC server, with a delay value of 25 ms introduced using netem. Additionally, I run this test by configuring different loss rates on the client's network interface:

- A random loss rate of 0.5%.
- A random loss rate of 3%.
- A loss rate of 0.5% with a correlation of 50%, to simulate the loss of packet bursts.

Each automated test was repeated 5 times, for a total of 15 tests.

Figure 5.2: Traffic flow of a download with packet loss at the client.

Packet Reordering

This kind of test is performed by configuring the Tester script to instruct QUIC Client to download the 55 MB file from the QUIC server, with a delay value of 25 ms introduced using netem. Additionally, I repeated this test by configuring different reordering rates on the client's network interface, in particular:

- A 25% reorder rate with a delay of 10ms and a correlation of 50%. This means 25% of the packets are sent with a delay of 25 ms, while the remaining 75% is sent with a delay of 35 ms, effectively causing packet reordering.
- A similar approach to the one described above, but a reorder rate of 10%.
- A similar approach to the one described above, but a reorder rate of 1%.

Each automated test was repeated 5 times, for a total of 15 tests.

5.1.2 Manual tests

The goal of these tests is to evaluate **how reliably a passive observer can use the Inside RTT monitoring technique** described in Chapter 3 to measure the RTT between two endpoints. As these tests are meant to emulate real world usage, their results will be impacted by human interaction and by the level of accuracy that the client application can guarantee when

it computes the Inside RTT: while this value is not computed as part of the the Inside RTT monitoring algorithm and it is therefore not under testing on its own, it is still valuable to evaluate the impact that it has when using a real browser, to understand the level of reliability that the Inside RTT monitoring technique can guarantee to an observer.
During the execution of the test the containerized Spindump will capture QUIC traffic, and at the end Parser will produce its output.

Downloads

This kind of test is performed by using the Chromium browser to visit the download page of the OpenLiteSpeed Web server, and then start the download of the 55 MB file.
The test was executed with 4 different delay values: 5 ms, 10 ms, 25 ms and 100 ms. The test was repeated 10 times for first three values and 5 times for the last one as it is not considered a normal network condition [22], for a total of 35 tests.

Images

This kind of test is performed by using the Chromium browser to visit the Images page of the OpenLiteSpeed Web server, with a delay value of 25 ms introduced using netem. The test was repeated 10 times.

Uploads

This kind of test is performed by using the Chromium browser to upload 10 small files at once to OpenLiteSpeed Web server, with a delay value of 25 ms introduced using netem. The size of the files ranges from 10 kB to 1.2 MB, and it includes text files and deb files for a total of approximately 3.5 MB.
It is important to notice that the tester must manually select the files to upload from a file picker, which might impact the traffic shape. The test was repeated 10 times.

Navigation

The goal of this test is to simulate unpredictable user behavior by randomly combining the possible interaction with the OpenLiteSpeed Web server: for this reason, there are no strict steps to reproduce in this case, but the tester

Figure 5.3: Traffic flow of an upload.

is expected to open the home page and visit the different pages, but also to perform downloads and uploads. The test was repeated 10 times, each with different user behavior patterns, but with the same delay of 25 ms introduced using netem.

5.2 Results

In this section I am going to present the results of the tests I described in Section 5.1.
In particular, in the first subsection I will discuss how the implementation performs in the task of reflecting the Inside RTT, by analyzing the results of the automated tests. Additionally, I will present how the results change when filtering some of the measurements with a pruning technique, and how the behavior of the Inside RTT monitoring technique is impacted by different traffic patterns.
In the second subsection I will evaluate what value the Inside RTT monitoring technique brings to a passive observer, by analyzing the results of the manual tests, with the same focus on different traffic patterns and pruning.
In the third subsection I will discuss the effects of packet loss and packet reordering on the technique, and finally in the fourth and last subsection I will present an overall evaluation and some final considerations.

5.2.1 Accuracy of the implementation

The first aspect I will evaluate is the accuracy of the Inside RTT monitoring implementation, so how precisely the code developed during the thesis can reflect the Inside RTT value computed by the client application. As previously explained, I executed a set of automated tests with a combination of different delay values and different file sizes, to see how the implementation would react in these instances: in particular, I was interested in analyzing the deviation between the Inside RTT computed by the client and the RTT measurement performed by the observer.

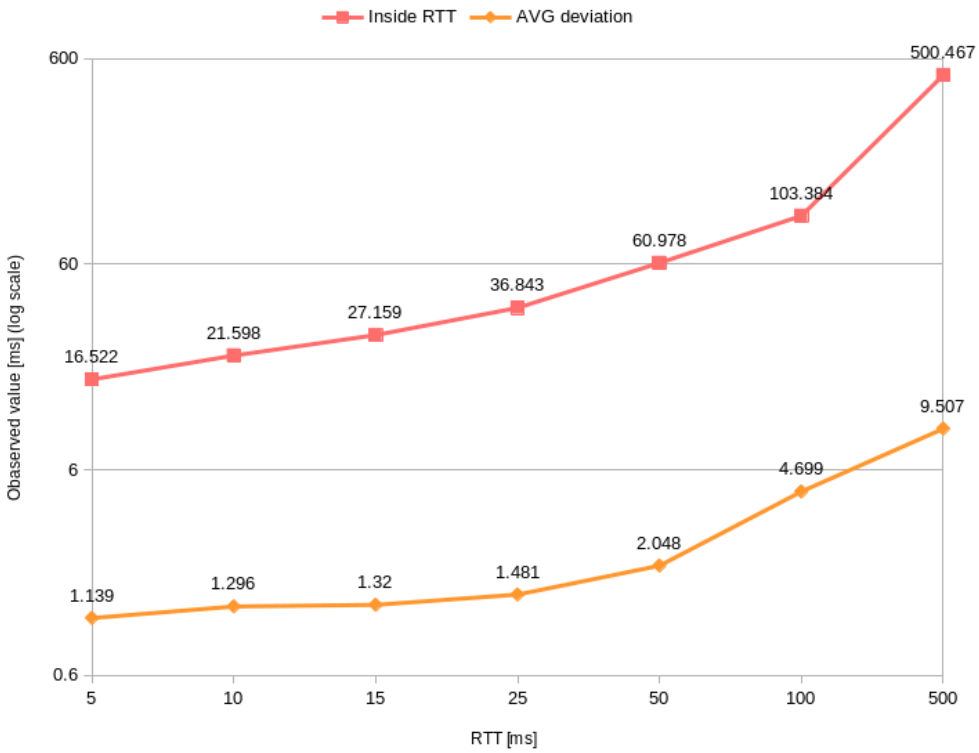

Figure 5.4: Average deviation trend for different Inside RTTs.

