POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

A new One-Way Delay measurement system for QUIC protocol

Candidate:
Saverio Massano

Supervisors:
prof. Riccardo Sisto
prof. Guido Marchetto

Telecom Italia Tutors:
dott. Mauro Cociglio
dott. Massimo Nilo

Academic Year 2020 - 2021

Abstract

The Quick UDP Internet Connection protocol (or QUIC) is a new transport protocol firstly introduced by Google in early 2010's for its needs and recently standardized by the IETF via the RFC 8999, 9000, 9001 and 9002 published in May 2021. The aim with this protocol is to find a valid, more performing substitute for TCP protocol in all its internet implications finding then a new base for modern HTTP protocol versions. A brand new protocol would be a great occasion to include, as a native feature and in contrast to the TCP, one or more performance measurement features in order to have a built-in network monitoring mechanism to check the network health's status with also positive effects on the QoS management aspect. As a consequence, current official IETF QUIC RFCs provide a reserved bit in (of a certain type) QUIC header packet exploitable in a performance measurement's algorithm. Goal of this thesis is to provide an in-depth study of a new algorithm, proposed by Telecom Italia, able to passively measure the One-Way Delay of a QUIC connection. A possible implementation of the theoretical mechanism has been realized and then analyzed by testing its behavior in different emulated network conditions. Finally, consideration about the algorithm's limitations have been discussed together with some adoptable countermeasures and optimizations.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

In a world dominated by an increasing number of interconnected devices and users, it's fundamental to constantly monitor and analyze the whole internet's infrastructure and keep it updated.

But this is not an easy task. Nowadays, the complexity of internet is so high that consistent updates on the TCP/IP stack, at the base of the internet's core, would require a giant amount of money and huge difficulties in coordination and interoperation between all the ISPs involved. The result is a technological standoff lasting almost 20 years and causing the necessity of found new ways and workarounds to use internet for every modern purposes.

Hard time also for the performance monitoring and network management aspects: the TCP/IP stack was born with no built-in functionalities of this kind. Analyze the internet health's status has almost always implied the use of different tricks, like compute the RTT values by tracking the TCP's handshake packets (the differences between the sending and receiving packet timestamps). Unfortunately, often these methodologies are complex and don't provide enough accuracy in all the possible network's conditions.

It's now clear that something new must be found.

A breath of fresh air might be brought by the QUIC protocol, a new transport layer protocol recently standardized by the IETF via the

RFC 8999[1], 9000[2], 9001[3] and 9002[4] published in May 2021. The aim with this protocol is to find a valid, more performing substitute for TCP protocol in all its internet implications finding then a new base for modern HTTP2[5] and the upcoming HTT3[1][6] protocol versions.

A brand new protocol would be a great occasion to include, as a native feature and in contrast to the TCP, one or more performance measurement features in order to have a built-in network monitoring mechanism useful to analyze the network health's status with also positive effects on the QoS management aspect. As a consequence, current official IETF QUIC RFCs provide a reserved bit in (of a certain type) QUIC header packet exploitable in a performance measurement's algorithm.

The first algorithm of this kind is the **Latency Spin Bit** implemented by Brian Trammell[7] from the ETH of Zurich and thought to be used for network latency passive measurements. Unfortunately, due to the simple idea at the base of the algorithm, it is able to provide reliability and accuracy only in case of optimal network's conditions.

For this reason, many other companies and universities tried (and continue up to the present) to propose new and more sophisticated passive measurement systems.

[1]In earlier IETF's drafts, it used to be called "HTTP over QUIC"

1.2 Goals and structure of the thesis

Goal of this thesis is to provide an in-depth study of a new algorithm, proposed by Telecom Italia, able to passively measure the One-Way Delay of a QUIC connection. A possible implementation of the theoretical mechanism has been realized and then analyzed by testing its behavior in different emulated network's conditions using the instruments inside the Telecom Italia's laboratory in Turin, Italy.

Below, the chapter's list with a short overview:

- **Chapter 2:**
  This chapter provide a more in-depth description of the limitations affecting the internet core's protocols and the reasons why QUIC protocol would be a valid substitute. An overview of the last official QUIC protocol version is then reported with a focus on its key characteristics. Following, some basic notions on network metrics and network performance measurements are provided. At the end, the Trammell's latency spin bit algorithm is explained along with its limitations.

- **Chapter 3:**
  The theoretical mechanism of the new One-Way Delay algorithm is described along with an overview of the on-path observer's role and what kind of measurement samples it can retrieve. Follow an introduction to some optimization aspects able to increase the overall accuracy of the measurement system. Finally, a pseudo-code implementation is provided with some explained details.

- **Chapter 4:**
  The fourth chapter provide more details about the endpoints software and the observer software with a focus on how this last can manage every tracked packet in according to the algorithm's requirements. Follow a quick explanation of the Telecom Italia laboratory's testing environment and how it has been configured for the tests. The remaining part of the chapter is dedicated to the report of all the meaningful collected measurement samples with possible considerations on them.

- **Chapter 5:**
  The two T0's setup phases, introduced in chapter 3, are discussed more in-depth with a possible code's implementation and considerations based on what observed trying to execute them in the laboratory's environment.

- **Chapter 6:**
  The conclusions chapter, with an overall recap of the work done and some final observations.

Chapter 2

Background

Since the standardization of the TCP/IP transmission suite in 1982, internet infrastructure's diffusion recorded a steady increment all around the world. Moreover, with the birth of the Web and the standardization of HTTP protocol in the 90s, tons of new services and applications running over the internet have been created resulting in an additional drastic increase of the data traffic volumes.

Unfortunately, the evolution and growth of the application layer is not followed by a corresponded evolution of the underlying layers. In fact, the Internet's core protocols remained almost the same across the last forty years leading to a pletore of tricks and hacks in order to adapt them for all the modern necessities.

But why this is happened?

2.1 Struggling to keep updated internet's core protocols

Despite the central role of the TCP protocol in internet infrastructure, since 1993 no big changes have been done to it. Different extensions has been released and standardized but de facto, just few of these are effectively implemented in a protocol version. The standard TCP packet format provide an option space of 40 Byte exploitable for additional functionalities. Unfortunately, part of this field is already occupied to provide useful basic information, like timestamp and congestion window Scale. The remaining part of this field is just not enough for every needs.

At this point, an idea could be change the original TCP packet format to gain more space, but this is quite impossible to achieve on a such a large scale. Internet infrastructure is disseminated of middleboxes, i.e. devices that manage the known TCP version at an hardware level. Change the TCP packet's format would means change the physical hardware stack of almost all this devices with a giant economic effort for the ISPs and many difficulties in coordinating the evolution between them. Something similar is valid also for the protocol's implementations in the consumer Operation Systems, where TCP is implement at kernel level. New kernel iterations require again time and money. All these constrains are just some of the obstacles that cause the so-called **Internet ossification** problem.

2.1.1 TCP is not tailored for new HTTP protocol versions

From the HTTP/2 version, IETF delivered the ability to multiplex different HTTP requests within the same TCP connection, allowing a smarter and more efficient management of network's bandwidth. Moreover, multiplexing would theoretically allow to block just one data stream in case of packets loss or packets reordering, i.e. only the stream the lost or the reordered packet belong to, continuing to process all the other streams.

Unfortunately, TCP causes a bottle-neck effect called **head-of-line blocking**. All the different streams sent in the same connection are seen as a unique, opaque stream of packets and a real streams multiplexing at the transport layer is not possible. In case of packet lost or reordered, TCP has not chance to detect the specific stream to block during the recovery phase with the result that all the streams will be blocked. But this is just an example, the concept is that TCP cannot provide all the support needed for the new mechanisms thought in recent years to enhance internet's performances.

2.2 The QUIC protocol

The Quick UDP Internet Connection protocol (or QUIC) is a new transport protocol, born from decades of network's knowledge, with the aim to find a valid, more performing substitute to TCP protocol in all its internet implications, i.e. it would like to be a possible solution to the ossification problem cited before. It has been developed firstly by Google in early 2010's and just a couple of year later, IETF started to study it's own version in order to write standard guidelines and extend its application area.[1].

2.2.1 Main characteristics

To circumnavigate the ossification problem, QUIC protocol is thought to exploit the UDP transport protocol as a substrate. In this way, middleboxes will not affect or discard QUIC packets[2] and the protocol could remain compatible with OS kernels and legacy devices. Keep in mind that UDP is a not reliable protocol by default and this continue to be true. It's the QUIC protocol that need to guarantee reliability. For this reason, the protocol re-implements lot of TCP functionalities like loss recovery and congestion control. Moreover, it includes a stream-based approach allowing to send multiple streams in the same QUIC connection with the possibility, in contrast to what happens with TCP, to manage them independently exploiting completely the multiplexing abilities of HTTP/2 and avoiding the TCP's head-of-line blocking phenomenon.

Another key aspect of QUIC is the security-by-default approach, radically different to the principle of TCP. In fact, while TCP packets are not authenticated nor encrypted by default[3], QUIC packets are

[1]The QUIC protocol developed by Google is known as gQUIC while the counterpart standardized by the IETF is called iQUIC. Today, the two versions are significantly different and no more compatible from each other.

[2]usually internet's middleboxes affect only TCP packets.

[3]TLS additional layer[8] can provide security for TCP

100% authenticated and almost completely encrypted.

Speaking about the handshake procedure, latest official QUIC protocol version integrates all the functionalities of the TLS 1.3 protocol[3]. In this way, it can rely on a well-known and accepted security layer avoiding, at the same time, to develop of an ad Hoc new protocol as has happened for the gQUIC versions.

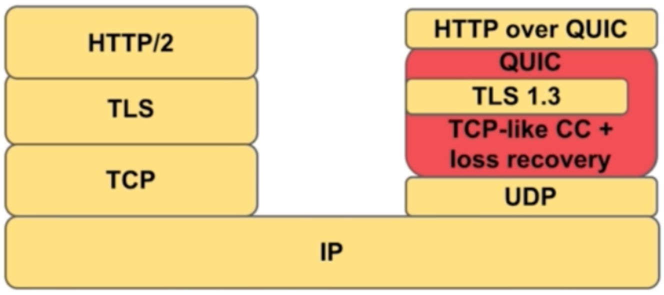


Figure 2.1: Comparison between TCP/IP "classic" stack and current QUIC stack

2.2.2 QUIC headers

QUIC protocol is designed to use packets with two different header's format: the long header (figure 2.2) and short header (figure 2.3).

Long header

Packets with long header are used during the handshake's procedure. In this category there are the packets: Initial, Handshake, 0-RTT and Retry.

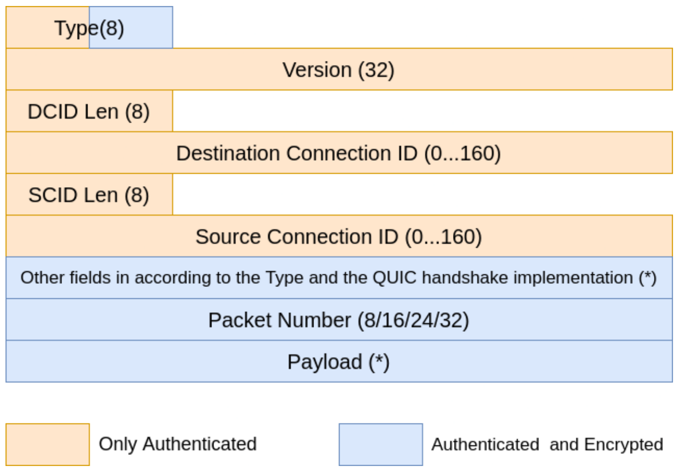

Type(8)
Version (32)
DCID Len (8)
Destination Connection ID (0...160)
SCID Len (8)
Source Connection ID (0...160)
Other fields in according to the Type and the QUIC handshake implementation (*)
Packet Number (8/16/24/32)
Payload (*)
Only Authenticated
Authenticated and Encrypted

Figure 2.2: A simplified graphical representation of the short header format. Between the round brackets, the expected size in Byte.

- **Flags Byte:** This is the first byte in a long header packet. The MSB is set to 1. The second bit is the fixed bit with a value of 1. The third and fourth bits are called **Type bits** and represent the long header packet type. The last four bits are encrypted and their values depend on the packet's type.
- **Version:** This field contains the QUIC version information and determines how the following packets byte need to be interpreted.
- **Destination Connection ID Length:** This field contains the length of the next field.

- **Destination Connection ID:** Field containing the unique ID value given to the destination endpoint in a QUIC connection.
- **Source Connection ID Length / Source Connection ID:** Same things as the previous two fields.
- **Payload:** The encrypted payload, containing the data and control frames.

Short header

Packets with short header are sent after the QUIC connection is created and the session encryption keys are available. The Official QUIC version defines only the **1-RTT** packet with the short header, used to exchange data frames and ack frames.

