POLITECNICO DI TORINO

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING

Master of Science in Computer Engineering

Master Degree Thesis

QUIC Performance Measurement

Algorithms to evaluate connection delay and loss rate

Supervisors:
prof. Riccardo Sisto
prof. Guido Marchetto

Candidate:
Fabio Bulgarella

Company Tutor
Telecom Italia Lab
dott. Mauro Cociglio

Academic Year 2018 – 2019

Summary

This thesis aims to provide the QUIC protocol with a complete mechanism to passively evaluate its performance. In particular, two algorithms proposed by Telecom Italia are analysed and evaluated to test their operation. The first one, the delay bit, is a single bit signal which enables reliable passive measurements of network delay. Its purpose is to improve the spin bit performance in the presence of network impairments, trying to behave better than a similar solution (i.e., the Valid Edge Counter) that uses two bits to validate spin bit measurements. In a nutshell, when a connection starts, the client sets the delay bit of the first packet to 1. Then, this single packet — called "delay sample" — is reflected indefinitely by the endpoints. An intermediate observer can measure the difference in time between two consecutive delay samples determining the RTT of that connection. The second algorithm enables the end-to-end round trip loss rate measurement using, also in this case, a single bit signal called loss bit. The client generates a train of marked packets having the loss bit set to 1. Then, these are reflected by the server back to the client that, in turn, reflects them again to the server which finally completes the process reflecting them to the client, realizing a two round trip reflection. A passive on-path observer, placed on whatever direction, can trivially count and compare the number of marked packets seen during the two reflections estimating the loss rate experienced by the connection. The two solutions are tested on an emulated network in different conditions. The delay bit, compared to the Valid Edge Counter, in the absence of significant network impairments, produces more or less the same amount of valid RTT samples. When impairments occur, its behaviour changes: in the presence of high loss rate the amount of accurate RTT measurements worsens because the algorithm is slower to restart the whole measurement process (compared to the counterpart); in the presence of high reordering rates, differently from the VEC that shows a high measurement rejection rate, the delay bit maintains a good amount of valid samples. Regarding the packet loss computation, a preliminary analysis shows that statistical loss rate evaluation can be performed looking at the loss bit signal. In particular, loss rates below 12% are more or less correctly estimated. For higher values, instead, the algorithm tends to underestimate the actual loss rate experienced by the channel.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

IP Networks are not reliable for definition. Their working principles, based on a best-effort strategy, do not guarantee data delivery in a predefined time window or the delivery itself. This means that every time a packet is transmitted there is a certain probability that this will be delayed or lost during its journey through the network. For instance, a packet could be delayed because one or more nodes on path get overloaded and the outgoing queues grow considerably determining waits on output. Alternatively, a packet may be lost due to a node or link failure or may be discarded by a node because of congestion of its ports. Or again, packets may be dropped by a node since they contain bit errors.

When a communication service — in particular, a real-time voice or data service such as call, video conference, etc. — is provided by means of a packet-switched IP network, a performance measurement in terms of packet loss, delay and/or jitter on packet flows carrying the service provides an indication of the quality of service (QoS) perceived by the end users of the communication. Although network providers already rely on different tools to measure the quality of their links, it is increasingly common to evaluate individual connections between two endpoints. In fact, simply evaluating the status of specific links (or portions of the network) does not give information about the overall performance experienced by the end users.

Moreover, measuring the performance of a specific connection enable network operators to determine with certainty whether or not any problems encountered by the user derive from disservices within its domain. Furthermore, networking hardware vendors have always expressed interest in these methodologies so they can integrate monitoring features inside their products to support network administrators.

With the exception of network protocols and tools specifically designed for performance monitoring and network management (such as ICMP, SNMP, IPPerf and others), the

entire TCP/IP protocol stack has not been conceived from the beginning with the intent to provide characteristics or functionalities exploitable to obtain information about delay and loss rate of a connection. In fact, the few features available in this field have been introduced later as optional fields. However, their application proved to be poor due to the fact that being connection-related features, to be effective, they should be adopted on the whole network path including the endpoints; surely a difficult task for non-native functionalities.

A new transport layer network protocol, QUIC [1], is currently in the process of standardization and is expected to replace TCP in the near future, at least for transmissions concerning the HTTP protocol which certainly represents a good portion of the entire traffic exchanged on the Internet. It is therefore in some ways mandatory to take advantage of the possibility of integrating a set of explicit features to natively measure performance in terms of delay and loss rate.

A first step in this direction has already been made by the introduction of the spin bit [2] within the official QUIC protocol specifications, enabling passive measurement of network delay. However, the algorithm behind its functioning does not guarantee proper measurements in every network conditions. To face this issue, an additional two-bit validation signal (i.e., the Valid Edge Counter [3]) has been proposed by the ETH of Zurich with the purpose to give an instrument to correctly detect those measurements that due to network impairments must be discarded because incorrect.

In a similar way, Telecom Italia is proposing the introduction of an additional single bit signal (i.e., the delay bit) able to improve the spin bit performance and — in theory — the VEC variant as well. In addition, in order to provide the QUIC protocol with a complete system for performance monitoring, TiLAB theorized a completely new algorithm to enable passive measurement of network end-to-end loss rate.

Throughout this paper, a description and analysis of these algorithms will be provided, including a comparison about network delay measurement between the spin bit, its VEC variant and the solution proposed by TiLAB.