The results show an **average deviation of approximately 3 ms** when performing downloads with the QUIC Client, meaning that the observer measures a RTT that is in the Inside RTT ± 3 ms range on average. Additionally, the plot in Figure 5.4 shows that the value of the Inside RTT has a strong impact on the average deviation: most notably, the measurements at the observer have 9.5 ms of average deviation when reflecting an Inside RTT

of 500 ms, while we get way more precision for smaller Inside RTT values, as the deviation stays below 1.5 ms for all the tests where the Inside RTT is ~36 ms or lower. Finally, with a a 60 ms Inside RTT the measurements have a 2 ms deviation on average. It is important to keep in mind that some Inside RTT values used during these tests are more relevant than others: for example, a 500 ms RTT is not a normal network condition [23] and even 100 ms is considered a poor condition for some use cases [22]. Furthermore, while this hypothesis was not analyzed in-depth during this thesis, the QUIC protocol standard [1] [24] explains that the RTT estimation computed by the client plays a significant role in the flow control mechanism and in congestion control, which means that this value impacts the shape of the traffic: since the Inside RTT monitoring technique generates a square wave using the header of packets in transit, a passive observer should be aware that under less than ideal network conditions the flow of the observed packets could be limited by the client application, resulting in less accurate measurements.

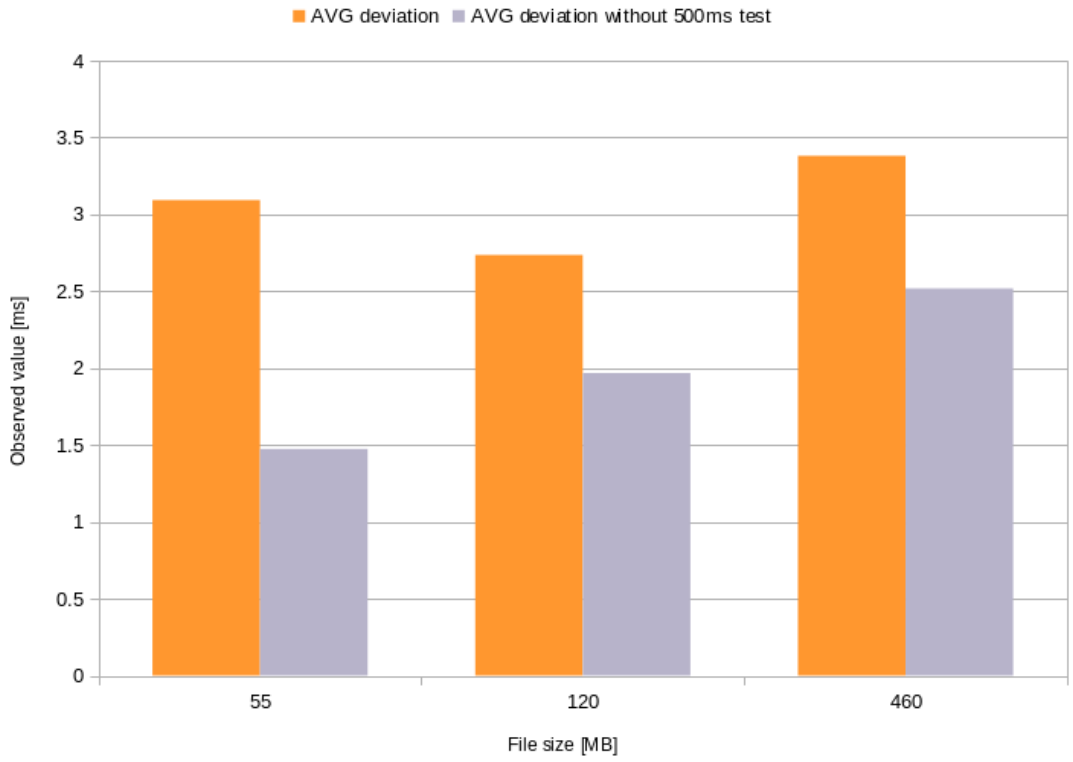

Figure 5.5: Average deviation trend for different file sizes.

The plot in Figure 5.5 shows instead the impact that the file size has on the average deviation. We can see that there is no clear correlation between the file size and the average deviation, however if we exclude the tests where the Inside RTT is 500 ms, the trend changes: other than the expected overall improvements in precision, the plot shows that the average deviation grows with the file size. In particular for the 55 MB file we have the best performance with an average deviation of 1.5 ms, while the 120 MB and the 460 MB file show respectively 2 ms and 2.5 ms. While downloading large files might not be as common as downloading smaller ones, the tests performed using the 460 MB file are the ones that contain the most measurements, as the traffic flow is longer and the number of Spin Bit periods is higher. This means that even if this might not be the most likely use case, it is also the largest data sample available to evaluate the accuracy of the implementation, so its results should not be downplayed.
Taken these considerations into account, the Inside RTT monitoring implementation does an overall good job at reflecting most Inside RTT values when it has a **constant flow of traffic**, with an average deviation of a few milliseconds experienced by the passive observer performing measurements. It is important to keep in mind that this does not mean that the observer is necessarily measuring the effective RTT, as in fact Figure 5.4 shows that QUIC Client does not estimate the RTT with precision during most of the tests.

Pruning the outliers

The average deviation decreases once the TTE script is set to prune all the measurements with a deviation percentage that is larger than 100%: with this approach most measurements deviate less than 2 ms from the Inside RTT value, and for Inside RTTs that are ~36 ms or lower the average deviation is less that 1 ms. Figure 5.6 shows a comparison of the average deviation pre-pruning and post-pruning, for a set of different average Inside RTTs.
It is important to keep in mind that a tester is clearly in a privileged position, as it has access to details that a passive observer would not have at its disposal: in fact, TTE allows the tester to easily detect outliers in this case, because it has access to the QUIC Client log file, hence it can compute the average Inside RTT. For a passive observer filtering is much harder, since it is not as obvious whether a measurement is an outlier or

not. Furthermore, the heuristic must be based on a RTT value that is external to the implementation and that the observer must estimate on its own. Additionally, in this controlled environment and with automated tests it is rather easy to attribute outliers to the application delay and to the shape of the traffic flow, while instead a passive observer must take a purely statistical approach to try to distinguish between an outlier that must be filtered, and a set of measurements that indicate an actual increase of the RTT between two endpoints.