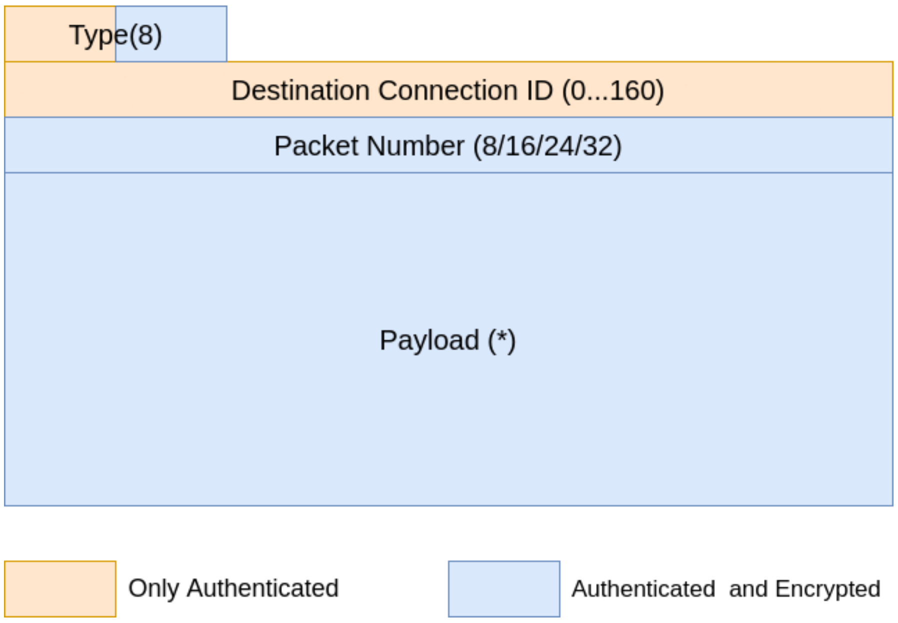

Type(8)
Destination Connection ID (0...160)
Packet Number (8/16/24/32)
Payload (*)
Only Authenticated
Authenticated and Encrypted

Figure 2.3: A simplified graphical representation of the short header format. Between the round brackets, the current expected size in Byte.

- **Type Byte:** The counterpart of the long header's Flags Byte. Currently, the IETF RFCs provide the format **01SRRKPP**: the MSB bit is set to 0. The second bit is the fixed bit with value equal to 1 (as for the long header). The third bit is the Spin bit (**S**) (discussed later in this chapter). The fourth and fifth bit are Reserved bits (**RR**) for future purposes. The sixth bit is the Key bit (**K**), useful to retrieve the used encryption keys. At the end, there are the Length bits (**PP**) used to identify the packet's length. Currently, only the first three bits are unencrypted (but authenticated).
- **Packet Number Length:** Contains the encoded packet number field's length, that can be from 1 to 4 Byte long.
- **Destination Connection ID:** Field containing the unique ID value given to the destination endpoint in a QUIC connection (same thing as long header)
- **Packet number:** Field reporting the current packet number.
- **Payload:** The encrypted payload, containing data and control frames.

2.3 Network Performance measurement concepts in a nutshell

A key aspect for a ISP is to analyze the behavior and quality of the controlled network's portion. This operation is fundamental in order to detect and fix issues but also to understand if the infrastructure is able or not to support the whole expected data traffic volume deciding then what kind of enhancement could be done.

2.3.1 Network metrics

Networks performances are affected mainly by three aspects[9]:

- **Latency**: Also called **network delay**, it represents the time required for a packet to travel from a source endpoint to the destination endpoint. More in detail, this metric is the sum of different delay types:
  - **Processing delay**: total time required to elaborate a packet from all the network devices between source and destination endpoint.
  - **Transmission delay**: time required for a packet to cross the link between the two endpoint, it depends on the link's bandwidth and the data traffic volume.
  - **Propagation delay**: time required by the physical signal to travel inside the physical means of transmission.
  - **Queuing delay**: total time spent by a packet in buffers along the network path.
- **Throughput**: The current amount of data transmitted over an internet connection link in a given amount of time.
- **Packet loss rate**: Percentage of packets lost during a data transmission, i.e. packets sent by a source host but never received by the destination host.

2.3.2 Measurement approaches

Two way can be followed to measure one or more network's metrics[9]:

Active measurement

Probe packets are sent through the network in addition to nominal data traffic. The biggest advantage with this method is the possibility to have packets flow also in absence of spontaneous traffic. It's also possible to mark the probe packets in order to easily recognize them between the other packets. At the same time, additional packets increase traffic volume and cause the consumption of more network's resources. These two factors can degrade network's performance and affect negatively the measurement samples retrieved.

Passive measurement

In contrast with the previous approach, no packets are added to the expected network traffic. The existing packets are exploited in some ways in order to compute measurements. For example, RTT values can be measured by means of the handshake's packets of a TCP connection, i.e. taking in account the sending and receiving packet's timestamps on the endpoints. Although the traffic volume is not increased, is more difficult to use the nominal transmission's packets. More complex algorithms and more expansive hardware are needed for the purpose.

2.4 The Latency Spin Bit

Current official IETF RFCs include in the short header a so-called **Spin Bit**, a reserved, not encrypted bit thought to be exploited in a native measurement performance algorithm.

The first simple algorithm of this kind has been proposed by the researcher of Zurich's ETH University Brian Trammel in one of his drafts [7]. This marking algorithm[10] works in the following way: Client and server contains an internal spin bit value used to **mark**[4] or not the outgoing packets.

At the beginning of every QUIC connection, both endpoints set this value to 0. Then, client and server manages the spin bit in different ways:

- **Client:** when it receives a short header packets with an higher sequence number than the last received, it toggles its spin bit value to the opposite value stored inside the received packet header.
- **Server:** when it receives a short header packets with an higher sequence number than the last received, it toggles its spin bit value in according to the value stored inside the received packet header.

The result is the generation of a periodical square wave signal exploitable by an on-path observer placed between the two endpoints and able to measure the RTT value by measuring the length of this detected signal[5]. Notice that by paying attention to the packet number received, is possible to avoid the reflection of a reordered packets sequence and then filter the signal from the **spurious edges** (see Section 2.4.1).

Moreover, an observer can works in one-way mode (it can only detects packets travelling in one direction, from client to server or vice versa) or in two-way mode (it is placed symmetrically on both upload and download directions). In this second case, its also possible to compute

[4]the operation of toggle to 1 the header's spin bit

[5]It retrieves the RTT by computing the difference in time between the first incoming marked packet timestamp and the first incoming not-marked packet timestamp.

the half-RTT components by measuring the delay between the square wave edge[6] observed in the upload direction and the square wave edge detected in the download direction (and vice versa).

2.4.1 Limitations of the algorithm

Due to the simplicity behind the algorithm, its only possible to retrieve RTT values with sufficient accuracy in optimal network conditions. In fact, there are some factors, called **network impairments**, that would degrade seriously the measurement system's accuracy:

Delays

RTT and half-RTT values are computed via the distance in time of two square wave's edges. What could happen is that an observer overestimate the RTT value due to a longer square wave's length (the two wave's edges are more distant in time due to the additional delay), maybe because of the activation of the QUIC congestion control or in presence of traffic holes.

Packet reordering

Client and server checks the correctness of the packet's number sequencing every time a packet is received. If there's no reordering events, endpoints toggle the internal spin bit value accordingly to the algorithm's logic continuing to mark (or not) the packets as required. Notice that the sequence number is readable only after removing the encryption layer over the packet's field where it is embedded, operation possible only for the two endpoints involved in the connection.

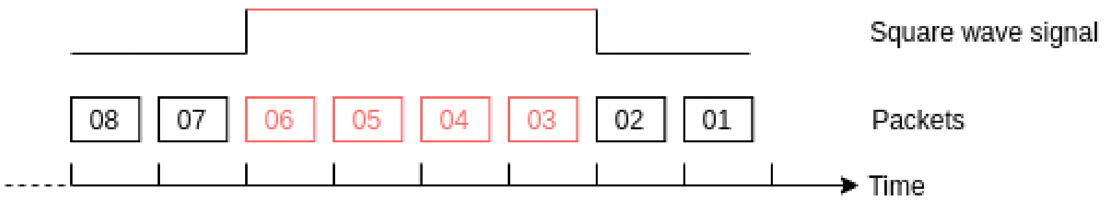
[6]An edge represent the time instant at which the square wave change its status from high to low and vice versa

This means that an observer cannot access the packet number's field. Without any additional precaution there is the possibility to detect one or more **Spurious square wave edge** [7]with the result of an higher number of underestimated RTT samples.

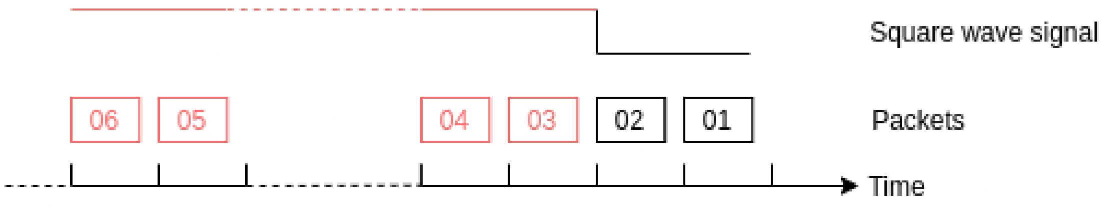
Packet Loss

A packet could be lost somewhere between the source host and the destination host. Also this event could affect negatively the latency spin bit accuracy. Packets lost on the square wave edges, would cause an underestimation of the RTT sample (the square wave is shorter than the expected one).

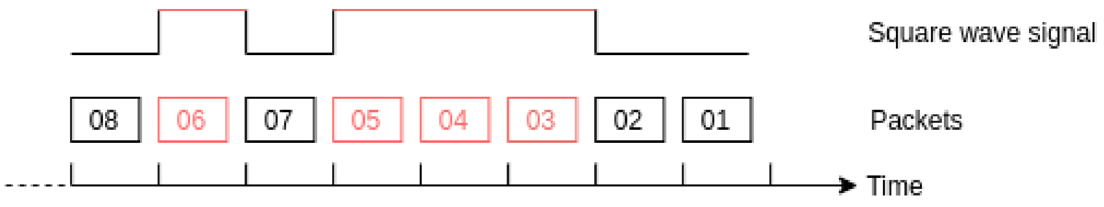
[7]If the reordered packets are on the border of the square wave, one or more marked packets could be detected after one or more not-marked packets



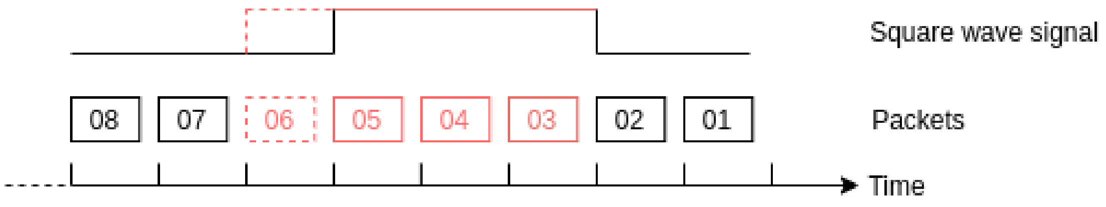
(a) No impairments affecting the square wave generated by the marked packets.



(b) An addition delay is present between the marked packets, maybe due to a traffic hole. The observer overestimate the RTT value.



(c) The reordered packet 06 generates a spurious edge detected by the observer.



(d) A lost marked packet at the square wave's border causes an underestimation of the current square wave's length and then an underestimation of the RTT sample.

Figure 2.4: Graphical representation of a square wave in different network's conditions as described above.

Chapter 3

The One-Way Delay algorithm

3.1 Preview

From Chapter 2, is clear that the Latency Spin Bit is not an accurate algorithm in a wide range of network's scenarios. Accordingly, many other performance measurement algorithms have been proposed to the IETF by universities and research group. In this research context is located Telecom Italia, that already proposed valid new performance measurement systems also in collaboration with the Politecnico di Torino University.[11] [12]

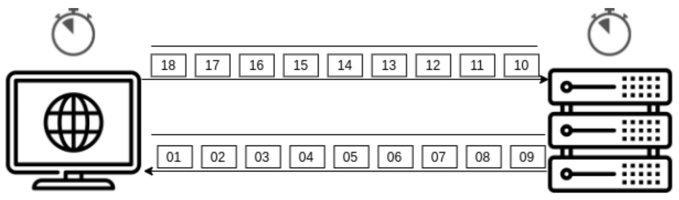
In this chapter, a theoretical explanation of the One-Way Delay algorithm, one of the latest algorithm proposed by the company to the IETF, is provided with a possible pseudo-code implementation.

3.2 One-Way delay algorithm overview

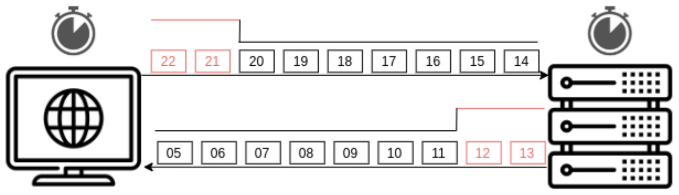
It's possible to divide the algorithm in two main phases: **Generation Phase** and **Termination Phase**. Optionally, it is possible to consider an initial **setup phase** before the first generation phase starts. Both client and server needs to be strictly synchronized to each other externally [1]. A QUIC connection is divided in measurement periods, everyone long **T0** , i.e. everyone long a value that need to be a divider of number 1000 (if T0 is expressed in milliseconds, microseconds or nanoseconds) or a divider of number 60 (if T0 is expressed in seconds). In any case, the T0 value must be decided before the first generation phase starts and must be the same on both client,server and observer. An internal spin bit value is stored for every QUIC connection on both endpoints and used to properly mark the outgoing packets but also used to keep track of the current algorithm's phase in which the endpoint is. At the beginning, both endpoint initialized the spin bit to 0.