1.2 Thesis structure

Chapter 2

This chapter aims to give an overview of the knowledge needed to understand the thesis work exposed in the following chapters. First of all, it deals with some motivations about the need for a new transport layer protocol at the expense of TCP. Then, it provides a description of the QUIC protocol highlighting its salient points and advantages. Finally, the spin bit algorithm is explained, focusing on its limitations and trying to understand how the Valid Edge Counter is supposed to overcome them.

Chapter 3

The third chapter provides a detailed description of the delay bit algorithm (i.e., the one related network delay measurement) as well as the logic used by the observer to extract the measurements from the spin and delay signals. Then, an overview of all possible measurement types achievable by this algorithm is given. In addition, it proposes a possible implementation in pseudocode — the one used during the testing phase — focusing on the impact it can have on endpoints that implement it from a computational point of view.

Chapter 4

This chapter analyses the loss rate algorithm giving a brief explanation of its general working principles. Then, it focuses on two possible implementation describing advantages and weaknesses of both. Finally, it gives information about the observer logic needed to mine connection loss rate from the observed traffic.

Chapter 5

The fifth chapter describes the design and implementation of the software observer used to evaluate both algorithms.

Chapter 6

This chapter outlines the testing environment used to perform a practical analysis of the functioning of individual protocols. It therefore provides the results obtained analysing data derived from the software observer, setting at the same time a comparison between the two solutions used to improve the spin bit performance: delay bit and Valid Edge Counter.

Chapter 7

Final chapter highlights the strength and weakness points, deriving final considerations on the results achieved. Finally, it outlines future work ideas to further improve the performance measurement capabilities of QUIC protocol.

Chapter 2

Background

2.1 Internet: big growth, small evolution

Even though in the last decades the Internet has changed and grown so quite outstandingly, the same can not be said for its core protocols. Since 1993, looking at the TCP/IP protocol suite, no significant changes have been seen in the internet architecture. This does not mean that nothing has been done to adapt the protocols to today's requirements but, for many reasons, it has been preferred to circumvent the known problems by introducing a series of minor changes rather than modifying the protocols at their heart.

The main motivation behind this evolutionary stasis is to be attributed to the ever-increasing economic and political interests which characterize today's network. In fact, in a commercial network, new technologies are introduced when they solve tangible problems — usually not otherwise bypassed using workarounds — or when a remarkable economic return can be made by ISPs. Even this last statement could be refuted if we consider that networks, in order to work in harmony, must be interoperable. This means that almost every significant change should be adopted by all providers to make it fully functional, effectively undermining the possibility of creating added value for the individual provider [4].

Furthermore, it is worth to consider that the development and the maintenance of a wide geographical network by an internet provider are highly burdensome in terms of investments and allocated resources. If we also consider the fact that in recent years the average cost of internet access has decreased significantly — making it practically within everyone's reach — it is easy to understand that the profit margins that can be achieved by a network operator have been reduced considerably and, with it, the chance of rapidly adapting the network to new technological frontiers.

For these reasons, almost all the improvements and new features made for the cardinal

internet protocols have been introduced as optional extensions in support of the already present core functionalities. Consequently, their adoption, except for rare cases, has never been such as to make them extensively deployed in the whole network. Some examples are the Explicit Congestion Notification (ECN), standardized in the distant 2001 but still not much used today; the various attempts in the field of quality of service such as DiffServ, which is also operating mostly in the domain of some ISPs rather than as an end-to-end model. Or again the Multipath TCP and the Multicast IP, not widely deployed; and the list could easily continue.

Transport layer protocols, and in particular the very popular TCP, do not escape from this analysis. Accordingly, a new transport protocol (QUIC) is currently being standardized by the IETF, which aims to overcome all these evolutionary related problems through an interesting working philosophy.

The next section gives a small overview of the main critical issues at the base of such a decision.

2.2 The need for a new transport protocol

TCP is a transport protocol defined as "ossified" by the IETF. Every effort to optimize it or to add new features collides with a number of complications that pretty much always hamper its evolution.

First of all, it is implemented at the Kernel level in the operating systems; any changes in its working principles require therefore lengthy time for the release of the update patches and, in any case, it is a risky operation especially for those sensitive services that rely on it.

Moreover, in the last decades, ISPs have equipped their networks with specific hardware devices designed to optimize and/or to have greater control over the traffic conveyed by their links. This is the case of TCP accelerators, firewalls and NATs — to name just a few — without which today's internet would not work. All these devices are grouped by the term "middlebox". Any substantial modification to the TCP protocol would require, therefore, the update of these middleboxes; that is a particularly complicated and expensive operation especially if they are implemented in hardware.

In general, networks make assumptions about the appearance of TCP packets and their behavior so, in case of TCP changes, the risk is that somewhere the packets are dropped. In addition, TCP brings with it various intrinsic problems that have pushed Google (and now the IETF) to invest time and money in the research of a substantial and definitive solution to the problem.

To list just a few, TCP headers are not encrypted or authenticated; anyone with network access can manipulate them, in good or bad faith. The space inside the header for current and future options is limited to 40 bytes, more than half of which are almost always used; there is, therefore, little scope for introducing new additional features. Coupled with TLS, handshake time is not satisfactory, especially over long distances. Lastly, it suffers from the problem of head-of-line blocking (a single unsatisfied request makes all the others wait)[1].