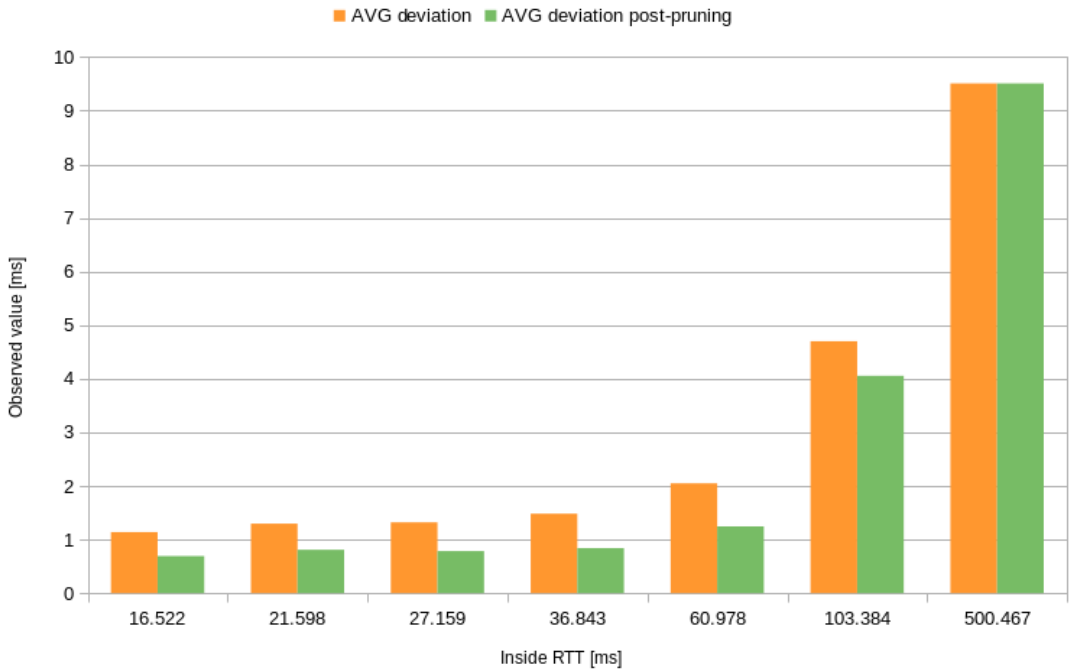

Figure 5.6: The effects of pruning for different Inside RTTs.

The pruning rate decreases as the Inside RTT value increases: in particular, there is no benefit for the measurements that reflect an Inside RTT of 500 ms. Instead, for the measurements performed with an average Inside RTT of 103 ms TTE pruned 0.3% of the values and it provided an improvement of 0.6 ms; for the smaller Inside RTT values the pruning rate ranges from 0.5% to 0.7% of the measurements and it brings improvements that range between 0.4 ms and 0.8 ms, so pruning a very small number of observed values results in significant improvements most of the time. The plot in Figure 5.7 shows the effects of pruning for different file sizes, presenting a comparison with the

data from Figure 5.5 that confirms the trends of that plot, with an overall improvements of the observed values guaranteed by TTE's filtering.

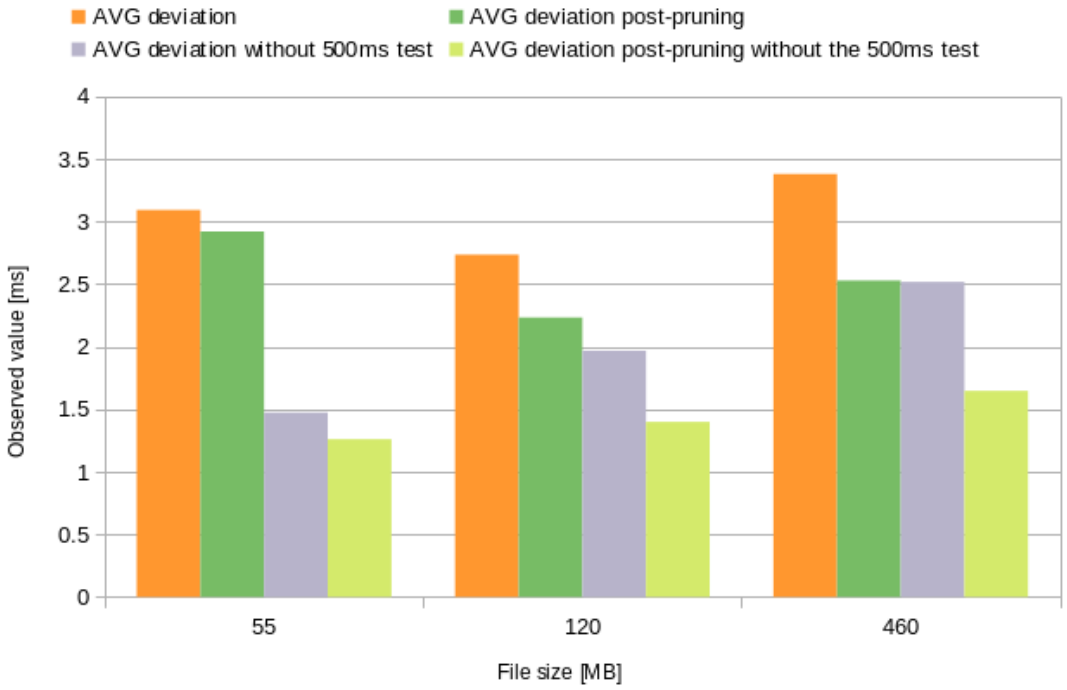

Figure 5.7: The effects of pruning for different file sizes.

Testing with the Images page

The results of this set of tests highlights very well the limitations of the Internal RTT monitoring technique that I described in Chapter 3, many of which are shared with the traditional Spin Bit. In fact, this algorithm suffers particularly when the traffic flow is not continuous, and since in this test QUIC Client is exclusively fetching the Images page once from the OpenLiteSpeed Web server, the client is sending a small amount of ACKs. As a result, only 36 out of the 70 tests of this kind produce valid measurements, as in the rest of the tests the traffic exchange does not last long enough to allow the Spin Bit period to end; this is particularly noticeable for the larger Inside RTT values, as the Spin Bin period lasts longer.
It is worth noting that the data sample produced by this kind of test shows an average deviation of 0.7 ms, however the 36 measurements are the result of captures where a single Spin Bit flip occurred, which makes the reliability

of this evaluation even more questionable. Overall, the results of the tests conducted with the Images page of the OpenLiteSpeed Web server confirm that **the Inside RTT monitoring technique**, like other explicit flow measurements, **needs a continuous flow of traffic** to operate reliably.

5.2.2 Value of the implementation for a passive observer

In the previous subsection I analyzed the results of the automated tests, which highlighted how the implementation can reflect the Inside RTT computed at the client with relatively good accuracy when in presence of a continuous traffic flow going from the client to the sever. Furthermore, I showed how filtering the results might allow to improve the precision of the RTT estimations computed by a passive observer. As previously explained, the Inside RTT monitoring technique transmits a RTT estimation computed by the client application independently from the implementation: for this reason, the performance of the algorithm might be strongly affected by the level of competence of the client in computing the RTT. It is therefore interesting to evaluate what is the impact of this caveat for a passive observer that is trying to estimate the RTT between the client and the server.