When a new period starts at T_i instance, both client and server enters in the generation phase and toggle to 1 their own spin bit value. From this moment, they will mark every outgoing packets generating in this way a square wave signal. Then, when the first marked packet is detected by the endpoint in generation phase and it has the highest sequence number received until that moment, the endpoint toggles back to 0 it's internal spin bit value entering in the termination phase and stopping then to mark the outgoing packets. Now it will wait for new incoming T_i instance.

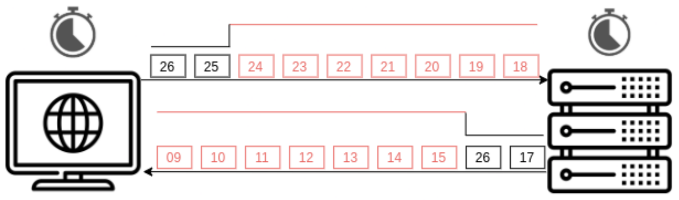
[1]For example via NTP time protocol.



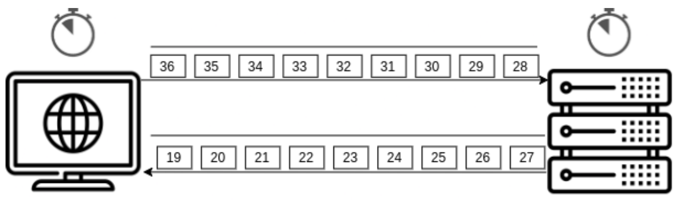
(a) Both endpoints are waiting for next T_i. No marked packets until at least that time instance.



(b) T_i arrives, both endpoints enter in Generation Phase. A square wave is present on both directions.



(c) The square wave is detected by the opposite endpoint that enters in termination phase and stop to mark outgoing packets. The square waves terminate.



(d) T0 is long enough to avoid overlapping of contiguous periods. Endpoints are waiting for next T_i. No square waves on both links until that time instant.

Figure 3.1: Graphical representation of the basic One-Way Delay's marking mechanism in absence of any impairment factors. The sequence shown is cyclically repeated until the end of the QUIC connection.

3.3 The Period T0

Due to the periodical nature of the measurement methodology, it's important to decide and set on both endpoints (and also on the observer) the length of time that every period lasts. A valid T0 value's constraint would be $T0 > RTT_{max}$. In order to be sure that the previous period is already terminated and no marked packet are still travelling between the two endpoints, is better to consider the more stronger constraints $T0 > 2OWD_{max}$, (or $T0 > 2RTT$ if no previously collected values of OWD are available), or even more $T0 > 3OWD_{max}$ (or $T0 > 3RTT$). In addition, as already said, T0 need to be a divider of number 1000 or 60 (in according to the time unit considered).

3.3.1 Compute the T0 value

There could be two ways to measure T0 value:

- **T0 "A priori":** The easiest way to decide a sufficient higher value for T0 is to manually set it on the involved actors. A reasonable, safe value could be $T0 = 1$ second with the only drawback to have a low number of measurement's samples performed in a single connection.

- **Setup Phase:** An additional initial phase for the OWD algorithm can be considered. The basic idea is to compute the RTT value between client and server and then to compute T0 accordingly to the constraints. In this way, there is the possibility to set a more tailored T0 value with the aim to retrieve more samples. This phase would also allow to perform a rough synchronization between client and server, reducing the possibility that one of the endpoints will start the first generation phase a period before the other endpoint. More details about this topic will be provided in chapter 5.

3.4 The on-path Observer

The endpoints marking mechanism alone is not enough, an on-path observer between client and server must exploits the square waves on the links in order to retrieve the measurement samples as allowed by the OWD methodology.

The biggest challenge with an observer software is the impossibility for it to access to all the information stored in a QUIC packet due to the security-by-design approach described previously and the lack of the encryption keys[2].

Fortunately, for the One-Way Delay algorithm only the the third MSB is involved in the marking process as for the Latency spin bit (see Section 2.4). The observer should know the same T0's value known also by the endpoints. In according to the strategy adopted, this value is stored "A priori" or after partecipating to the implemented setup phase with client and server.

The observer OWD algorithm is then able to compute two types of measurements:

Half-OWD measurement

The observer computes the delay between the beginning of the current measurement period T_i and the detection time instance of the square wave sent by the endpoint. If the observer is placed on the upstream direction, it is able to compute the uplink half-OWD. Vice versa, an observer sniffing packets on the downlink direction can compute the downlink half-OWD. It's clear that half-OWD measurement samples can be retrieved only if the observer is synchronized with the two endpoint knowing the current T_i period's starting time instance and then also T0's length.

[2]only the endpoints partecipate to the QUIC connection handshake

OWD opposite measurement

Firstly, the observer is place on the upload link or on the download link waiting for the square wave edge coming from one endpoint. When it detects the first marked packet coming from the client (if placed on the uplink) or the server (if placed on the downlink), it stores the detection timestamp. Then, the square wave terminates and from the same endpoint the observer will receives the first not-marked packet. It stores the second detection timestamp. The OWD measurement sample is then the distance in time between the two stored timestamp. As a constraint, the observer stores all the OWD measurement samples with a value not around the 50% T0 length's value.

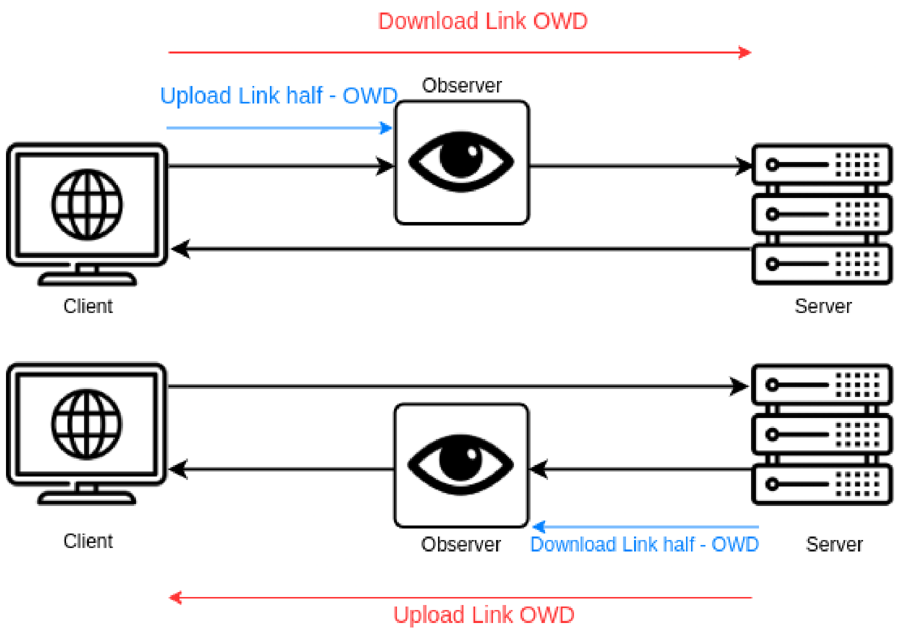


Figure 3.2: The two possible OWD measurement configurations in according to where the observer is placed

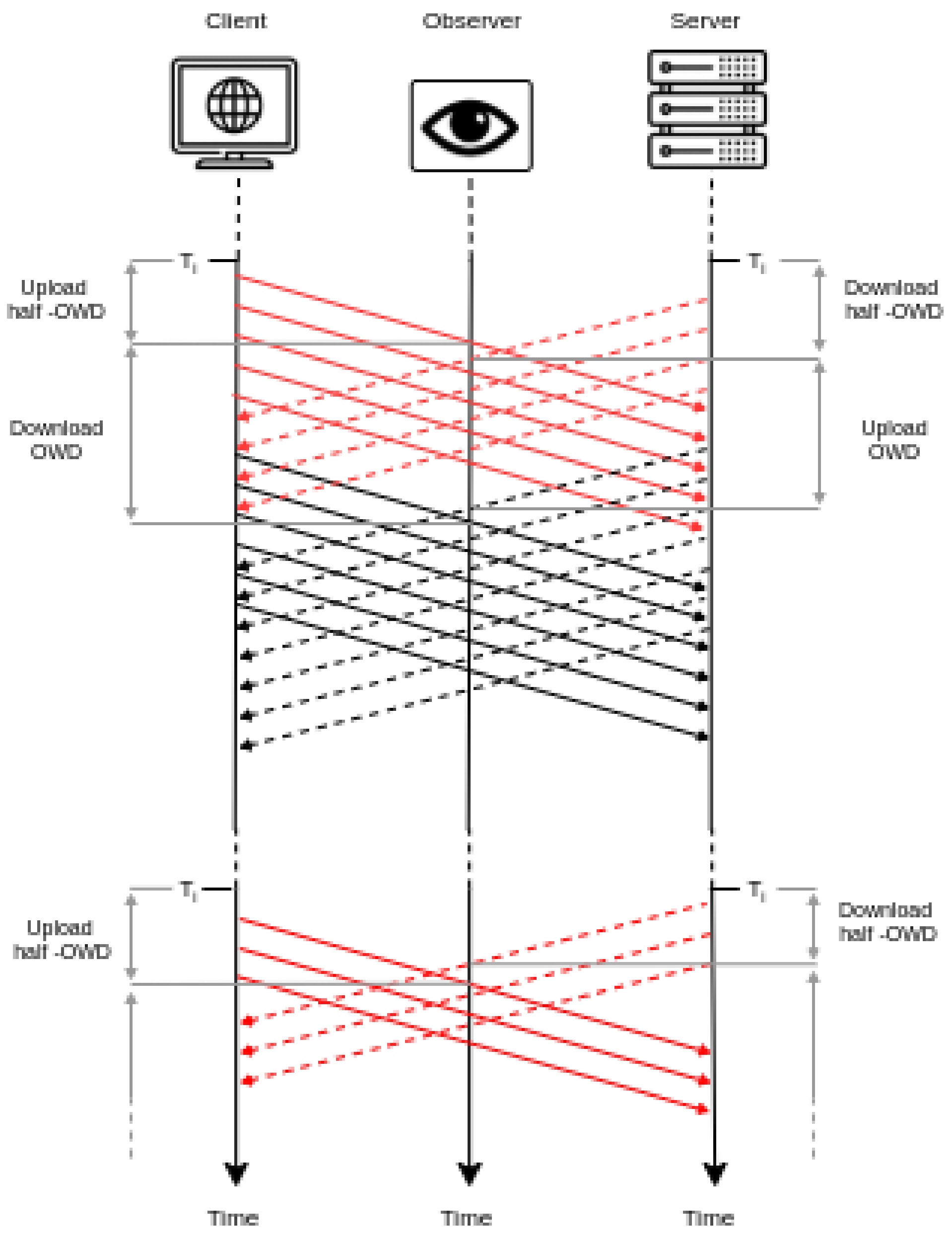


Figure 3.3: Graphical representation of half-OWD and OWD measurements exploiting the incoming marked and not-marked packets.

3.5 Algorithm's optimizations

The basic algorithm's mechanism explained could be enough in not-impaired network conditions. But as already said speaking about the Latency Spin Bit (see Chapter 2.4), a QUIC connection could be affected by one or more factors and for this reason a measurement algorithm should provide some countermeasures.

3.6 The Elapse interval

Spontaneous traffic is not always available in proximity of a T_i time instant. A Traffic hole in this case could add an unpredictable delay to the first marked packet sent by an endpoint in generation phase and then detected by the observer. As a consequence, the observer will overestimate the half-OWD sample. In the same way, a traffic hole could happen in proximity of a termination phase. The first not marked packet sent by an endpoint could be affected by an unpredictable delay. As a consequence, the observer computing the OWD sample, will receive the first not-marked packet with an unpredictable delay resulting again in an overestimation of the current OWD sample.

A possible countermeasure could be the addition of a time threshold to decide if enter in the generation phase and if enter in the termination phase. When T_i instant comes, an endpoint enters in generation phase if and only if the first outgoing packet is sent at T_{out} between T_i and $T_i + E$, where E represents the Elapse Interval. If the condition is true, endpoint toggles the internal spin bit to 1 starting the square wave signal. Otherwise, the endpoint will wait for next T_i instant without mark any packet. Instead, speaking about the the termination phase, it is triggered on the endpoint in generation phase if and only if the first outgoing packet is sent at T_{out} between the arrival timestamp of the first incoming marked packet T_{in} and $T_{in} + E$ time instant. If the condition is NOT true, the endpoint leaves its internal spin bit to 1 reflecting then the square wave and continuing to mark outgoing packets.

An observer computing the OWD sample, will receive marked packet at least until half of the T0's length and the current sample will be discarded as discussed earlier.

It's clear that in both cases, another termination condition need to be adopted in addition to the detection of the opposite square wave. In the first case, if only one of the endpoints enter in generation phase, it will never receive any marked packets from the opposite side and there will never the condition to try to enter in termination phase[3]. Instead, in the second case, if the termination phase is aborted, the endpoint still in generation phase will continue to mark packets indefinitely.

[3]Important to keep in mind is that an endpoint, with spin bit to 0, will not toggle its value to 1 also if a marked packet is detected.