These are some of the problems to which the internet community is trying to find a solution in the most painless way possible. The next section explains how QUIC is supposed to solve them.

2.3 The QUIC Protocol

QUIC is a new transport layer protocol created by Google and adopted by IETF community with the intent to generalize its features and use it as a replacement for the old and ossified TCP protocol in the near future. The goal behind its conception aims to reduce latency times — speeding up communications — and, at the same time, making the protocol upgradable by exceeding the limits of the current TCP.

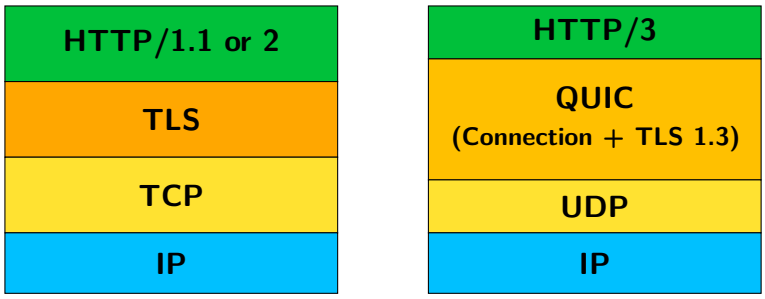
Initially conceived as a transport protocol for the exclusive use of the web and therefore of its HTTP protocol, under the current guidance of the IETF has been revised in a general purpose key to increase its possible applications. Currently, the works for standardization are under way, however the first official version should not take long and will be probably ready by the end of 2019. This thesis is based on revision 17 of the official QUIC draft [1]. At the time of writing, revision 18 is already published and, in all likelihood, several other revisions will be made available in the months ahead in preparation for the final release of the official RFC. Despite this, the key features of the protocol are now firmly defined, leaving room just for small changes and refinements.

2.3.1 Key features

QUIC is designed on the basis of decades of transport and security experience. This means that every decision made on its design is inspired by reliable and efficient mechanisms tested and matured in other technologies.

QUIC exploits UDP protocol as a substrate; this way it can easily maintain compatibility with current middleboxes, operating systems and legacy devices. Since UDP is a connectionless protocol, QUIC handles all of the logic needed to guarantee a reliable

[1] This is due to the fact that TCP open a unique bidirectional stream between endpoints; therefore, one lost packet in this unique stream makes all following traffic wait until that packet is retransmitted and received.



(a) Current configuration (a) QUIC configuration

Figure 2.1. QUIC protocol stack compared to the classic TCP+TLS configuration.

connection between two endpoints. In addition, it operates in the application layer — so in the user space — meaning that any updates on its working principles, even substantial ones, do not require OS changes which would take long time to be deployed. By doing so, whenever a new feature or protocol revision will be introduced, an update of servers and application clients will be enough.

Following the pattern traced by HTTP/2, QUIC is a multiplexed protocol. This implies the fact that a single connection can contain multiple independent streams — unidirectional or bidirectional — each of which can carry an arbitrary amount of data divided into frames. Taking advantage of this feature, it is easy to understand how the problem of the head-of-line blocking is solved at the core of the issue. In fact, in case of packet loss, only the streams whose frames were contained in the lost packet will be blocked. As a result, streams that have not suffered any loss will be able to continue their delivery process to the application.

In a world that is constantly moving towards the encryption of every single byte exchanged on the network, QUIC is not an exception. Indeed, the TLS cryptographic protocol — in its latest revision[2] — is an integral part of the QUIC protocol. Used in AEAD configuration, it provides authentication and confidentiality of the entire transmission, with the exception of a few fields in the header which are simply authenticated to maintain visibility and ensure protocol functionality. Furthermore, by embedding TLS into the protocol, QUIC guarantees significantly lower handshake time compared to the TCP+TLS counterpart; i.e. 0 to 1 RTT and 1 to 3 RTT, respectively.

Confidentiality is certainly a good thing if you look at the user's privacy and the possibility of keeping the protocol long-lived by protecting it from any attempt at tampering by middleboxes. On the other hand, however, the task of the network administrator

[2]The latest revision of TLS, RFC 8446 [5], is currently a proposed standard under examination by the IESG. Its use within QUIC is described in the relevant QUIC-TLS [6] draft.