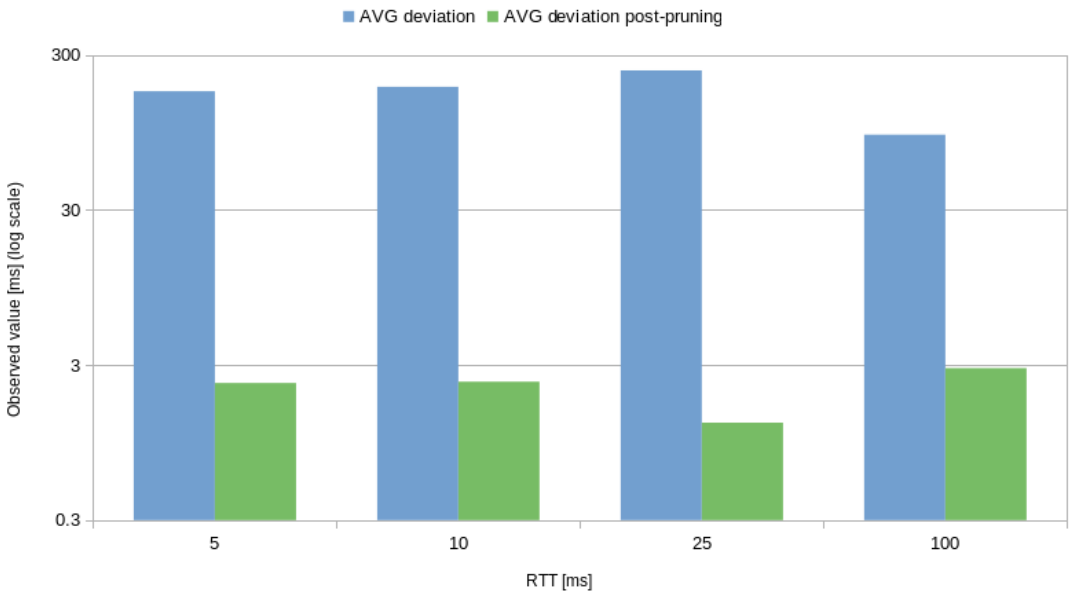

Figure 5.8: The importance of pruning when testing manually.

Generally speaking, Chromium does a good job at estimating the RTT, by producing Internal RTT values that are comparable to the ones observed with the ping command. Instead, QUIC Client does a poor job for RTT values below 100 ms as it tends to over-estimate in these instances; the Inside RTT values are comparable for larger RTTs.
To further replicate a real world scenario, the manual tests include human interaction, which also impacts the shape of the traffic: this introduces outliers, and we can see in Figure 5.8 that if we do not prune them the average deviation is very high. In the same figure we can instead notice that the average deviation drops below 3 ms in all the instances, once we prune the measurements with a percentage of deviation that is larger than 100%. While this is the same approach used for the automated tests, in this case the pruning rates increase by a significant margin: figure 5.9 shows that the percentage of pruned measurements passes the 5% mark for RTTs of 25 ms, and it reacheas 9% for RTTs of 10 ms. Furthermore, this approach causes the pruning rate to reach more than 50% for the tests with a RTT of 5 ms.

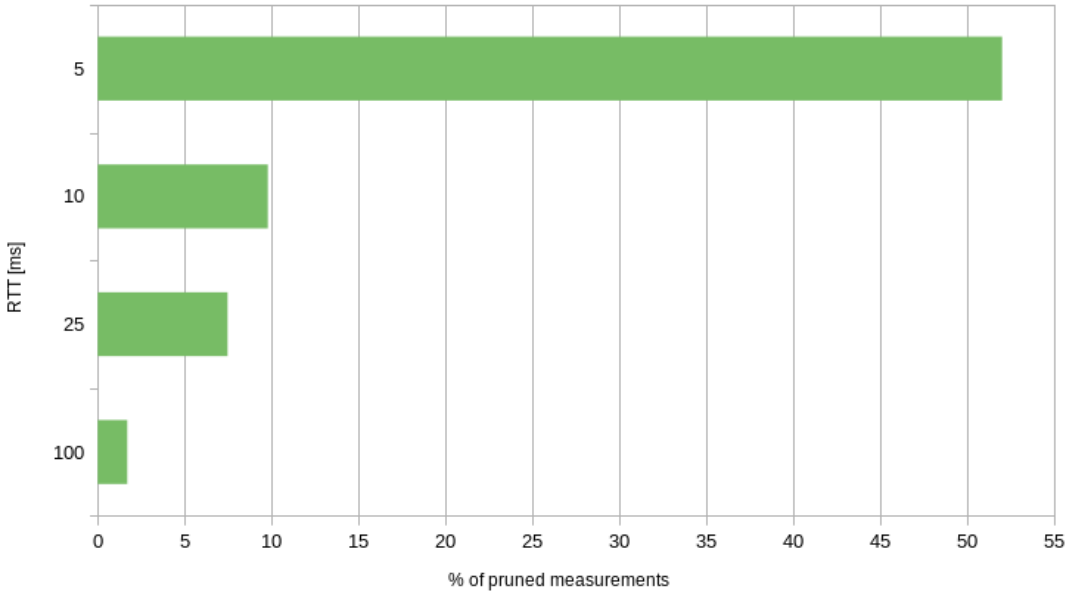

Figure 5.9: The pruning rates when testing manually.

I tried to prune the results for the 5 ms tests with a more relaxed rate, removing exclusively the results with a deviation of 200% or higher, which resulted in an average deviation of more than 4 ms. This tells us that

picking a filtering mechanism is not an easy task, and it might be a trade-off between number of measurements and precision; additionally a network operator might be interested in observing outliers. It is important to notice that, while a 5 ms RTT is not a realistic value in today's Internet, it provides a good testing platform for the implementation, as it highlights the complexity that a network operator might face when using it to passively monitor the performance of the QUIC protocol.
The plot in Figure 5.10 shows the value of the average deviation for Chromium and QUIC Client after testing with the same RTT values set using netem, and after pruning the outliers with the same filtering mechanism.