3.6.1 The half period timeout

When an endpoint is still in generation phase after $T_{\text{Timeout}} = T_i + ½ T0$, then it stops to mark outgoing packet forcibly.

More in details, T_{Timeout} must be half of period's length for a couple of reason:

- In order to leave the time to compute the OWD sample:

$$T_{\text{Timeout}} > OWD_{\text{max}}$$

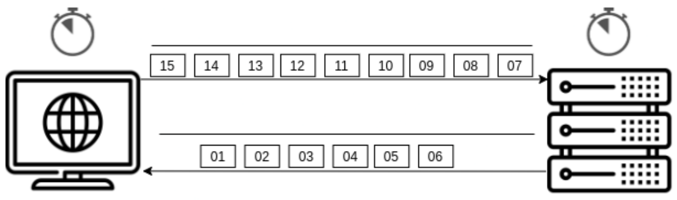
- In order to avoid previous period's marked packet detected by the observer after the successive Ti instance that could cause an anticipated termination phase[4]:

$$T0 > (T_{\text{Timeout}} + OWD_{\text{max}}) => T0 > (½ T0 + OWD_{\text{max}}) => T0 > 2OWD_{\text{max}}$$

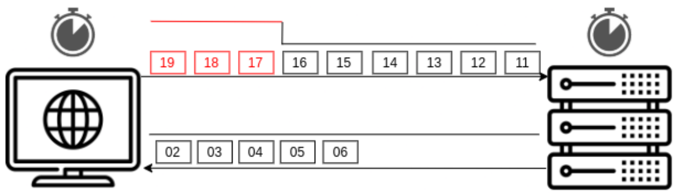
Considering this optimization, it's possible to define acceptable error ranges for both OWD and half-OWD measurement values:

- Half-OWD $\text{Error}_{\text{max}}$ = Elapse Interval ± client-server sync error.
- OWD $\text{Error}_{\text{max}}$ = 2∗Elapse Interval ± client-server sync error.

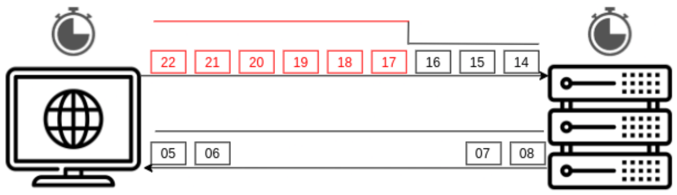
[4]In addition, the termination phase happens if only if a marked packet is detected after a not-marked one paying attention to their sequence number.



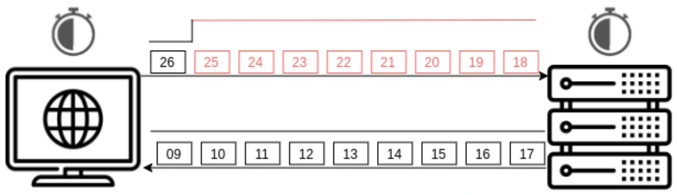
(a) Before the T_i, the client is sending some packets while server register a traffic hole after it has sent some other packets.



(b) T_i arrives, suppose that the client has sent the first packet in the interval between T_i and $T_i + E$ and then enters in generation phase. The server is still not sending packets.



(c) The server starts sending packets but the sending timestamp of the first packet is outside the acceptable interval. Server doesn't enter in generation phase for the current period.



(d) After the 50% of the T0 value, T_{Timeout} goes off and client stop marking outgoing packets.

Figure 3.4: Graphical representation of the violation of the Elapse interval constraints in proximity of the generation phase server-side. Similar situation for the termination phase.

3.6.2 The Waiting Interval

Packet reordering is a big threat for an observer. If this event happens, the observer could interpret a spurious edge as the end of the square wave resulting in the underestimation of the OWD sample. Moreover, observer will detect more than one square wave in the same period, saving multiple underestimated OWD samples. Due to the impossibility for an intermediate node to access to the sequence number field, a possible trick could be the so-called **Waiting Interval (W)**. When the first marked packet is detected, the observer starts a timer[5]. During this length of time, if not-marked packet are detected, they are not considered as the termination of the squared wave, rejecting de facto every possible spurious edges. Only after the timeout goes off the observer will consider the first not-marked incoming packets as the end of the wave computing then the OWD value.
In order to avoid overestimation of the OWD, the value of the waiting interval must respect the constraint:

$$\mathrm{W} < \mathrm{OWD}_{\mathrm{min}}$$

Clearly, the longer is the waiting interval's length, the higher is the filter effect against spurious edges.

[5]The timer duration is set a priori.

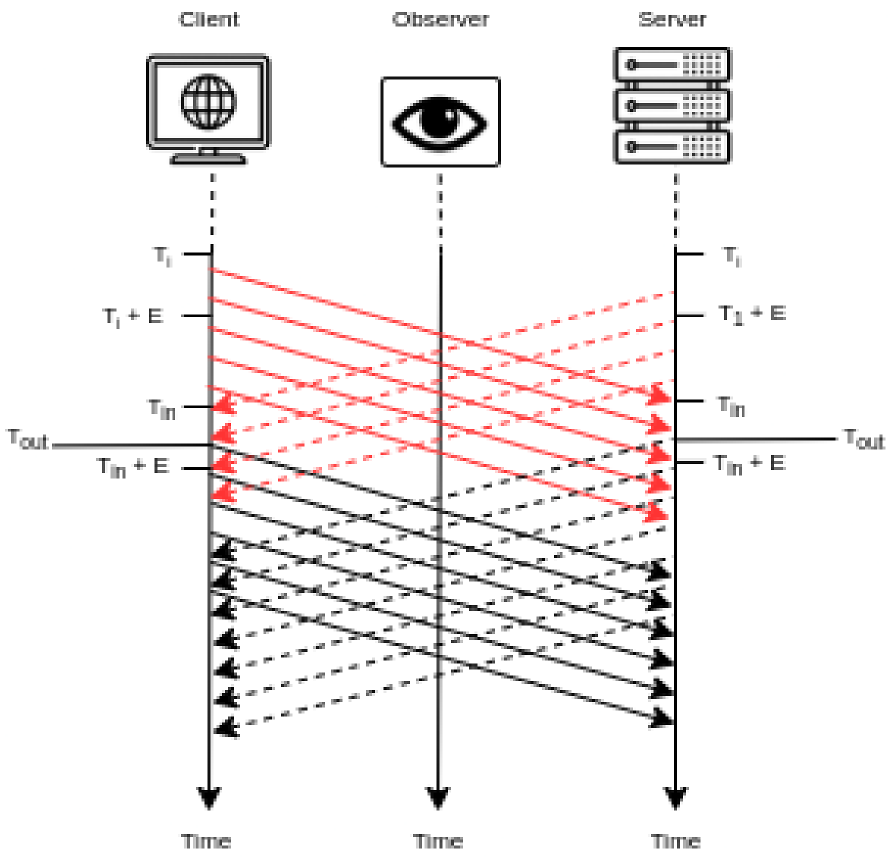


Figure 3.5: OWD nominal scenario. Both endpoints enters in Generation Phase and then in Termination Phase.

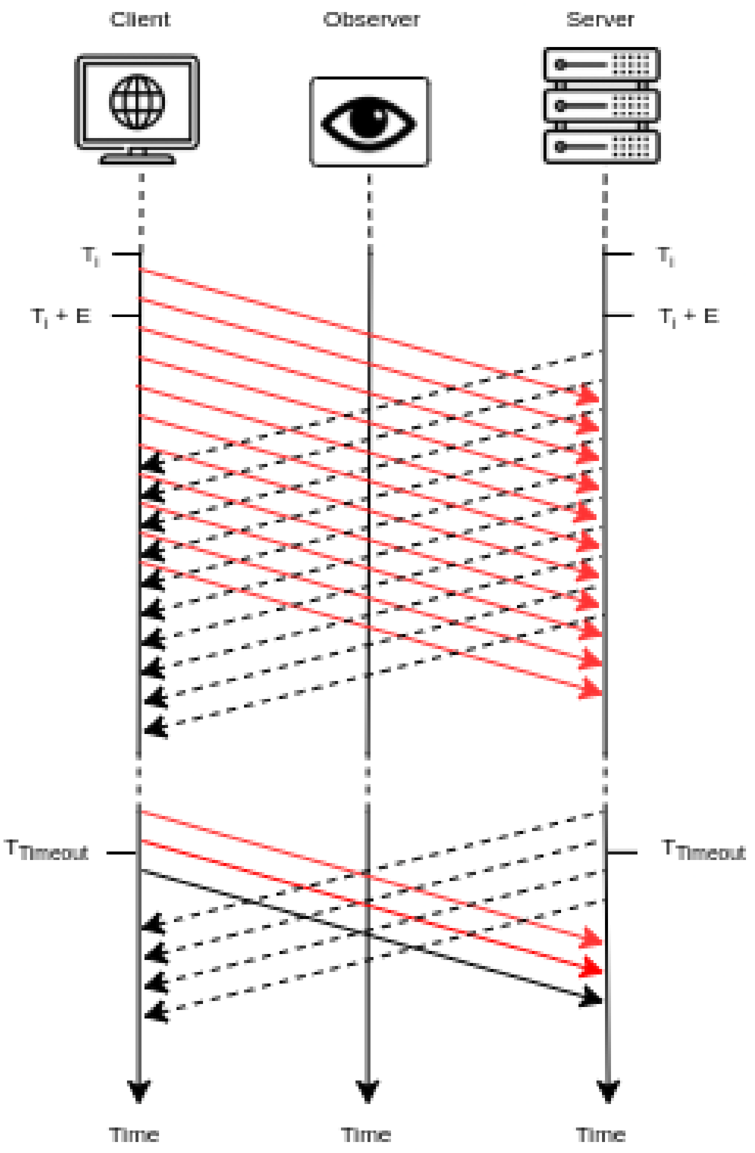


Figure 3.6: The server violates the elapse interval constraint. It doesn't enter in generation phase until the next period. Client stops marking packets only after T_{Timeout} instant.

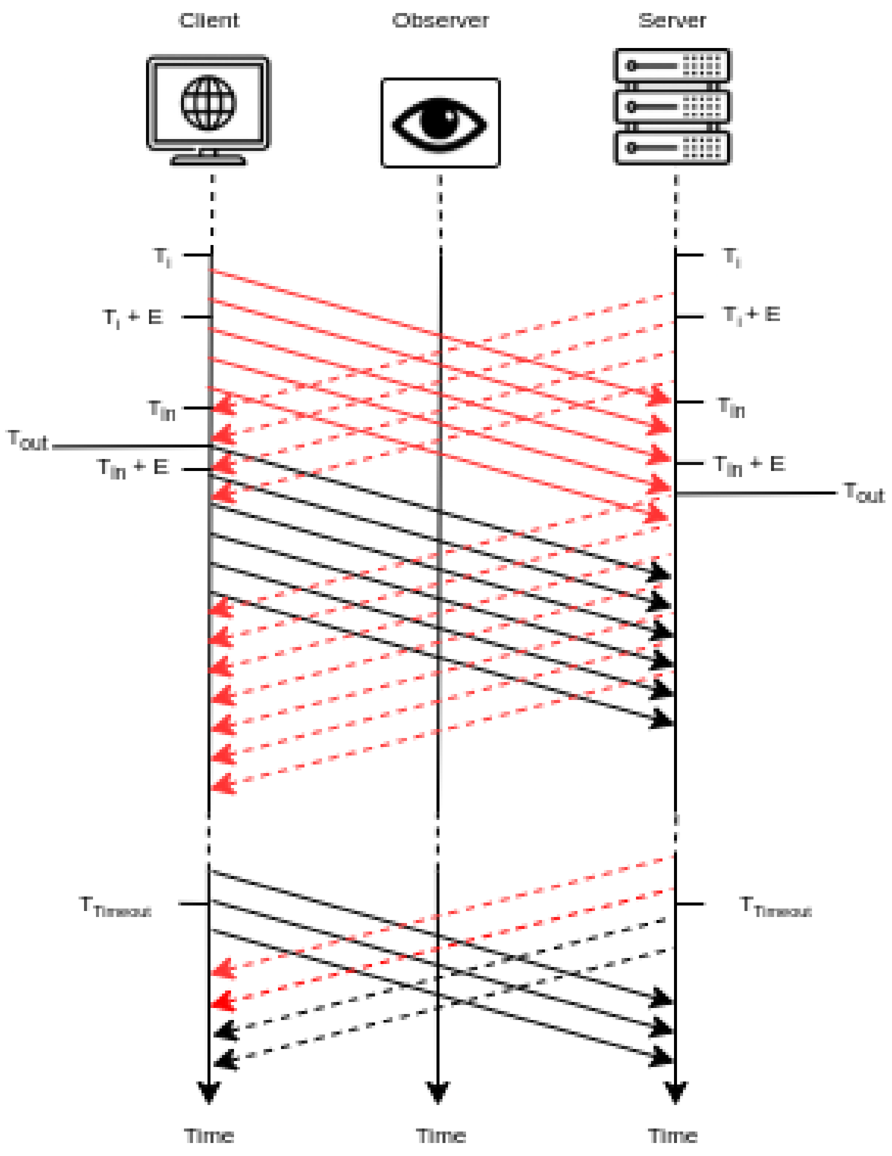


Figure 3.7: The server fail to enter in termination phase. It continue to mark packets until the $\mathrm{T}_{\mathrm{Timeout}}$ instant.