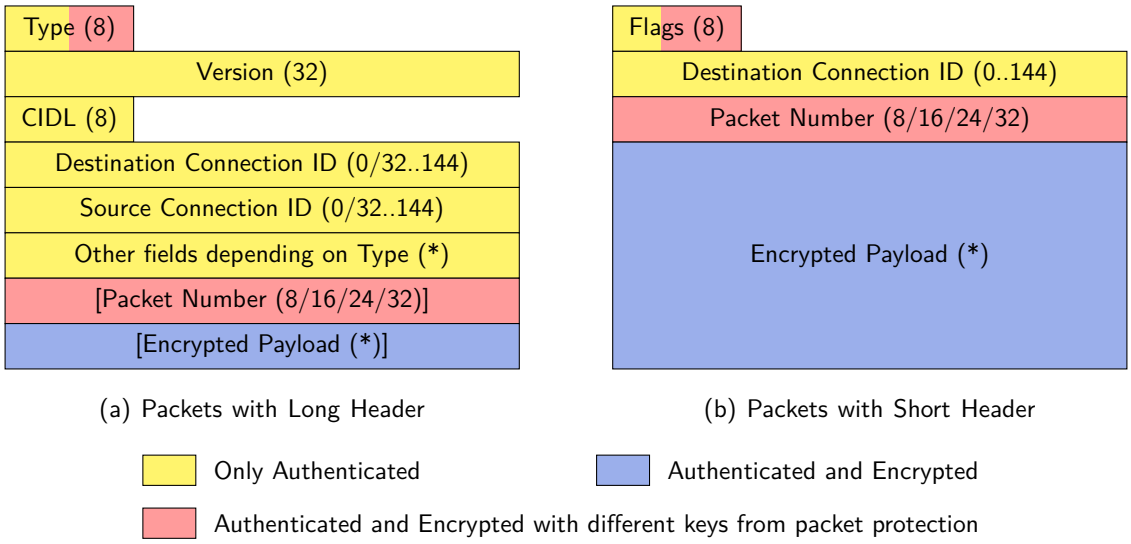


Figure 2.2. The structure of a QUIC packet. Field size in bits.

becomes particularly complicated, since he has to guarantee the correct functioning of the network without being able to observe what is happening on it.

The next section explains what information are available in the QUIC header.

2.3.2 QUIC header

In order to minimize the amount of data "wasted" in each packet for control information, QUIC uses two different formats for its header. Packets with the *long header* are used during connection establishment. Packets with the *short header* are designed for minimal overhead and are used after a connection is established and encryption keys are available (see Figure 2.2).

The first byte of a QUIC packet determines which type of header structure is in use. For the long header its most significant bit is set to 1, making that first octet the **Type** byte; for the short header it is set to 0 and, in this case, the first byte takes the name of **Flags**.

The **Type** byte is used to distinguish between different types of long header QUIC packets used for version negotiation, connection establishment and key agreement.

The **Version** field indicates which version of QUIC is in use and determines how the rest of the protocol fields are interpreted.

The **CIDL** field provides the length of the two following connection ID fields: the four

high bits are used for Destination, while the four low bits for Source. These values are encoded as unsigned integers; non-zero lengths must be increased by 3 to get the full length of the connection ID. This way the possible length values are 0 and [4-18] bytes.

The **Packet Number** enumerates a QUIC packet sequentially. If present[3], its length can be in the range of 1 to 4 bytes according to the two least significant bits of the Type (or Flags) byte.

The **Flags** byte of the short header assumes the following pattern: `01SRRKPP`. The most significant bit, as already said, identifies a short header. The next bit — also called fixed bit — is set to 1 and packets containing a zero value for this bit must be discarded. The third bit (`S`) has been assigned to the SpinBit functionality discussed in the next section. The next two bits (`RR`) are reserved for future uses. The sixth one (`K`) is the Key Phase bit and it is used to help a packet recipient to identify the used packet protection keys. Finally, the last two bits (`PP`) provide the packet number length.

Finally, it should be noted that in this case the **Encrypted Payload** carries and protects not only the application data frames but also control frames used for connection management purposes.

In the next chapters, in an attempt to provide the QUIC protocol with a performance measurement mechanism, two methods for RTT and packet loss computation will be discussed and analyzed. Their implementation involves the use of the two bits reserved for future purposes discussed in this section (i.e. only short header packets will expose performance measurement related information).

[3]The packet number is always present in the short header; in contrast, in the long header its presence changes depending on the type byte.

2.4 Network performance measurement

Performance measurement is one of the key points for the proper functioning of a network. Basically, the three factors influencing the perceived quality of a network are throughput, delay and loss rate. Accordingly, network administrators need to be able to monitor these parameters in order to constantly evaluate their links, understand if they are undersized — thus requiring an upgrade — and act promptly to any problems that is varying the expected behavior of the network.

Different techniques can be used to achieve this result; in general, these can be classified into two macro groups:

passive techniques that refer to the process of measuring a network, without creating or modifying any traffic, but merely observing certain characteristics exhibited by the various protocols; and

active techniques by means of which specific packets are introduced into the network for the purpose of tracing them — as they traverse the links — or simply to exchange traffic between two endpoints which can mine information from this interaction.

Even though both approaches are widely used nowadays, this thesis takes into account only passive techniques. These, in fact, are extensively deployed by network providers to monitor the performance of TCP connections which represent the overwhelming majority of all traffic exchanged on internet. For instance, RTT of a TCP connection can be evaluated tracing handshake messages, or by computing the time difference between TSval (timestamp value) and TSecr (timestamp echo reply) option fields as explained in the official RFC [7]. Or again, a passive observer could count TCP retransmissions and estimates the connection loss rate accordingly. As you can see, all these methods rely on information freely readable from the protocol header.

Since the purpose of the QUIC protocol is to replace — even only partly at first — the use of TCP as a transport layer protocol, a valid alternative to the information exposed by TCP has to be found. Considering that QUIC encrypts essentially the whole transmission including control data, some explicit information, inevitably, must be exposed on the wire in order to allow network operators to measure more or less accurately RTT and loss rate of a connection.