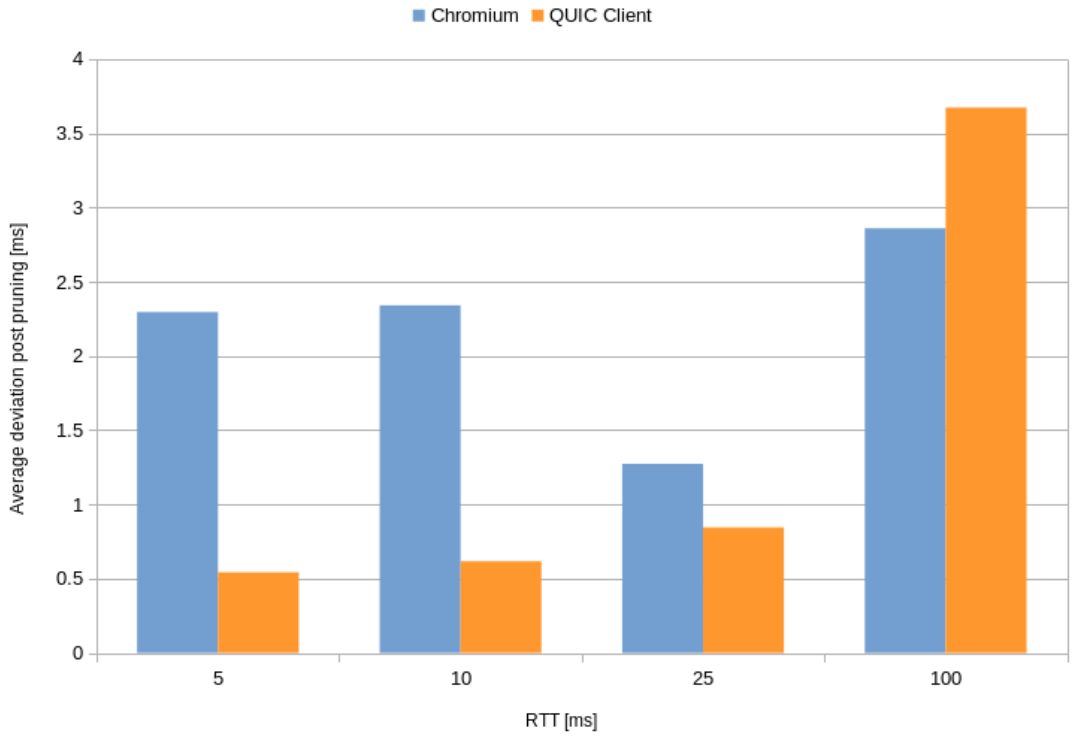

Figure 5.10: Comparison of the average deviation for Chromium and QUIC Client.

We can see that on average QUIC Client performs better, however if we consider that these tests were automated and that the average deviation for Chromium's results is always below 3 ms we can say that overall the Inside RTT monitoring implementation developed during the thesis gives good results with the Chromium browser **in presence of continuous flows of**

traffic. In particular **the average deviation after pruning the outliers is of approximately 2.5 ms**, meaning that the observer measures a RTT that is in the RTT $\pm$ 2.5 ms range on average. Thanks to the tests I presented in the previous subsection, we can say that the implementation transmits the Inside RTT with good precision, so **the Inside RTT monitoring technique provides value to a passive observer trying to measure the RTT** between a client and the sever, **as long as the RTT estimation at the client is reliable**.

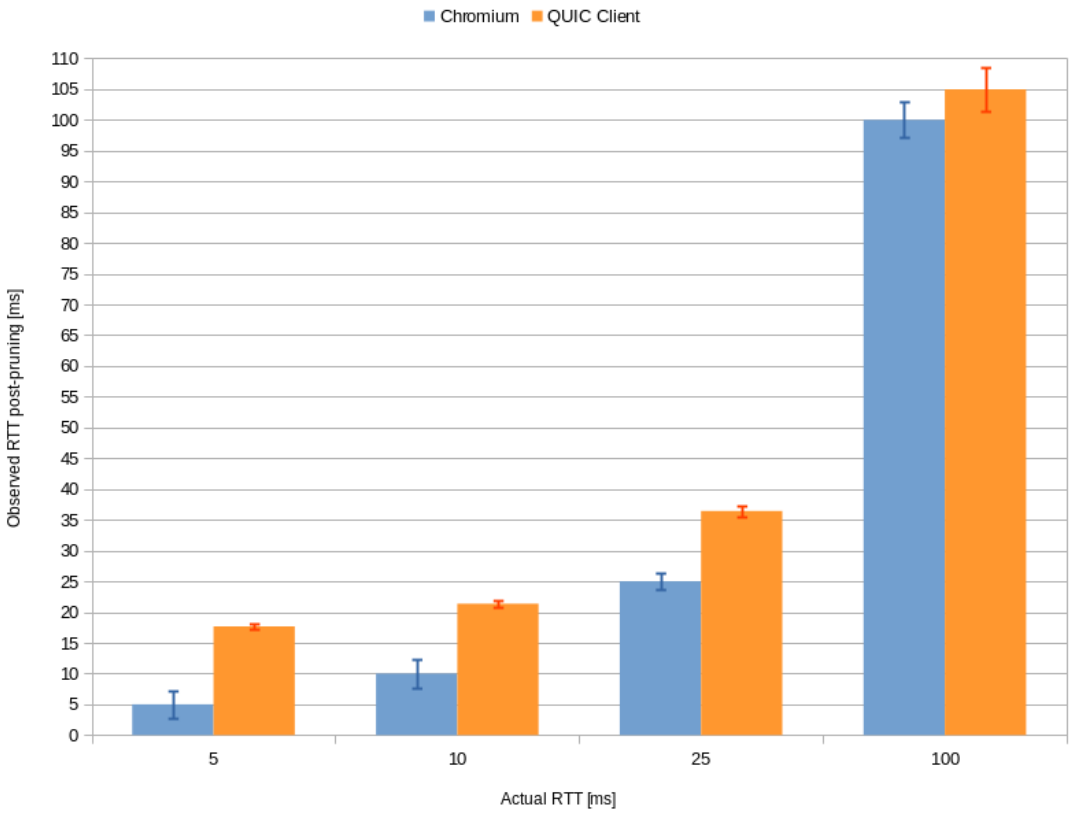

Figure 5.11: Comparison of the observer RTT for Chromium and QUIC Client.

While the inside estimation performed by the client application was never under test on its own, nor it is part of the algorithm, Figure 5.11 shows how important that aspect is for the success of the technique: Chromium might have a larger average deviation, but the implementation is **precise enough to allow the passive observer to measure a RTT value that**

is close to the actual RTT, with a few milliseconds of error. Instead, despite being slightly more accurate in reflecting the Inside RTT value, the results obtained with QUIC Client are not as close to the actual RTT: for this reason, when it uses the Inside RTT monitoring technique, a passive observer should always remember that it is measuring exclusively the Inside RTT as it was experienced and computed by the client application, which may or may not be equivalent to the actual RTT of the connection. Figure 5.11 also shows that, when trying to estimate the actual RTT between two QUIC endpoints, the impact of the reliability of the Inside RTT computed by the client application is far larger than the impact of the accuracy of the Inside RTT monitoring implementation, which is a good result for the scope of this thesis and in general for network operators that might be interested in using this technique in the Chromium browser.