3.7 One-Way Delay Implementation

The following pseudo-code is a possible implementation of the OWD algorithm considering the elapse interval optimization (elapse) and a T0 value set a priori.

```
1:  $spinBit \leftarrow 0$
2:  $generationPhase \leftarrow false$
3:  $lastSpinBitReceived \leftarrow 0$
4:  $endingGenerationPhase \leftarrow false$
5:  $waitNextPeriod \leftarrow true$
6:  $periodStart \leftarrow 0$
7:  $halfPeriodTimeout \leftarrow 0$
8:  $highestPktNum \leftarrow 0$
9:  $t0 \leftarrow t0\_val$
10: $elapse \leftarrow elapse\_val$

11: procedure On_Outgoing_Pkt(hdr)
12:     if $hdr.isLongHeader$ then
13:         return    ▷ Marking logic affects only short header packets
14:     end if
15:     $out\_timestamp \leftarrow time.Now()$
16:     if $waitNextPeriod == false$ AND
17:         $out\_timestamp > halfPeriodTimeout$ then

18:         $generationPhase \leftarrow false$
19:         $spinBit \leftarrow 0$
20:         $waitNextPeriod \leftarrow true$
21:         $periodStart \leftarrow \textsc{roundUpper}(t0, out\_timestamp)$
22:     end if

23:     if $waitNextPeriod == true$ then
24:         if $periodStart == 0$ then
25:             $periodStart \leftarrow \textsc{roundNear}(t0, out\_timestamp)$
26:         end if
27:         if $out\_timestamp \geq periodStart$ then
```

```
28:             if $out\_timestamp \leq (periodStart + elapse)$ then
29:                 $waitNextPeriod \leftarrow false$
30:                 $generationPhase \leftarrow true$
31:                 $spinBit \leftarrow 1$
32:                 $halfPeriodTimeout \leftarrow (periodStart + t0/2)$
33:             else
34:                 $periodStart \leftarrow \textsc{roundUpper}(t0, out\_timestamp)$
35:             end if
36:         end if
37:     end if
38:     if $endingGenerationPhase == true$ then
39:         if $(out\_timestamp - terminationDetected) \leq elapse$ then
40:             $generationPhase \leftarrow false$
41:             $spinBit \leftarrow 0$
42:             $waitNextPeriod \leftarrow true$
43:             $periodStart \leftarrow \textsc{roundUpper}(t0, out\_timestamp)$
44:         end if
45:         $endingGenerationPhase \leftarrow false$
46:     end if
47:     $hdr.SpinBit \leftarrow spinBit$
48: end procedure
```

The procedure above is called every time one of the endpoints is sending a packet to the opposite endpoint. First of all, the IF BLOCK on line 12 allows to returns immediately from the procedure if the current outgoing packets has long header.

It’s possible to divide the procedure in 3 parts:

- **From line 23 to 37:** this part manages the start of a new generation Phase. The "waitNextPeriod" need to be true, meaning that the endpoints is not marking yet. On line 24 there is a nested IF block that allows to compute the nearest measurement period’s starting instance (called once as initialization of the variable). With this pre-computed reference value, it is possible to verify if the current packet is going to be sent before or after the expected period's starting instance. In the first case, noting happen, otherwise: if the packet is going to be sent in the interval between the expected starting period's instance and the elapse interval, then the Generation Phase can start. If the elapse interval constraint is not satisfied instead, a new reference value is computed as the next expected period's starting point (the generation phase is aborted, the endpoint will wait for the next period).

- **From line 16 to 22:** this block is executed in case of the generation phase is lasting more than T1 = 50%T0. The generation phase is aborted and the next expected period's starting instance is computed.

- **From line 38 to 46:** this last part decides if the termination phase can occurs or not. If the packet is sent within the expected time interval in according to the Elapse Interval, then the Generation Phase is interrupted and endpoints will wait for the new starting period's instance. Otherwise, the generation phase will continue and will be interrupted thanks to part 2 of the algorithm. Notice that this part is called only once per period.

```
1: procedure Handle_Incoming_Packet(hdr)

2:     if hdr.isLongHeader then
3:         return
4:     end if
5:     if $hdr.pktNum > highestPktNum$ then
6:         $highestPktNum \leftarrow hdr.pktNum$
7:         if $generationPhase == true$ then
8:             if $hdr.SpinBit == 1$ AND
9:                $lastSpinBitReceived == 0$ then
10:                $terminationDetected \leftarrow hdr.rcvTime$
11:                $endingGenerationPhase \leftarrow true$
12:            end if
13:        end if
14:        $lastSpinBitReceived \leftarrow hdr.SpinBit$
15:    end if
16: end procedure
```

This procedure is called by both client and server when a packet is received and, as the previous procedure, it affects only short header packets. The packet number is checked as a protection against packet reordering. If the endpoint is in generation phase, then the current incoming packet's timestamp is saved and it will be used to decide if the generation phase would terminate or not in according to the Elapse Interval optimization.

Chapter 4

Evaluation

4.1 Software and utilities

The QUICGO. A GO implementation of the protocol

The QUICGO [1] is a Go [2] written, open source implementation of the latest official IETF RFCs QUIC version. By extending this project, it's possible to insert the OWD algorithm logic enabling then the marking mechanism as required. All the lines of code added in the QUICGO, are concentrated mainly in the procedures to manage the incoming and outgoing packets but modifications in the encryption layer over the packet header could be necessary in order to free one or both the reserved bit. For the OWD algorithm, the only bit used for marking is the spin bit, already not encrypted. Clearly, all this code modifications shouldn't interfere with the protocol's core functionalities.

[1]https://github.com/lucas-clemente/quic-go/

[2]https://golang.org/

The On-path observer software

The observer used to perform tests is an extensions of the software implemented by Fabio Bulgarella in his master's degree thesis[13]. This software is completely developed in C++ language and for all its core features exploits the PcapPlusPlus library [3], a cross-platform framework born with the aim to wrap the most popular packet processing engines[4] providing easy-to-use C++ APIs and simplified data structures for capture, process and analyze all the packets sniffed from a network link.

The observer is then thought to work in two different modes:

- **Live Traffic Packets Analysis:** Traffic packets are captured asynchronously during the QUIC connection, stored and analyzed sequentially in real-time without wrote them on a dedicated file.
- **Traffic Packets Capture and then Analysis:** Packets are captured via a sniffing tool like Tcpdump [5], stored in a .pcap file on the local disk and then analyzed sequentially at the end of the QUIC connection, in second time.

In both modes, observer can sniff in one or both up and down directions[6]. Every time a packet is captured directly from a link or is retrieved from the .pcap file in which it has been stored previously, all the useful information in the header are stored in a C++ structure and then exploited by the OWD algorithm's logic.

[3]https://pcapplusplus.github.io

[4]The laboratory's testing environment is hosted over a Unix environment that provide the libpcap packet processing engines

[5]https://www.tcpdump.org/

[6]for the OWD measurement it needs to sniff traffic data only over one of the two links

More in details, from every packets analyzed are retrieved the following field's values:

- **The MSB header byte** used to decide whether the current packet has a long header, and then discard it, or a short header and then extract other information from it. Notice that long header packets would be kept for additional computation only if the 3-packets handshake setup phase is present as discussed in Chapter 5.
- **Incoming timestamp** i.e. the time instant when the packet is sniffed from a link and stored in a .pcap file or directly managed asynchronously in real-time.
- **Spin bit** by which is possible to distinguish between marked and not-marked packets.

4.2 Testing Environment

In a first phase of the evaluation, the correctness of the endpoint's marking mechanism implementation and the observer measurement's capabilities have been tested on a local machine without impairment or by means of Mininet [7], a powerful network emulator tool able to emulate a whole virtual network topology on a single machine with or without the addition of one or more impairment factors.

Then, the algorithm's evaluation has been performed using the hardware installed inside the Telecom Italia's testing laboratory in Turin, Italy, with the possibility to test its behavior by applying different pseudo-real network conditions analyzing then the measurement's samples retrieved by the observer.

Here, all the QUIC packets exchanged between client and server pass through a Network Impairment Emulator (NIE), a network device able to apply to all or part of them the configured impairment's factors.

[7]http://mininet.org

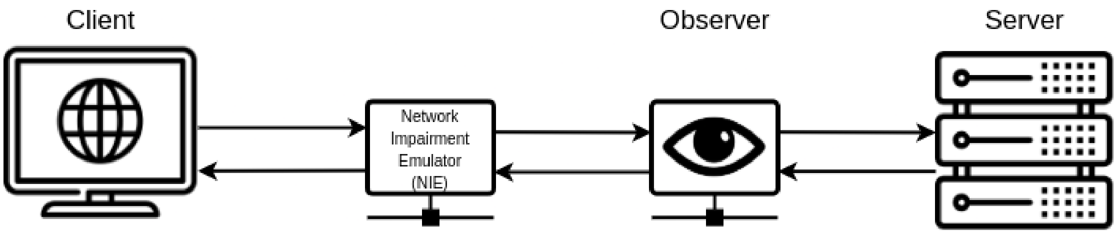


Figure 4.1: The simplified schematic of the network topology in the laboratory. The observer result to be placed AFTER the installed NIE. Packets from the server are detected BEFORE any impairment factors could be applied on them.

4.3 Methodology

Firstly, no variations have been done to the QUIC protocol's core implemented in the QUICGO. This means that the congestion control mechanism is left as-is without additional customization as also the resizing management of the transmission window.

On both upload and download links a minimum 10 ms packet delay has been setup, value considered truthful for a real network link while the links bandwidth is limited to 1 Gbps. The observer has been placed on the upstream link enabling the measurement of the upload half-OWD component and the download OWD values.

To generate network traffic, the QUICGO client sends download and upload HTTP requests for a certain amount of data to the HTTP server wrapped inside the QUICGO server. For all the performed tests, download requests of 400 MB have been sent to the server. The T0 value has been set to 1 second a priori, length of time already identified as long enough to avoid overlapping of consecutive periods. The Elapse time has been set to 1 ms for both generation and termination phase. Accordingly to what reported in section 3.6.1, the half-OWD and OWD samples errors is expected to be:

- Half-OWD $\text{Error}_{\text{max}} = +1\text{ms} \pm$ client-server sync error.
- OWD $\text{Error}_{\text{max}} = +2\text{ms} \pm$ client-server sync error.

Three impairment factors have been taken in account: **packets delay**, **packets reorder rate** and **packet loss rate**. The value of these factors has been varied progressively one at a time following a Fibonacci's Series-like progression and for every emulated network condition, the algorithm has been tested multiple times. All the valid measurement's samples[8] retrieved by the observer have been stored and used to compute the MAX, MIN and AVG value of both half-OWD and OWD.

Another analysis has been performed over the progression of the periods number in a QUIC connection and on how the number of valid OWD and half-OWD samples retrieved have been changed increasing the impairment factors one by one. For this last analysis, a single QUIC connection has been taking in account and represented graphically via charts.

4.4 Result

Below, the results of the tests divided by applied impairment factor:

4.4.1 Packet delay variation

The download link delay is increased from 10 ms up to 31 ms.

Periods number and valid / discarded samples progression

For all the delay values configured, the total number of periods remained from 11 to 14 with at most one period skipped, i.e. no OWD or half-OWD sample has been retrieved during all the period. This would happens due to a traffic hole that blocks the generation phase on the client (no incoming square wave detected) and, as a consequence, no OWD is stored. In this situation, the server could either not enter in

[8]sample's value must be not around the half period's value. A 6% margin has been considered.

generation phase as the client or not enter in termination phase after the half period timeout goes off.

In general, the algorithm can retrieve a valid OWD and half-OWD sample from every periods in which the QUIC connection is divided. Only with higher value of delay, near the 50% T0 value will be discarded. But due to the T0 value set to 1 second, in a real network situation packets delayed of a length of time around 500 ms would be considered as a failure somewhere along the network link taken in account.

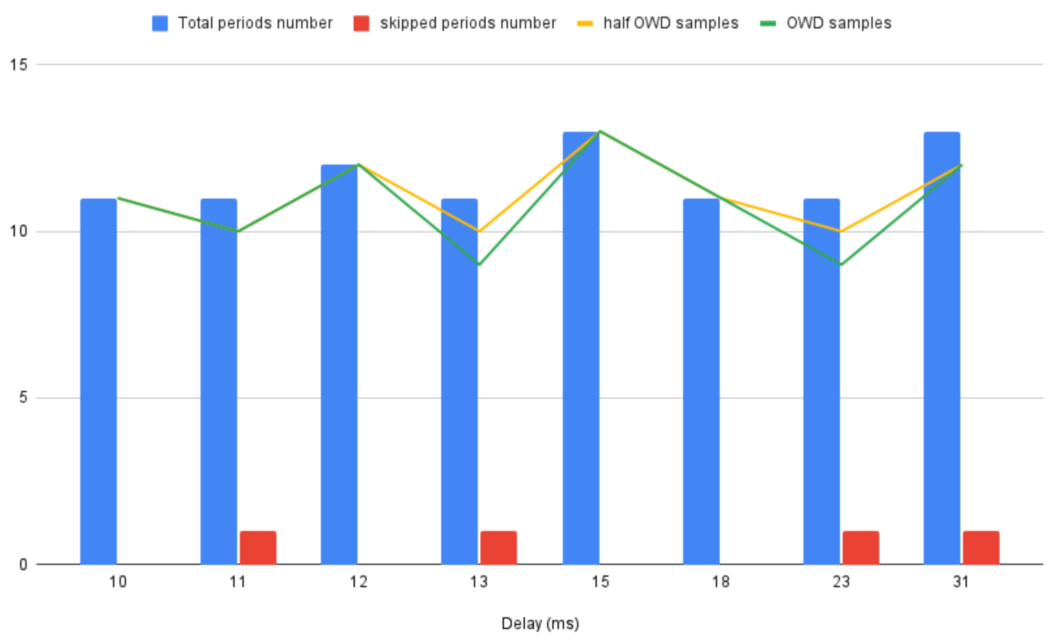


Figure 4.2: Periods number and sampling progression in according to the average downlink delay applied.