2.5 The latency Spin Bit

Several efforts have already been made to introduce a passive delay measurement mechanism within the QUIC protocol. Brian Trammel in one of his draft [2] describes the addition of a so-called "latency spin bit" — or briefly "spin bit" — in the QUIC header, which allows end-to-end RTT measurements of a QUIC connection.

The latency spin bit is a single bit value — placed in the third most significant bit of the QUIC short header[4] according to the official QUIC draft — that toggles once per RTT, enabling latency monitoring from intermediate observation points. Basically, client and server, maintain an internal per-connection spin value (i.e. 0 or 1) used to set the spin bit on outgoing packets for that connection.

The spin value is set to the appropriate value in accordance with the following rules. Both endpoints initialize the spin bit to 0 when a new connection starts. Then:

> when **the client** receives a short header packet that increases the higher packet number seen so far, it sets the connection spin value to the opposite value contained in the received packet.
>
> when **the server** receives a short header packet that increases the higher packet number seen so far, it sets the connection spin value to the same value contained in the received packet;

The resulting spin value is therefore used for the following outgoing packets until an incoming packet with the opposite spin bit arrives changing the configured value. Note that the packet number control is performed to prevent reordered edges from being reflected by the endpoints (see Section 2.5.1).

This simple mechanism allows the endpoints to generate a square wave such that, by measuring the distance in time between pairs of consecutive edges[5] observed in the same direction, a passive on-path observer can compute the round trip delay of that QUIC flow.

Moreover, if the observer is symmetrically placed on the channel — then it has visibility both on the upstream channel and on the downstream channel — it can determine the components of the RTT by measuring the delay between an edge observed in the upstream direction and the one previously detected in the downstream direction and vice versa.

---

[4] The latency spin bit draft [2] proposes a different placement for the spin bit inside the short header, specifying the sixth most significant bit (0x04) of the first byte. The reason is that a new configuration for the short header's first byte has been introduced in the last QUIC draft [1].

[5] Edge means the point at which the square wave changes from the high to the low state, and vice versa.

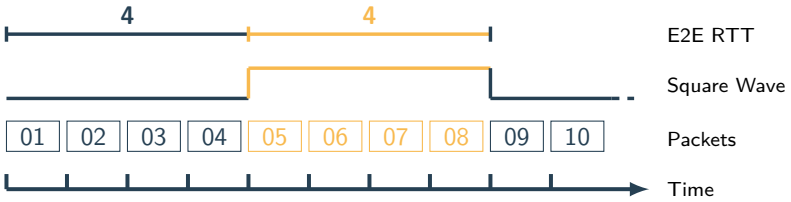


Figure 2.3. The spin bit square wave observed by an intermediate point that measuring the distance in time between two edges can determine the RTT of connection. Packet numbers, even if shown, are not visible on the wire.

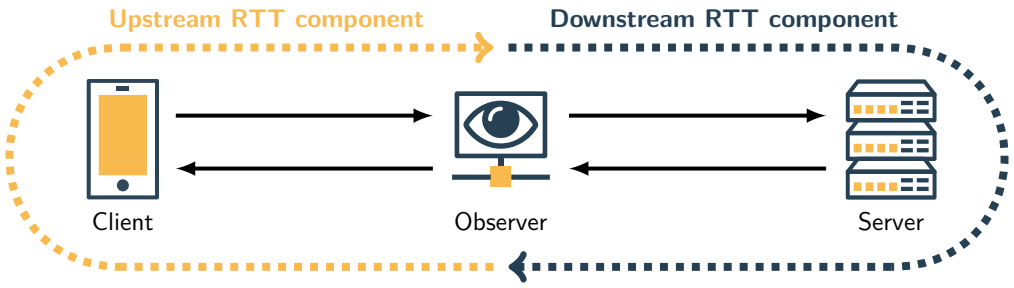


Figure 2.4. Graphical explanation of RTT components. The grey dashed arrow describes the segment of network path traveled by the same edge determining the downstream RTT component (i.e., from its detection in the upstream direction to the one in the downstream direction). Specular considerations for the yellow arrow.

2.5.1 Spin Bit Limitations

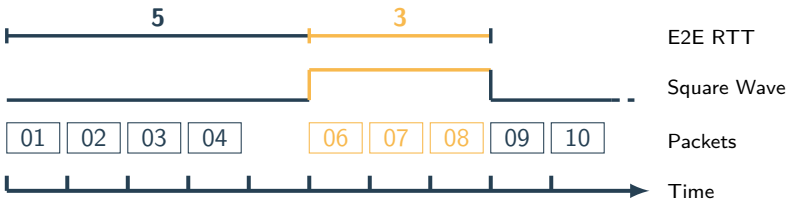
The spin bit mechanism allows passive observers to measure the transmission delay of a QUIC flow easily and quite thoroughly. This statement is true under certain conditions and in the absence of network impairments.