Testing with the Images page

The already questionable results I described in the subsection dedicated to automated tests, are even worse when repeating this test manually with the Chromium browser. In particular, only 4 out of the 10 tests produce results, and even those have very high pruning rates that range between 60% and 80% of the measurements; I tried to relax the pruning mechanism to remove exclusively the results with a deviation of 200% or higher, however this resulted in an average percentage of deviation of more than 100%, with a number of results that was still very low.
Like in the tests conducted with QUIC Client, I want to emphasize the necessity of a continuous flow of traffic to allow the algorithm to work properly. Furthermore, the manual tests introduce human interaction so the results are even more unreliable. It is important to keep in mind that the Images page is 6 MB of size, which is more than the average size of a desktop Web page in today's Internet [25]: this means that the Inside RTT monitoring technique might not be completely usable if the user of the client application is casually browsing Web pages, because just like in the case of the traditional Spin Bit, **it is hard** to use the algorithm **to monitor the performance of a client that sends small amounts of data periodically**. This problem might be further amplified by the fact that modern browsers tend to cache resources to save bandwidth and to increase the navigation speed perceived by the user.

Upload and navigation

When testing with uploads, we see a behavior that is similar to the one observed for downloads: as we have a continuous flow of traffic between the client and the server, the average deviation is approximately 2.4 ms after pruning. It is however important to notice that in these tests the human component is even more impactful, as the tester must first manually pick the files to upload, and then he must hit the upload button. As a result, the pruning rate is higher, at around 28%; this number is also amplified by the fact that the number of Spin Bit periods is lower, because the size of the uploaded files is smaller than the size of the downloaded files.

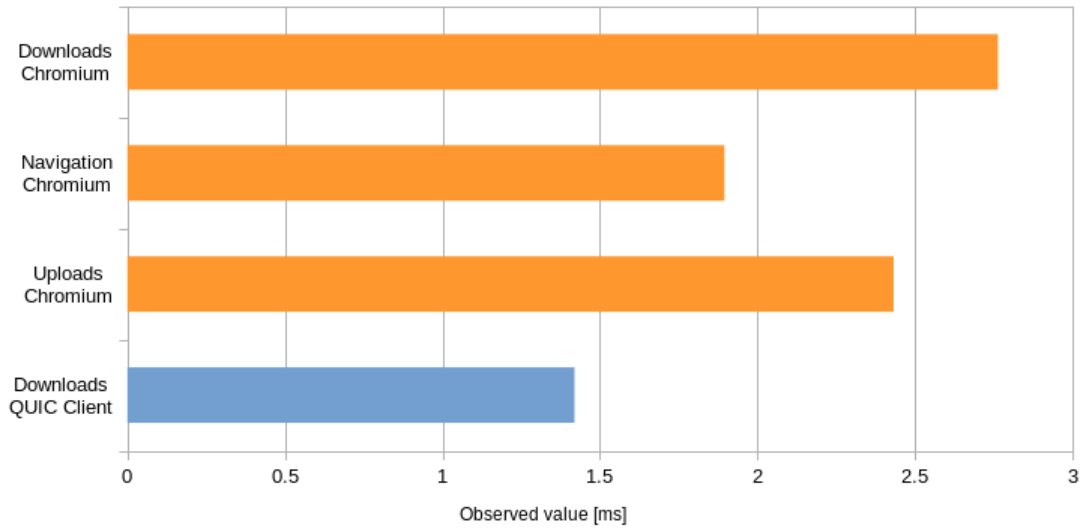

Figure 5.12: Comparison of the average deviation for Chromium and QUIC Client under different testing conditions.

The tests with navigation register an average deviation of 1.9 ms, however this average value does not capture the full picture: the 10 tests conducted vastly differ between each other, depending on the traffic pattern of the test. As a result the pruning rate is really high, at around 36%, and in the tests that do not include downloads or uploads it jumps over 50% leaving a very limited number of measurements to the tester, like in the case of the tests with the Images page. Instead, the tests that include a combination of uploads and downloads perform well, confirming the trends I already described in the course of this chapter. Overall, while on average the navigation results

look reliable, they are strongly impacted by the presence of uploads and downloads, and they require a very high pruning rate because, once again, the Inside RTT monitoring technique does not operate reliably when the user is generating small amounts of data periodically.

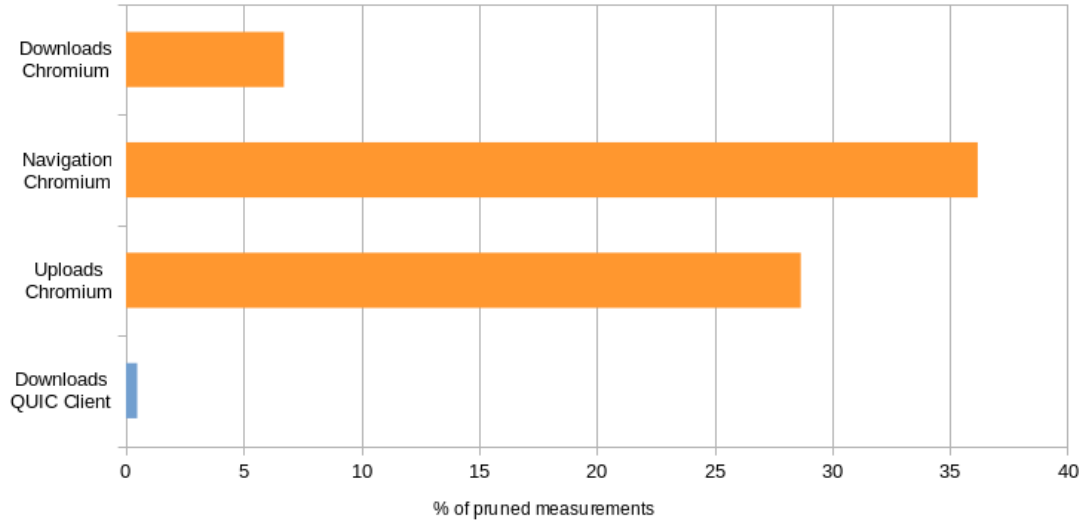

Figure 5.13: Comparison of the pruning rate for Chromium and QUIC Client under different testing conditions.

As a final consideration on the tests with uploads and navigation, it is important to keep in mind that the sample size of the measurements is much smaller than the one of the tests with downloads, so it is safe to say that the slightly worse average deviation of the tests with downloads might be a better indication of the actual accuracy of the implementation in reflecting the RTT to a passive observer.

5.2.3 The effects of packet loss and packet reordering

In this subsection I am going to analyze the results of the tests where I introduced packet loss and packet reordering using netem, in order to understand the impact that these events have on the measurements that a passive observer can perform when using the Inside RTT monitoring technique.
Packet Loss can impact the precision of the algorithm, creating holes in the traffic flow that extend the duration of the Spin Bit period by stretching

its edges. Instead, packet reordering can change the alternance of packets with the Spin Bit value set to 0 or 1, causing spurious edges detection and changing the duration of the Spin Bit periods as a result.