OWD and half-OWD evaluation

From all the tests performed, the observer has been able to provide an half-OWD and OWD AVG values near the ideal values expected. The OWD AVG trend is from 0.7% to 2% higher than ideal. Instead, the half-OWD trend is 3% to 5% higher the half-OWD ideal trend. It is then observable that half-OWD samples are a little bit less accurate than the OWD samples. In general half-OWD samples are a bit more susceptible to variations in synchronization between the three involved actors.

Reasonable values also for MIN e MAX values: for every delay scenarios, the MAX value remain inside the interval between the $\mathrm{OWD}_{\mathrm{min}}$ set and the $\mathrm{Error}_{\mathrm{max}}$. More interesting to notice the MIN value, that in some cases is lower that the expected $\mathrm{OWD}_{\mathrm{min}}$. This fact put evidence on the synchronization error component, unpredictable and causing a contraction / dilatation of the measured square wave's length.

Delay(ms)	**10**	**11**	**12**	**13**	**15**	**18**	**23**	**31**
max (ms)	10,664	10,925	10,929	10,711	10,903	10,467	10,641	10,605
min (ms)	10,264	10,227	10,241	10,244	10,282	10,244	10,250	10,176
avg (ms)	10,376	10,483	10,452	10,405	10,427	10,319	10,340	10,326
ideal (ms)	10	10	10	10	10	10	10	10

Table 4.1: Uplink half-OWD values in according to the downlink delay applied.

Delay(ms)	**10**	**11**	**12**	**13**	**15**	**18**	**23**	**31**
max (ms)	10,437	11,507	12,399	13,457	15,983	18,495	23,646	31,458
min (ms)	9,952	10,739	11,857	12,972	14,527	18,003	23,099	31,039
avg (ms)	10,143	11,155	12,109	13,224	15,289	18,234	23,265	31,22
ideal (ms)	10	11	12	13	15	18	23	31

Table 4.2: Downlink OWD values in according to the downlink delay applied.

4.4.2 Packet reordering rate variation

In order to test the behavior of the algorithm with packet reordering, the NIE has been configured to apply a **uniform packets distribution reordering** i.e. the NIE try to uniformly distribuite the reordered packet over a configured number of packet. For the executed tests, N packets have been uniformly distributed over every 1000 packets received and then forwarded after forwarding up to 4 following packets.

Uniform Reorder rate is varied from 0.1% to 0.8% . Higher rates have been tested but not reported in the following tables, mainly because with higher reorder rates the number of measurements samples will be to small to compute meaningful AVG, MIN, MAX values (always considering T0 = 1 second).

Periods number and valid / discarded samples progression

Increasing the packet reorder rate, the periods number during the same QUIC connection also increased with instead a decreased percentage of valid half-OWD and OWD samples retrieved. For OWD samples, the retrieved percentage decrease drastically from 73%, with a reorder rate equal to 0.1%, to 32% in the last case with a slightly fluctuation in the middle. The percentage of stored half-OWD samples has changed from 80%, in the first case considered, to a percentage between 50% and 57% in the last cases.

The packet reorder event triggers the congestion control mechanism with a consecutive reduction of the transmission window's size causing then a longer QUIC connection and clearly an higher number of periods. Less packets are sent in every periods in comparison with a not reordered scenarios. This would cause the violation of the Elapse Interval's constraints in a higher number of periods. It is then less probable that a packet is sent and/or reflected inside the acceptable length of time.

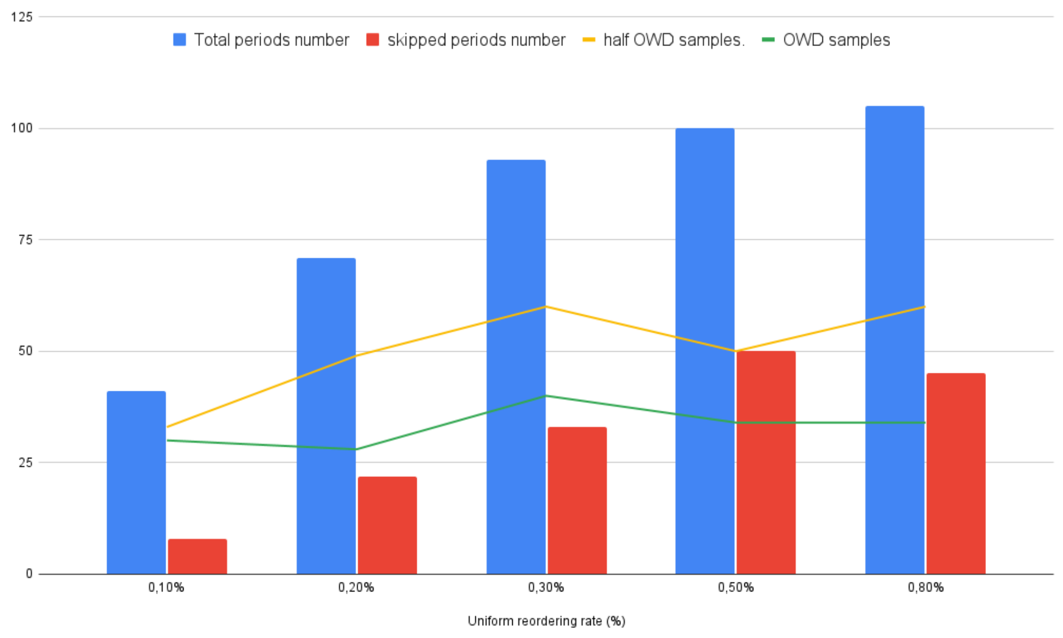


Figure 4.3: Periods number and sampling progression in according to the downlink packet reordering rate applied.

OWD and half-OWD evaluation

From the tables below is possible to notice that the AVG values are not far from the ideal expected values. The OWD AVG trend is from 2.5% to 3.8% higher than the ideal expected trend while the half-OWD trend is about 4.8% to 7% higher than the expected one. As for the delay scenarios, half-OWD AVG is a bit more inaccurate than OWD AVG. Again, this could be caused firstly by the synchronization error component, variable along all the tests performed in an unpredictable way.

Speaking about the OWD, especially the MIN values retrieved, it is noticeable that all of them tend to be lower than the OWD_{min}. The reason could be found again in the sync error component but there is also the spurious edges threat in presence of border reordered packet, causing an underestimation of the samples. The Waiting Interval W has been set to 9 ms during the tests. With an higher value, the MIN values could be a bit more higher and near the ideal values expected.

Unif. Reorder rate(%)	**0,10%**	**0,20%**	**0,30%**	**0,50%**	**0,80%**
max (ms)	11,114	11,139	11,235	11,428	11,331
min (ms)	10,092	10,144	10,196	10,202	10,085
avg (ms)	10,475	10,578	10,639	10,700	10,582
ideal (ms)	10	10	10	10	10

Table 4.3: Uplink half-OWD values in according to the downlink packet reorder rate applied.

Unif. Reorder rate(%)	**0,10%**	**0,20%**	**0,30%**	**0,50%**	**0,80%**
max (ms)	11,367	11,243	11,244	11,349	11,343
min (ms)	9,475	9,452	9,275	9,265	9,378
avg (ms)	10,338	10,248	10,289	10,262	10,376
ideal (ms)	10	10	10	10	10

Table 4.4: Downlink OWD values in according to the downlink packet reorder rate applied.

4.4.3 Packet loss rate variation

As for packet delay and packet reorder, it is interesting to test the algorithm's behavior with different packet loss rates. The NIE has been configured to apply a uniform packet loss distribution, i.e. it drops N packets uniformly distributed over every 1000 packets received.

The rate has been varied from 0.1% to 2%. Some tests have been performed also with higher packet loss rates, but just in a few of these the QUIC connection is effectively started and terminated with the possibility to store OWD and half-OWD samples. In most of the cases, the QUICGO reacted to the increased number of lost packets by cutting off the starting or already established QUIC connection [9]

Periods number and valid / discarded samples progression

The number of periods is higher than an impaired network conditions with just a fixed delay of 10 ms per links. Valid OWD samples are retrieved only from a portion of all the QUIC connection's periods, from a minimum of 24% to a maximum of 35% of all the periods depending on the packet loss rate applied for that sequence of tests. Better for the half-OWD samples where in the worst case, the'47% of all the period provided a valid samples while for the best case up to 60% of all the periods provided a valid sample. Moreover, is possible to notice a steady trend in both total periods number, OWD and half-OWD samples by increasing the packet loss rate (in contrast with what happened increasing the packet reorder rate).

QUICGO protocol's implementation is an experimental trial to realize a client and a server able to communicate by means of this protocol still in developing. It could be just less optimized to manage packet reorder scenarios than packet loss scenarios at the moment. In any case, packet reordering in a QUIC connection is a common situation, more than in a TCP connection and a bit more complex to manage. In fact, because the almost total encryption of every packets, no reordering action can

[9]QUICGO could decide to interrupt the connection returning a **no recent network activity** error.

be computed by any of the possible intermediate nodes between the endpoints.

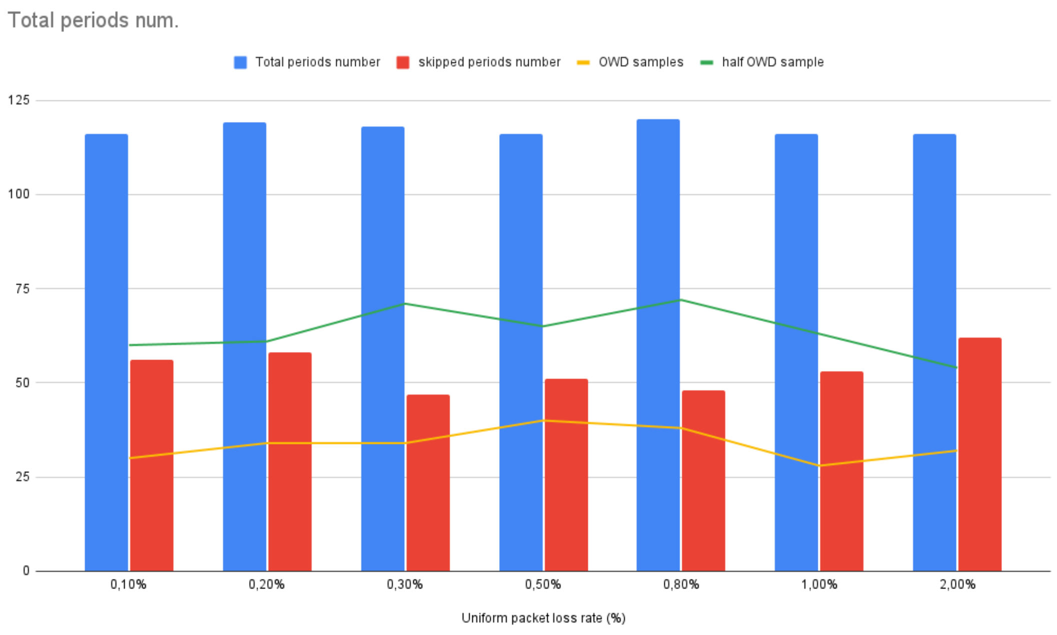


Figure 4.4: Periods number and sampling progression in according to the downlink packet loss rate applied.

OWD and half-OWD evaluation

The algorithm provides reasonable AVG value in every tested packet loss rates for both half-OWD and OWD metrics.

In the first case, the AVG trend is from 6% to 7% higher than the ideal expected AVG values while in the second case, the AVG trend is from 2.6% to 3.9% higher than ideal values. As for the packet reorder and delay, half-OWD values are a bit more inaccurate in relation to the ideal values expected.

The synchronization error component also with packet loss rates is not negletable and more evident in half-OWD sampling.

Speaking about the MIN values, for the OWD measurements are reported underestimated values in all the cases. This could happen by computing one or more OWD samples using a square wave shorter than the expected one due to a square wave's border packet lost.

Unif. Loss rate(%)	**0,10%**	**0,20%**	**0,30%**	**0,50%**	**0,80%**	**1%**	**2%**
max (ms)	11,269	11,252	11,204	11,193	11,321	11,264	11,257
min (ms)	10,263	10,261	10,241	10,174	10,210	10,159	10,185
avg (ms)	10,700	10,674	10,645	10,656	10,686	10,600	10,674
ideal (ms)	10	10	10	10	10	10	10

Table 4.5: Uplink half-OWD values in according to the downlink packet loss rate applied.