Putting aside for the time being network impairments issues such as reordering and packet loss, three main factors influence the delay determined by the spin bit analysis. They are the application transmission frequency, the flow control algorithm and the latency introduced by the underlying connectionless transport layer used by QUIC (UDP). This last one can be neglected because it is generally small enough not to alter the delay significantly. On the contrary, application limited and flow control limited senders may introduce further delays that bring the overall measured delay to be considerably higher than network RTT. Therefore, the spin bit provides reliable network latency information only when the sender is neither application nor flow control limited. In any other cases, the spin bit, instead of providing the network RTT, exposes information about the application period [2]. Practically, to obtain valid RTT samples, whenever an edge reaches an endpoint, this one must be reflected in the opposite

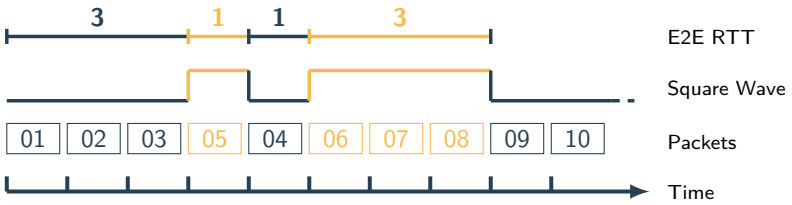
direction as soon as possible without adding significant delay. This implies that the application must always have something to transmit and that no waits are experienced by transmission queue due to slowdowns introduced by the flow control.

Taking into account network impairments instead, the situation becomes more interesting but at the same time not easy to manage by a hypothetical observer. *Packet loss* may tend to cause wrong estimates of RTT due to period width changes. In fact, when a packet carrying a spin edge is lost, its adjacent periods surely will suffer a variation in width which results in overestimating the concluded period and instead underestimating the one that just begun.

Reordering can cause troubles in the spin signal as well. In this case, the problem occurs when one or more packets close to one edge — or the edge itself — are reordered in a way that leads to the generation of at least a new spurious edge. As a consequence, the observer will inevitably detect more periods, instead of the two initials, producing a greater amount of wrong RTT measurements. As previously mentioned, spin bit signal generation is resistant to reordering because the endpoints can trivially observe packet numbers and evaluate from time to time if an incoming edge is caused by a reordered packet, producing in output a filtered signal cleaned of any spurious edges.



(a) The loss of an edge causes the wrong estimates of the two related adjacent period.



(b) A reordered edge leads to the creation of two intermediate small fake periods, shortening the initials too.

Figure 2.5. The spin bit square wave observed in the presence of network impairments.

2.6 The Valid Edge Counter

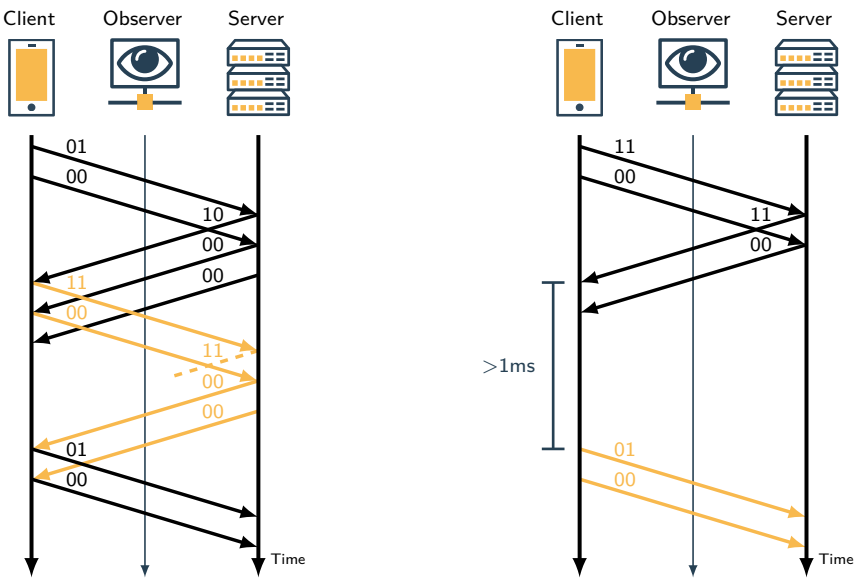
The Valid Edge Counter [3] — or simply VEC — is an additional signal introduced to handle the just seen limitations experienced by the spin bit alone. Using the last two future reserved bits available in the header, it provides the observer with valuable information that can be used to validate with certainty the goodness of the edges.

We have seen that packet numbers can be used by the endpoints to filter out spurious edged caused by reordering. However, since the packet number is transmitted ciphered on the wire, without its support, a passive observer cannot recognize reordered edges and even worse detect lost edges. Moreover, due to delayed edges, the measurements obtained from the spin signal should be preferably post-processed in order to discard in a heuristic way those evidently overestimated. To face these problems, the VEC algorithm adds a two-bit counter to each packet whose purpose is to report whether an edge was valid when transmitted by the endpoint.

The working principles are quite straightforward. Each endpoint initializes the VEC in the same way (i.e., there is no difference between client and server). First of all, a value greater than zero is assigned exclusively to valid edges; therefore, every transmitted packet between two consecutive spin edges carries a VEC value of 0. Then, when an endpoint detects an incoming packet carrying a spin transition, the VEC value of the next generated edge is set to the value contained in the received packet incremented by 1. The VEC holds at 3, the maximum value achievable with a two-bit counter. As a result, an edge received with a VEC of 3 will be reflected with the same value.

Moreover, to address the spin bit limitation where the duration of the period is overestimated in case of application-limited senders, the VEC algorithm introduces a delay threshold (defaulting to 1ms) exceeded which the outgoing edge is transmitted with a VEC of 1 so that a passive observer understands that a problem occurs and then the measurement must be restarted.