Packet loss

In the case of a loss rate of 0.5%, the results are almost identical to the ones obtained when testing with similar delays and no packet loss, which tells us that if the loss rate is fairly low the implementation does not regress significantly. Instead, with a loss rate of 3% the average deviation increases by 0.8 ms, so in case of subpar network performance the technique loses almost a full millisecond of precision on average. While this level of impact is tolerable, it is important to keep in mind that such a small increase in deviation will not be easy to filter out, as the deviation is small enough that measures might not look like obvious outliers in the presence of a constant traffic flow. A final consideration should be made about the fact that the impact of packet loss has a certain degree of randomness to it, as the packets that are lost may or may not be at the edge of a Spin Bit period.

Packet reordering

Reordering rates of 1% show no significant impact on the average deviation, however TTE reports one or two extra Spin Bit periods at the observer, compared to the ones actually generated by the client application, in this case QUIC Client. Instead, with a reordering rate of 10%, not only we see an increased number of Spin Bit periods, but some measurements deviate by approximately 30 ms from the Inside RTT value: these entries are not obvious outliers, so a packet reordering rate of 10% or higher can impact the accuracy with which a passive observer can measure the Inside RTT, and more importantly it can do so silently, as the Packet Numbers is not in cleartext and the observer cannot tell that packets have been reordered. It is worth mentioning that one particular Spin Bit period was affected by packet reordering and it increased past 110 ms, but in this case TTE did a good job at recognizing it as an outlier and pruning it.
Unfortunately, **with a reordering rate of 25% the Inside RTT monitoring technique is completely unusable**: Figure 5.14 shows that TTE reported an increase in the number of Spin Bit flips at the observer of almost 200%, compared to what QUIC Client actually transmitted. As a result

the Spin Bit periods are very short and the RTT estimations computed at the observer are completely unreliable, sometimes measuring less than a millisecond which is not a realistic value.
Ideally, a network operator could develop an heuristic that can filter these kinds of results based on the RTT history, allowing it to detect packet reordering at such a high rate, and ignore these results until the reordering rate decreases.

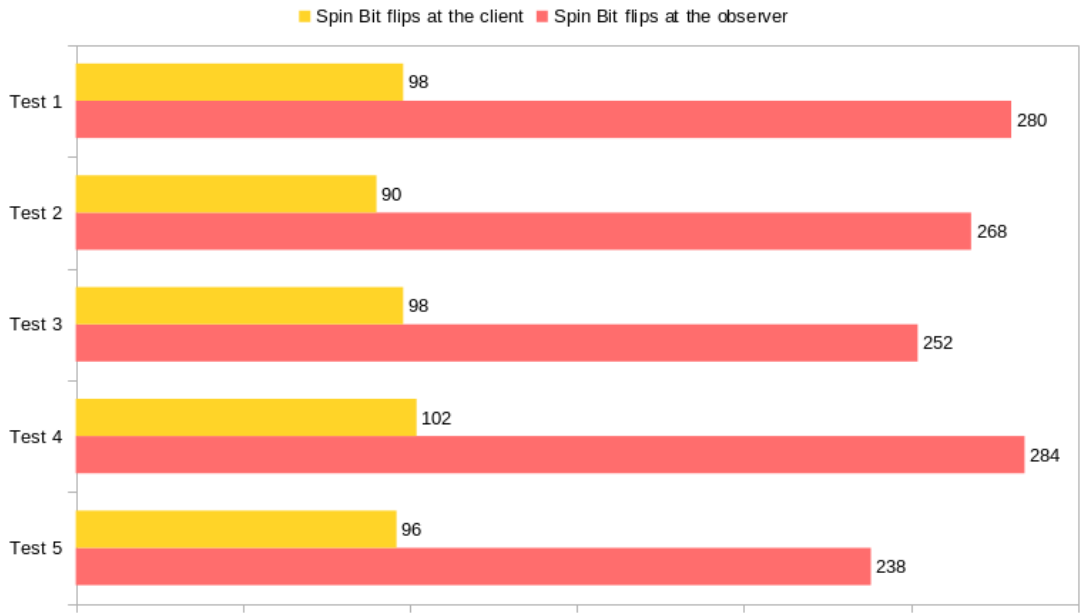

Figure 5.14: Number of Spin Bit flips at the client and at the observer when testing with a packer reordering rate of 25%.

During the tests with a packet reordering rate of 25%, TTE also reported an average of 95 Spin Bit periods with a single packet in it, meaning that the amount of spurious edges is very high in these conditions.

5.3 Overall evaluation and final considerations

The testing and the evaluation conducted during the course of this chapter, show that **the Inside RTT monitoring does a competent job at reflecting the Inside RTT** computed by a client application in the presence

of a continuous flow of traffic going from the client to the server. If the client application can provide a good estimation of the RTT, **the implementation allows a passive observer to monitor the RTT** between two QUIC endpoints with an average error of a few milliseconds: this is the case when using the Chromium browser. Additionally the Inside RTT monitoring technique solves one of the main problems of the traditional Spin Bit, as **it does not require server side reflection**, and it can therefore be used in all instances as soon as it is adopted in client side libraries. The same would be true in case the technique was adopted exclusively in server side libraries, meaning that network operators that want to monitor the performance of QUIC connections would no longer need to rely on the agreement and the cooperation between client and server, like in the case of the Spin Bit.
It is however important to notice that by design the Inside RTT monitoring technique relies on the RTT estimation provided by the client application, and while this value is not part of the implementation, it will play a significant role in the observer's ability to estimate the actual RTT of the connections. Additionally, the Inside RTT monitoring technique suffers from the **same issues of the Spin Bit when the traffic flow is small**, or in the presence of significant packet reordering and packet loss. Finally, QUIC packets are affected by both transport layer and application layer delay, and the traffic flow might be impacted by human interaction: for this reason an adequate filtering mechanism could be useful to the network operator to filter outliers.

Chapter 6

Conclusions

The Inside RTT monitoring technique is a valuable and effective alternative for network operators that want to perform passive network monitoring of QUIC traffic, as it solves the issues of adoption and cooperation that are likely to render the traditional Spin Bit unusable in most real world scenarios. The ability to unilaterally generate a square wave means that once the Inside RTT monitoring techniques is deployed in QUIC libraries, a passive observer can immediately start to measure RTT values that it would not be able to measure otherwise, as even if some libraries were to support the Spin Bit the lack of cooperation from a certain endpoint would be enough to make it useless. Furthermore, the tests conducted during the course of this thesis, show that the provided implementation allows to measure the Inside RTT with a few milliseconds of error, and when using a client like the Chromium browser this translates into the ability for passive observer to measure the actual RTT between two endpoints with precision.
While the Inside RTT monitoring technique solves the most important issues of the Spin Bit, it is affected by some of the same problems when it comes to traffic shape, application layer delay and use of heuristics at the observer. It is however important to notice that these issues are not specific to the algorithm, so it is fair to say that the benefits of this technique outweigh the disadvantages, as the technique and its proposed implementation provide immediate gains for network operators that want to monitor the levels of performance and health in their own network.