Unif. Loss rate(%)	**0,10%**	**0,20%**	**0,30%**	**0,50%**	**0,80%**	**1%**	**2%**
max (ms)	11,024	11,204	11,024	11,265	11,269	11,353	11,251
min (ms)	9,724	9,403	9,556	9,403	9,529	9,393	9,464
avg (ms)	10,348	10,312	10,265	10,384	10,336	10,337	10,386
ideal (ms)	10	10	10	10	10	10	10

Table 4.6: Downlink OWD values in according to the downlink packet loss rate applied.

Chapter 5

The Setup Phase

Period's length T0 is a central aspect for the OWD algorithm and must to be known from both client, server and observer BEFORE the first measurement period. It's very easy to set the T0's value "A priori". The overall OWD algorithm implementation complexity remains the same without any additional line of code. The main drawback is the impossibility to set a tailored value in according to the current network conditions and the probability to store an overestimated T0 value.

For this reason, an initial setup phase would be a valid idea in order to try to retrieve an higher number OWD and half-OWD samples during the same QUIC connection.

5.1 The "classic" spin bit method

This is quite the same algorithm of the latency spin bit algorithm described by Brian Trammell[7]: At the beginning of the new QUIC connection, both endpoints set its own internal spin bit value to 0 and then:

1. After the Handshake is confirmed on client-side [1], client toggles its spin bit value to 1 starting a square wave in direction of the server. It saves the timestamp of the first outgoing marked packet.

[1]this happens when client receive the HANDSHAKE-DONE frame

2. Server detects the square wave edge from the upload link and toggles it's spin bit value from 0 to 1 starting reflecting back the square wave on the download link. It saves the timestamp of the first incoming marked packet.

3. Client receives the first reflected packets. It toggle back to 0 its internal spin bit value interrupting the square wave on the uplink in direction to the server. It saves the incoming timestamp of the first reflected marked packet received.

4. Server receives the first not-marked packet, i.e. the second square wave's edge meaning the end of the square wave itself. It then toggles back the internal spin bit value to 0 interrupting the square wave on the download link. It saves the incoming timestamp of the first not-market packet.

When an endpoints has both the timestamps, it can compute the RTT value as the difference between the two stored timestamp values and then compute the T0 value accordingly to the constraints (see Chapter 3). Now, they will wait for the first T_i instant to try to enter in the first generation phase.

Important to notice that client needs to wait a length of time equal to ½ RTT after computing the T0 value and then before waiting the first Ti instance. In fact, the server will compute RTT value approximately ½ RTT after the client and this detail could provide a rough synchronization between the two endpoint avoiding the client to start the first generation phase a period before the server.

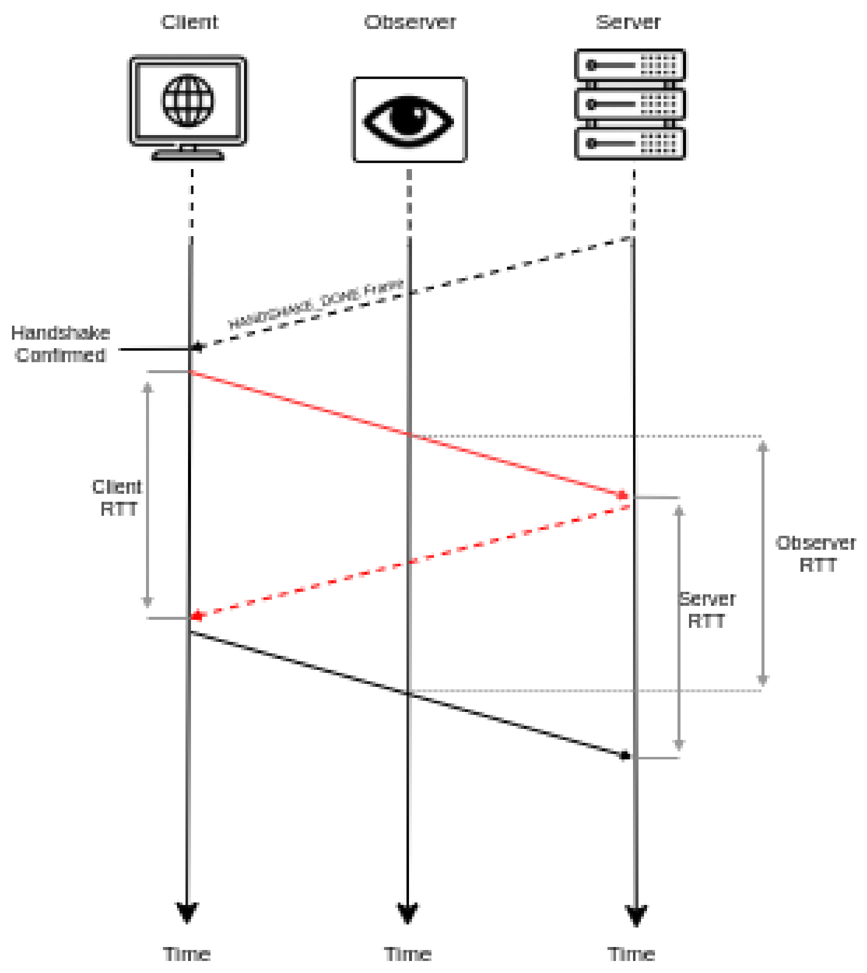


Figure 5.1: Example of the Latency Spin Bit setup phase. The red arrows represents the market short header packets sent firstly by the client after the confirm handshake event. The black arrows represent instead the not-marked packets

5.1.1 Implementation

```
1: $generationPhase \leftarrow false$
2: $setupPhase \leftarrow false$
3: $endingGenerationPhase \leftarrow false$
4: $waitNextPeriod \leftarrow true$
5: $periodStart \leftarrow 0$
6: $halfPeriodTimeout \leftarrow 0$
7: $highestPktNum \leftarrow 0$
8: $spinBit \leftarrow 0$
9: $rtt \leftarrow 0$
10: $lastSpinBitReceived \leftarrow 0$
11: $t0 \leftarrow t0\_val$
12: $elapse \leftarrow elapse\_val$

13: procedure $On\_Handshake\_confirmed$(hdr)
14:     if $hdr.isLongHeader$ then
15:         return
16:     end if
17:     if $isClient$ then
18:         $setupPhase \leftarrow true$
19:     end if
20: end procedure


21: procedure $Handle\_Outgoing\_Packet$(hdr)

22:     if $hdr.isLongHeader$ then
23:         return      ▷ Marking logic affect only short header packets
24:     end if

25:     $out\_timestamp \leftarrow time.Now()$
26:     if $periodStart == 0$ then
27:         if $setupPhase == true$ then
28:             if $isClient$ AND $rtt == 0$ then
```

```
29:                 $spinBit \leftarrow true$
30:                 $rtt \leftarrow out\_timestamp$
31:             end if
32:         end if
33:     else
34:         ......
35:         ......
36:     end if
37:     $header.SpinBit \leftarrow spinBit$

38: end procedure
```

Comparing this procedure with the same procedure without the setup phase, is possible to notice that there is a IF-THEN-ELSE block to decide if a periodStart value is already stored[2] or not and so if the client need to start the setup phase or not. Instead, the ELSE statement from line 31 wraps all the basic OWD algorithm's logic as seen in previous pseudo-code (for both client and server).

```
 1: procedure $Handle\_Incoming\_Packet$(hdr)

 2:     if $hdr.isLongHeader$ then
 3:         return
 4:     end if
 5:     if $hdr.pktNum > highestPktNum$ then
 6:         if $periodStart == 0$ then
 7:             $Handle\_Setup\_Phase$(hdr)
 8:         end if
 9:         ......
10:         ......
11:         $lastSpinBitReceived \leftarrow hdr.SpinBit$
12:     end if
13: end procedure
```

[2]start time instant of the nearest measurement's period

```
procedure $Handle\_Setup\_Phase$(hdr)
    $out\_timestamp \leftarrow time.Now()$
    if $isClient$ then
        if $setupPhase == true$ AND $hdr.spinBit == 1$ then
            $spinBit \leftarrow 0$
            $setupPhase \leftarrow false$
            $rtt \leftarrow (out\_timestamp - rtt)$
            $t0 \leftarrow \textsc{computeT0}(rtt)$
            $t1 \leftarrow (out\_timestamp + rtt/2)$
            $periodStart \leftarrow \textsc{roundUpper}(t0, t1)$
            $waitNextPeriod \leftarrow true$
        end if
    else
        if $setupPhase == false$ AND $hdr.spinBit == 1$ then
            $setupPhase \leftarrow true$
            $spinBit \leftarrow 1$
            $rtt \leftarrow out\_timestamp$
        else if $setupPhase == true$ AND $hdr.spinBit == 0$ then
            $spinBit \leftarrow 0$
            $setupPhase \leftarrow false$
            $rtt \leftarrow (out\_timestamp - rtt)$
            $t0 \leftarrow \textsc{computeT0}(rtt)$
            $periodStart \leftarrow \textsc{roundUpper}(t0, out\_timestamp)$
            $waitNextPeriod \leftarrow true$
        end if
    end if
end procedure
```

The procedure handling the incoming packet calls another procedure in order to manage the initial setup phase. There is a different behavior in according to the endpoint's side. In case of client side, the procedure decide if the setup phase is completed or not and compute the first periodStart timestamp. In case of server side, both start and end of the setup phase is managed here.

Also for the observer the RTT value follow the latency spin bit algorithm: firstly, it is placed on the uplink direction sniffing traffic incoming from the client. When the first marked packet is detected, the packet's incoming timestamp is stored. Then, when the first not marked packet is received after the last marked one detected, the second timestamp is stored and the RTT value is computed as the difference between this two values. The T0 value is derived as for client and server.

5.2 Consideration

Being that this mechanism derive from the classical latency spin bit, both client and server need to take care about the packets sequence number. For the same reason, the observer need to use the waiting interval tricks trying to avoid spurious edges as already discussed in chapter 3. Another aspect to take in account is that QUIC protocol allow to coalesced an application packets (1-RTT) to a long header packet during the handshake phase [3]. What could happen is that the endpoint has not yet the required cryptographic keys[4] for different reason (maybe a delayed packet during the handshake). In this situations, the 1-RTT packet cannot be processed and can be stored temporary in a buffer or even discarded.

This factor could lead to a more variable RTT values computed on the two endpoint. For this reason, is better to wait to receive the HANDSHAKE_DONE frame and then the handshake confirm event before allow the client to send the first marked packet.

[3]in this case, the short header packet is the last in the UDP datagram due to the lack of length field in the header's format

[4]available after the handshake confirmed event

5.3 3-Handshake packets method

This second method uses packets sent in the normal QUIC connection's handshake, based on the TLS 1.3 handshake[4]. The basic idea is the following:

1. Client sends a first long header packet to start a QUIC connection with the server and saves the outgoing packet's timestamp.
2. Server receives the first packet and, in nominal case, sends back a second long header packet as a response saving the sending timestamp.
3. Client receives the second packet, saves the receiving timestamp and computes RTT with the two timestamp provided by the 1° and 2° packet. The endpoint continues the handshake by sending a third packet to the server.
4. Server receives the third packet and saves the timestamp. Now also the server can compute its own RTT value via the difference in time between timestamps retrieved from 2° and 3° packet.

The Observer need to be placed on the uplink in order to detect the first packet sent by the client signalling a new tentative of connection between the two endpoint). The RTT value is computed taking in account the detection timestamp of the first packet and of the third packet (the second packet sent by the client after receiving the first packet from the server).

But what packets to consider for the timestamps?

Firstly, QUIC protocol allows to send coalesced packets in a UDP datagram with the aim to reduce the number of in-fight packets and accelerate the handshake procedure. Then, different packets can be exchange in according to the QUIC implementations and the information already stored on the endpoints[5].

[5]For example, is possible to anticipate some 1-RTT packets via the 0RTT packet, but to do that, information from a previously established connection need to be stored on the endpoints

Speaking about the QUICGO, during the tests emerged that after the first Initial packet, server always sends a **retry packet** (type 0x3) in response to the client with the goal to provide a unique identification token for security purposes and forcing the client to restart the connection trial by sending a new Initial packet[6].

The following simple implementation of the setup phase is thought to exploit this common exchange packets sequence for the RTT and the T0 values.

5.3.1 Implementation

```
1: procedure Handle_Outgoing_Packet(hdr, rcvTime)

2:     if hdr.isLongHeader then
3:         if isClient then
4:             if hdr.LongType == InitialType then
5:                 if firstResponseRecv == false then
6:                     rtt ← rcvTime
7:                 end if
8:             end if
9:         else
10:            if hdr.longType == initialType OR hdr.longType == retryType then
11:                if firstResponseRecv == false then
12:                    rtt ← rcvTime
13:                end if
14:            end if
15:        end if
16:    end if
17: end procedure
18: procedure Handle_Incoming_Packet(hdr, rcvTime)
19:     if hdr.isLongHeader then
```

[6]If the client receives the NONCE token, it will attach a copy of it in every successive outgoing packets.

```
        **if** $isClient$ **then**
            **if** $hdr.longType == initialType$ OR $hdr.longType == retryType$ **then**
                **if** $firstResponseRecv == false$ **then**
                    $firstResponseRecv \leftarrow true$
                    $rtt \leftarrow rcvTime - rtt$
                    $t0 \leftarrow \textsc{computeT0}(rtt)$
                **end if**
            **end if**
        **else**
            **if** $hdr.longType == initialType$ **then**
                **if** $firstResponseRecv == false$ **then**
                    **if** $rtt == 0$ **then**
                        **return**
                    **end if**
                    $firstResponseRecv \leftarrow true$
                    $rtt \leftarrow rcvTime - rtt$
                    $t0 \leftarrow \textsc{computeT0}(rtt)$
                **end if**
            **end if**
        **end if**
    **end if**
**end procedure**
```

Notice that the "firstResponseRecv" flag is required in order to avoid multiple computations of the RTT and then T0 value every time a long header packet is sent and/or received.