Basically, the value of the VEC is increased every time a valid edge is reflected by one of the two endpoints, actually counting the number of semi-paths (i.e., the path between client and server or between server and client) correctly crossed by the edge, so without incurring network impairments. Instead, when the endpoint detects an impairment, the counter is set back to 1 so that the observer avoids completing incorrect measurements, restarting the whole process.



(a) A valid edge is reflected incrementing its VEC value.

(b) If an edge is delayed more than a threshold (1ms) its VEC value is set back to 1.

Figure 2.6. Graphic explanation of how the Valid Edge Counter works. Black and orange arrows for spin bit 0 and 1, respectively.

2.6.1 Use of the VEC by a passive observer

An on-path passive observer, looking at the VEC value of each packet, can evaluate the validity of an edge detected on the spin signal and decides whether to take a measurement or not. In details, when a packet containing a spin bit transition is observed:

- if it contains a VEC value of 0, it has to be considered as a not valid edge unusable to take a measurement; it is probably the result of reordering or loss;
- if it contains a VEC value of 1, it can be used as a left edge to start a new RTT measurement, but can not be used as a right edge to complete the computation of a RTT delay.
- if it contains a VEC value of 2, like before, it can be used as a left edge but not as a right edge unless the observer is symmetrically placed on the channel. In this case, the packet can be also used as a right edge to take a RTT component measurement regarding the portion of the path between the observation point and the opposite direction in which the previous edge carrying a VEC of 1 was observed.
- if it contains a VEC value of 3, it can be used at the same time as a left edge, to start a new RTT measurement, and as a right edge, to complete a previously started one. In addition, the edge can also be used to determine RTT component delay in either direction.

2.7 QuicGo

QuicGo[6] is an implementation of the QUIC protocol in Go language[7]. It roughly implements the IETF QUIC draft and has been used as a starting point to implement the algorithms discussed in the following sections. This means that all tests done to evaluate their performance have been performed using the resulting modified version whose code is available at `https://github.com/fabiobulgarella/quic-go`.

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

[7] `https://golang.org/`

Chapter 3

Delay Measurement

This chapter introduces the delay bit algorithm to the reader. Through it we try to solve the limitations shown by the spin bit while proposing an alternative solution to the Valid Edge Counter which, using two bits, prevents the addition of further features within the QUIC protocol (e.g., a loss signal). The topics covered in this part of the thesis have already been presented in the IETF draft "New Spin bit enabled measurements with one or two more bits"[10].

3.1 Algorithm overview

As we have seen in the previous chapter, the spin bit signal alone is not enough to evaluate correctly in every network condition the RTT of a QUIC flow. In order to solve this problem, Telecom Italia theorized the possibility of introducing an additional delay signal, along the lines of what done by the Valid Edge Counter, but using just one bit instead of two.

The delay bit, as it has been called by Telecom Italia, is a single bit signal that can be used by passive observers to measure the RTT of a network flow, avoiding the spin bit ambiguities that arise as soon as network conditions deteriorate. Unlike the spin bit, which is actually set in every packet transmitted on the network, the delay bit is set only once per round trip. Therefore, the main idea is to have a single packet, with a second marked bit (the delay bit indeed), that bounces between client and server during the entire connection life (figure 3.1). This single packet from now on, for the sake of simplicity, will be called *Delay Sample*.

A simple observer placed in an intermediate point, tracking the delay sample and the time in which it is encountered in every spin bit period, can measure the end-to-end round trip delay of the connection. In the same way as seen with the spin bit and the VEC, it is possible to carry out other types of measurements. Paragraph 3.5 gives an overview of the observer capabilities.

From an implementation point of view, the delay bit is placed in the partially unencrypted (but authenticated) QUIC short header, alongside the spin bit, occupying one of the two bits left reserved for future experiments. As things stand, according to the last QUIC official draft [1], the proposed scheme of the first header byte would be `01SDRKPP`.

In order to describe the delay sample working mechanism in detail, we have to distinguish two different phases which take part in the delay bit lifetime: initialization and reflection. The former leads to the generation of the delay sample, while the latter is in charge to realize the bounce behavior of this single packet between the two endpoints.