6.1 Future research

In the future, it would be interesting to see the Inside RTT monitoring technique implemented in other client side libraries, but also in server side libraries to enhance the capabilities of network operators that have partial visibility of the traffic flows. Furthermore, in Chapter 2 I explained that the users of the client application should explicitly agree to the usage of network monitoring techniques, so in an effort to preserve their privacy a Web browser like Chromium should also contain a section of the user interface that allows users to opt-in and to opt-out of this mechanism.
Considering how the Inside RTT monitoring technique is reliant on a good RTT estimation and on continuous flows of traffic, it is a strong candidate for custom clients made specifically for active network monitoring. Additionally, while the Inside RTT monitoring technique presented in this thesis provides an effective way to measure the RTT, it could bring even more value once it is combined with other explicit flow measurements techniques that take advantage of the Reserved Bits in the header of QUIC packets. The Reserved Bits could also be used to enhance the capabilities of the existing techniques, once they are transmitted in cleartext. For example, a library could support both the Spin Bit and the Inside RTT monitoring technique, and it could use one of the Reserved Bits to indicate to the passive observer which one is in use, based on the willingness of the server to reflect the Spin Bit. Another interesting option would be to use one of the Reserved Bits to indicate an amplification factor for which the Inside RTT has been multiplied before using it as the Spin Bit period: while this would sacrifice a number of measurements, it would also allow to reduce the impact of spurious edges caused by packet loss.
Surely Telecom Italia and Politecnico di Torino will continue to research and develop other network monitoring techniques, in an effort to allow network operators to monitor the performance of QUIC connections. Other protocols that play a significant role in today's Internet should also be subject of future research, as network operators continue to chase the best quality of service possible inside their infrastructures.

Bibliography

[1] Ed. J. Iyengar and Ed. M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. 2021. DOI: 10.17487/RFC9000. URL: https://www.rfc-editor.org/info/rfc9000 (cit. on pp. 1, 4, 26, 42).

[2] W3Techs. *Usage statistics of HTTP/3 for websites*. URL: https://w3techs.com/technologies/details/ce-http3 (cit. on pp. 1, 4).

[3] Mozilla Developer Network. *Evolution of HTTP*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP (cit. on p. 3).

[4] Haxx. *TCP head of line blocking*. URL: https://http3-explained.haxx.se/en/why-quic/why-tcphol (cit. on p. 4).

[5] URL: https://caniuse.com/http3 (cit. on p. 4).

[6] Ed. M. Thomson and Ed. S. Turner. *Using TLS to Secure QUIC*. RFC 9001. 2021. DOI: 10.17487/RFC9001. URL: https://www.rfc-editor.org/info/rfc9001 (cit. on p. 5).

[7] M. Kühlewind and B. Trammell. *Manageability of the QUIC Transport Protocol*. RFC 9312. 2022. DOI: 10.17487/RFC9312. URL: https://www.rfc-editor.org/info/rfc9312 (cit. on pp. 9, 21).

[8] URL: https://github.com/fabiobulgarella/quic-go/commit/15f89f861cd2198c081dda01a515b5a4d11ce43a (cit. on p. 11).

[9] M. Di Natale, R. Sisto, G. Marchetto, and M. Cociglio. «QUIC performance monitoring: implementation of Spin Bit in Chromium». MA thesis. Politecnico di Torino, 2023. URL: https://webthesis.biblio.polito.it/secure/26911/1/tesi.pdf (cit. on pp. 11, 22).

[10] URL: https://github.com/litespeedtech/lsquic (cit. on p. 11).

[11] M. Cociglio, A. Ferrieux, G. Fioccola, I. Lubashev, F. Bulgarella, M. Nilo, I. Hamchaoui, and R. Sisto. *Explicit Host-to-Network Flow Measurements Techniques*. draft-ietf-ippm-explicit-flow-measurements-07. 2023. URL: https://datatracker.ietf.org/doc/draft-ietf-ippm-explicit-flow-measurements/ (cit. on p. 11).

[12] Fabio Bulgarella, Mauro Cociglio, Giuseppe Fioccola, Guido Marchetto, and Riccardo Sisto. «Performance Monitoring of QUIC Communications». In: (July 19, 2019), pp. 8–14. URL: https://dl.acm.org/doi/10.1145/3340301.3341127 (cit. on p. 11).

[13] URL: https://www.openhub.net/p/chrome/analyses/latest/languages_summary (cit. on p. 11).

[14] URL: https://chromium.googlesource.com/chromium/ (cit. on p. 11).

[15] URL: https://quiche.googlesource.com/quiche (cit. on p. 12).

[16] URL: https://github.com/EricssonResearch/spindump (cit. on p. 15).

[17] M. Cociglio, M. Nilo, F. Bulgarella, and G. Fioccola. *User Devices Explicit Monitoring*. draft-cnbf-ippm-user-devices-explicit-monitoring-04. 2022. URL: https://datatracker.ietf.org/doc/draft-cnbf-ippm-user-devices-explicit-monitoring/ (cit. on pp. 16, 17).

[18] Cociglio Mauro. «Transmission of a measurement result through a packet-switched communication network». 2023. URL: https://worldwide.espacenet.com/patent/search?q=WO2023099372 (cit. on pp. 17, 18).

[19] URL: https://github.com/google/quiche/compare/main...fxbrit:quiche:internal-spin-bit (cit. on p. 22).

[20] The Chromium Projects. *Playing with QUIC*. URL: https://www.chromium.org/quic/playing-with-quic/ (cit. on p. 30).

[21] URL: https://github.com/fxbrit/spindump-docker (cit. on p. 31).

[22] Amazon. *Amazon WorkSpaces client network requirements*. URL: https://docs.aws.amazon.com/workspaces/latest/adminguide/workspaces-network-requirements.html (cit. on pp. 39, 42).

[23] Microsoft. *Stream Classification in Call Quality Dashboard*. URL: https://learn.microsoft.com/en-us/microsoftteams/stream-classification-in-call-quality-dashboard (cit. on p. 42).

[24] Jana Iyengar and Ian Swett. *QUIC Loss Detection and Congestion Control*. RFC 9002. 2023. DOI: 10.17487/RFC9002. URL: https://www.rfc-editor.org/info/rfc9002 (cit. on p. 42).

[25] HTTP Archive. *Page Weight*. URL: https://almanac.httparchive.org/en/2022/page-weight (cit. on p. 50).