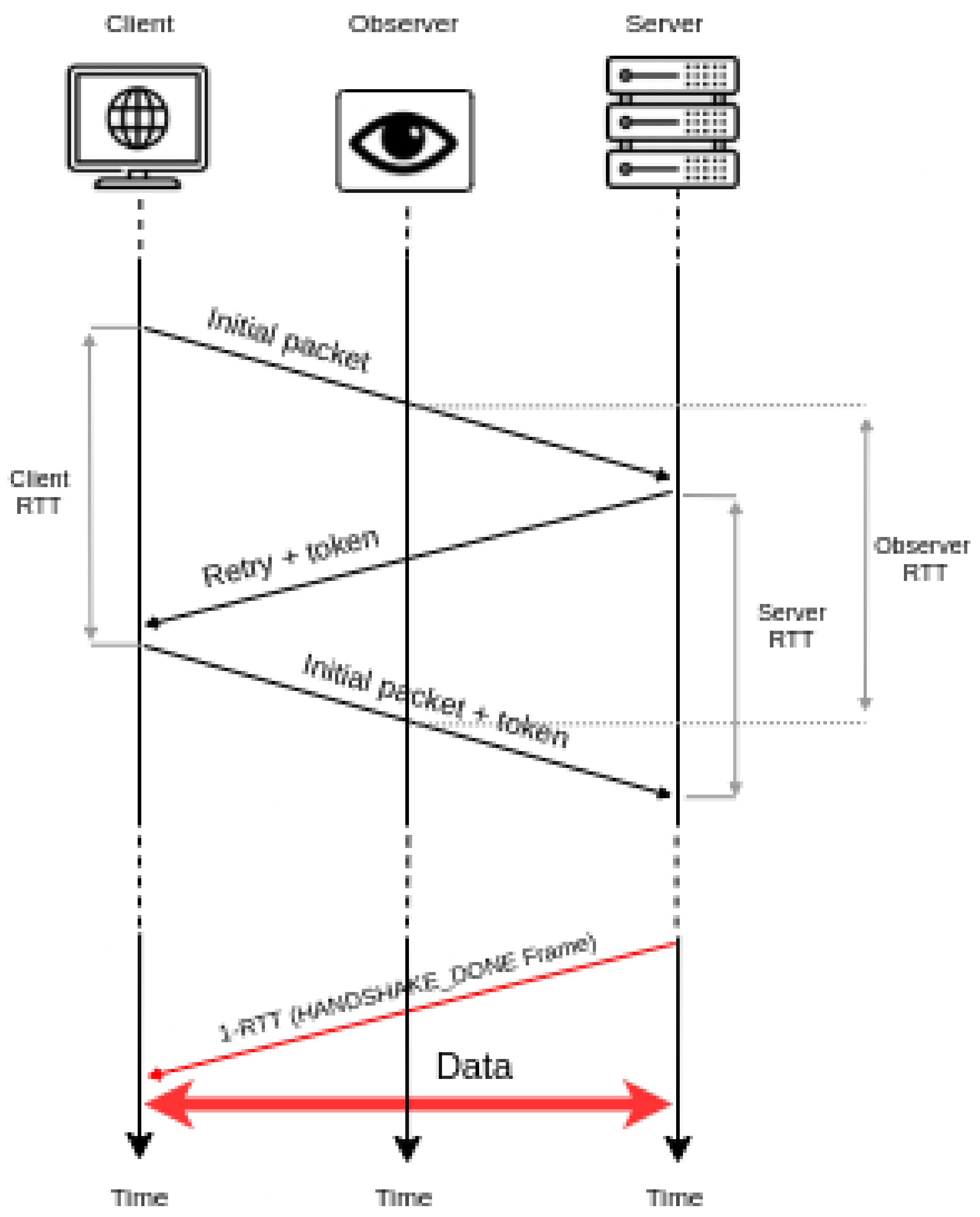


Figure 5.2: QUIC handshake's packets proposed for the setup phase as tracked with QUICGO

5.3.2 Consideration

This method doesn't require to add information to any packet's header, like the first setup phase reported and avoid to "waste" a couple of initial measurement periods for this scope. Comparing the pseudo-code of the two setup phases, this second is simplest but is more difficult to track the meaningful same three packets also due to the lack of any sort of marking strategy. After the first initial packet different coalesced packet are send, fact more evident in case of packet reorder and packet loss during this phase.

By using the three proposed packet, additional unpredictable delay could be present due to the different processing delay required by the two packet's type but also due to the processing delay required to manage the NONCE token.

Another packet that the server can send after receiving the first Initial packet is the Version packet[7], a special long header packet sent to the client with the aim to signal what QUIC versions it should consider and interrupting, de facto, the connection trial. The client will try then to send another Initial packets as a way to restart the handshake procedure. This last packet could be used instead of the retry packet [8].

The Observer, in the basic setup phase's idea, is only sniffing on the uplink direction. This means that it will compute RTT value considering the distance in time between the first incoming Initial Packet and the second Initial Packet from the client. The problem is the impossibility for it to know if the second Initial packet is in response to a server packet (Initial, version or a retry packer) or it represents the first long header's packet for a new handshake trial. This could cause an overestimation of the RTT value on the observer. The two endpoint will compute the RTT value in a second connection tentative while observer will already store a T0 value waiting for the first square wave signal. A possible optimization could be place the

[7]Not considered in the pseudo-code

[8]Option not considered in the pseudo-code above

observer on both the directions. Until now observer has been place on the upload link OR on the download link, but this because is one of the requirements of the OWD methodology. Usually, an ISP is able to monitor both the directions of the managed network's portion and then is also able to configure an on-observer to sniff packets in two-way mode. In this way, if the observer sniff a long header packet on the downlink after the detection of a first long header packet on the uplink, then the second Initial packet should be considered as the response of the server to the client meaning that the handshake is ongoing between the two endpoints. It's anyway possible that the detected packet on the downlink could be lost before be received by client and AFTER be detected by the observer causing again an erroneous RTT value computation on stored on the observer.

Chapter 6

Conclusions

This work wants to evaluate the theoretical validity of a new algorithm proposed by Telecom Italia with the goal to provide the QUIC protocol of a new One-Way Delay measurement system exploitable by a passive on-path observer placed on the upload link or on the download link between two endpoints. The mechanism has been implemented by extending the QUICGO project, i.e. an experimental running implementation of the QUIC protocol, and then tested in Telecom Italia's laboratory emulating different network conditions.

The One-Way Delay would like to be a replacement of the Latency Spin Bit proposed by Brian Trammell able to provide accurate measurements only in optimal network scenarios. For this reason, the new algorithm exploits the spin bit considering some additional mechanism in the trial to overcome (most of) Trammell's algorithm limitations.

The main drawback with the One-Way Delay algorithm is the need of a strong time synchronization between client, server and the observer, aspect that could introduce a not negletable error component and an increasing of inaccuracy in OWD and half-OWD samples as reported mainly performing tests in presence of packet reorder rates and packet loss rates.

In general, what emerged analyzing the results obtained, is that the OWD algorithm is able to provide valid OWD and half-OWD samples with more than acceptable accuracy in every tested conditions.

The Elapse interval optimization on the endpoints provide high protection against traffic holes occurring during the generation phase and the termination phase of the marking logic allowing to define acceptable error ranges in both OWD and half-OWD samples but also to discard the overestimated samples.

Interesting protection also in presence of packet reordering rates, thanks to the introduction of the waiting interval, with the possibility to filter an high percentage of spurious edges during the sample computation. Unfortunately, if the reordered packet are on the square wave's borders, the OWD sample will be underestimated in any case, fact noticeable also in case of marked packet lost in proximity of the borders. In this case the OWD algorithm cannot provide more protection than the Trammell's algorithm. A possible solution could be the use of a second bit, maybe using one of the reserved bit in the short header packet, with the goal to add another mark over the already marked packets on the square wave's borders and then uniquely define the expected signal's length. Unfortunately, this algorithm is thought to use only one bit and so this optimization cannot be considered.

Speaking about the retrieved samples performing the tests, the number of valid measurements decrease mainly with the increasing of packet reorder and packet loss rates. This phenomenon can be attributed principally to the activation of the congestion control and the reduction of the transmission window's size causing then an important reduction in the number of packets sent during a measurement period and a more probability to violate the constraints imposed by the Elapse interval to avoid overestimated samples. The solution could be reduce the length of every measurement periods T0 in which the QUIC connection is divided, maybe via an initial setup phase, paying attention to avoid the overlapping of contiguous marking phases.

Bibliography

[1] Martin Thomson. *Version-Independent Properties of QUIC*. RFC 8999. May 2021. DOI: `10.17487/RFC8999`. URL: `https://rfc-editor.org/rfc/rfc8999.txt` (cit. on p. 2).

[2] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. May 2021. DOI: `10.17487/RFC9000`. URL: `https://rfc-editor.org/rfc/rfc9000.txt` (cit. on p. 2).

[3] Martin Thomson and Sean Turner. *Using TLS to Secure QUIC*. RFC 9001. May 2021. DOI: `10.17487/RFC9001`. URL: `https://rfc-editor.org/rfc/rfc9001.txt` (cit. on pp. 2, 8).

[4] Jana Iyengar and Ian Swett. *QUIC Loss Detection and Congestion Control*. RFC 9002. May 2021. DOI: `10.17487/RFC9002`. URL: `https://rfc-editor.org/rfc/rfc9002.txt` (cit. on pp. 2, 58).

[5] Mike Belshe, Roberto Peon, and Martin Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. May 2015. DOI: `10.17487/RFC7540`. URL: `https://rfc-editor.org/rfc/rfc7540.txt` (cit. on p. 2).

[6] Mike Bishop. *Hypertext Transfer Protocol Version 3 (HTTP/3)*. Internet-Draft draft-ietf-quic-http-34. Work in Progress. Internet Engineering Task Force, Feb. 2021. 75 pp. URL: `https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-34` (cit. on p. 2).

[7] Brian Trammell and Mirja Kühlewind. *The QUIC Latency Spin Bit*. Internet-Draft draft-ietf-quic-spin-exp-01. Work in Progress. Internet Engineering Task Force, Oct. 2018. 8 pp. URL: `https://datatracker.ietf.org/doc/html/draft-ietf-quic-spin-exp-01` (cit. on pp. 2, 14, 51).

[8] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: `10.17487/RFC8446`. URL: `https://rfc-editor.org/rfc/rfc8446.txt` (cit. on p. 7).

[9] Kim Ervasti. *A Survey on Network Measurement: Concepts, Techniques, and Tools*. 2017. URL: `https://www.cs.helsinki.fi/u/kervasti/projects/A%20Survey%20on%20Network%20Measurement%20-%20Concepts,%20Techniques,%20and%20Tools%20-%20Kim%20Ervasti%20-%2031-12-2016.pdf` (cit. on pp. 12, 13).

[10] Giuseppe Fioccola, Alessandro Capello, Mauro Cociglio, Luca Castaldelli, Mach Chen, Lianshu Zheng, Greg Mirsky, and Tal Mizrahi. *Alternate-Marking Method for Passive and Hybrid Performance Monitoring*. RFC 8321. Jan. 2018. DOI: `10.17487/RFC8321`. URL: `https://rfc-editor.org/rfc/rfc8321.txt` (cit. on p. 14).

[11] Mauro Cociglio, Alexandre Ferrieux, Giuseppe Fioccola, Igor Lubashev, Fabio Bulgarella, Isabelle Hamchaoui, Massimo Nilo, Riccardo Sisto, and Dmitri Tikhonov. *Explicit Flow Measurements Techniques*. Internet-Draft draft-mdt-ippm-explicit-flow-measurements-02. Work in Progress. Internet Engineering Task Force, July 2021. 40 pp. URL: `https://datatracker.ietf.org/doc/html/draft-mdt-ippm-explicit-flow-measurements-02` (cit. on p. 18).

[12] Fabio Bulgarella, Mauro Cociglio, Giuseppe Fioccola, Guido Marchetto, and Riccardo Sisto. «Performance Measurements of QUIC Communications». In: *Proceedings of the Applied Networking Research Workshop*. ANRW '19. Montreal, Quebec, Canada: Association for Computing Machinery, 2019, pp. 8–14. ISBN: 9781450368483. DOI:

10.1145/3340301.3341127. URL: https://doi.org/10.1145/3340301.3341127 (cit. on p. 18).

[13] Fabio Bulgarella. «QUIC Performance Measurement Algorithms to evaluate connection delay and loss rate». MA thesis. Politecnico di Torino, 2019, pp. 47–48. URL: http://webthesis.biblio.polito.it/id/eprint/10904 (cit. on p. 38).