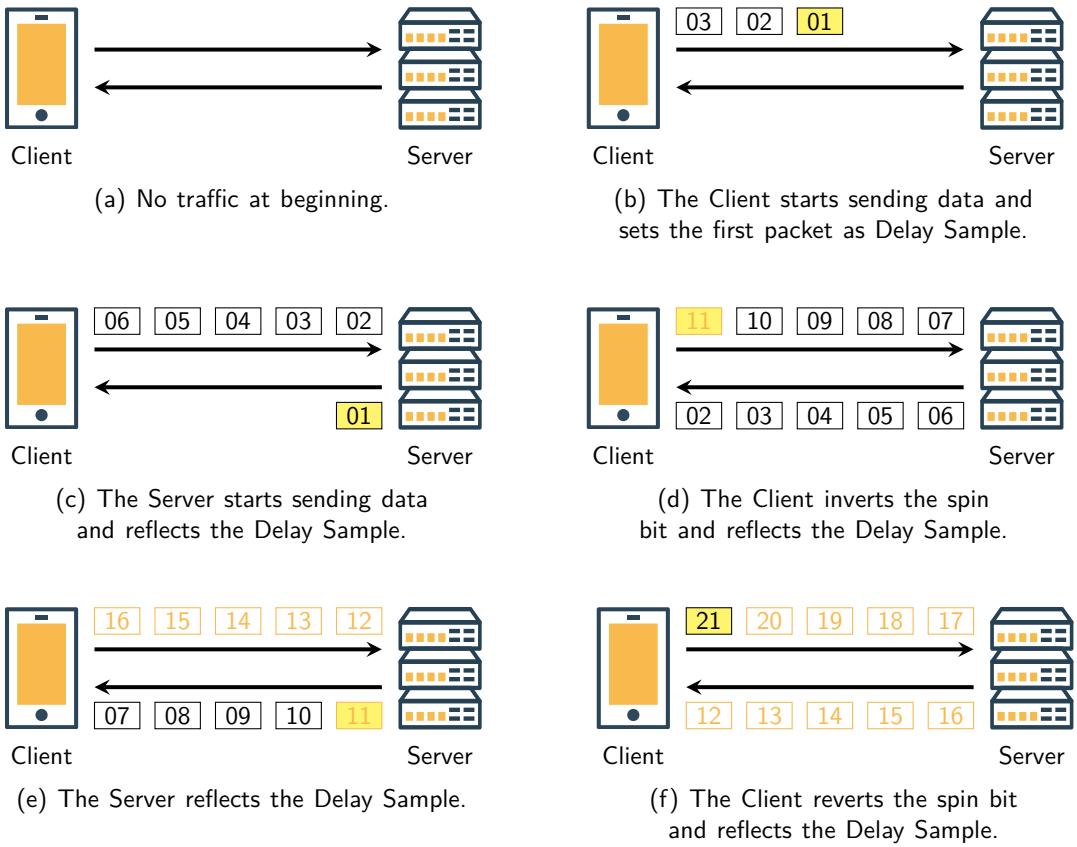


Figure 3.1. Schematic representation of the delay sample mechanism in absence of network impairments. Packets are represented with packet numbers and different colors: black or orange for spin bit equal to 0 or 1, yellow background when delay bit is set.

3.2 Generation phase

During this first phase, endpoints play different roles. First of all, it has to be clear that a single delay sample should be bouncing per round trip period (and so per spin bit period). According to that statement, in order to simplify the general algorithm, it was decided to relegate the delay sample generation just to one of the two endpoints:

the client, when the connection starts and spin bit is set to 0, initializes the delay bit of the first packet to 1, so it becomes the delay sample for that marking period. It follows that only this packet is marked with the delay bit set to 1 for this round trip period; the other ones will carry only the spin bit;

the server never initializes the delay bit to 1; as we will see later, its only task is to reflect the incoming delay bit into the next outgoing packet only if certain conditions occur.

Theoretically, in absence of network impairments, the delay sample should bounce between client and server continuously, for the entire duration of the connection. Actually, that is highly unlikely mainly for two different reasons:

- the packet carrying the delay bit might be lost during its journey on the network which is unreliable for definition;
- one of the two endpoints could stop (or delay) sending data because the application is limiting the amount of traffic transmitted;

To deal with these problems, the algorithm provides a procedure to regenerate the delay sample and to inform a possible observer that a problem has occurred, and then the measurement has to be restarted.

3.2.1 The recovery process

In order to relieve the server from tasks that go beyond the mere reflection of the sample, even in this case, the recovery process belongs to the client.

We said before that a delay sample is strictly related to its spin bit period. Considering that relation, the client is in charge to verify that every spin bit period ends with its delay sample. If that does not happen and a marking period terminates without a delay sample, the client waits a further *empty period*; then, in the following period, it restarts the generation phase setting the delay bit of the first outgoing packet to 1, making it the new delay sample. The empty period, as you would expect, is needed to inform the intermediate points that there was an issue and a new delay measurement session is starting.

3.3 Reflection phase

Reflection is the process by which a delay sample is bounced between client and server. At this stage, the behavior of the two endpoints is slightly different. With the exception of the client that in the previously exposed cases generates a new delay sample, by default the delay bit is set to 0.

Server side reflection: whenever a packet with the delay bit set to 1 arrives, the server marks the first packet in the opposite direction as the delay sample, *if it has the same spin bit value*. While if it has the opposite spin bit value this sample is considered lost.

Client side reflection: whenever a packet with delay bit set to 1 arrives, the client marks the first packet in the opposite direction as the delay sample, *if it has the opposite spin bit value*. While if it has the same spin bit value this sample is considered lost.

In both cases, if the outgoing marked packet is transmitted with a delay greater than a predetermined threshold since the reception of the incoming delay sample (1ms by default), reflection is aborted and this sample is considered lost.

It is noteworthy that differently from what happens with the VEC for which the reflection always concerns the edge of the period, in this case reflection takes place for the packet that is carrying the delay bit regardless of its position within the period. Reason for which it is necessary to introduce that condition of validation — emphasized in the description — in order to identify and discard those samples that, due to reordering, might end within a period not their competence.

Furthermore, as already seen with the VEC, by introducing a threshold for the retransmission delay of the sample, it is possible to eliminate all those measurements which, due to lack of traffic on the endpoints, would be overestimated and not true. Thus, the maximum estimation error, without considering any other delays due to flow control, would amount to 2ms per measurement, in the worst case.

3.4 Delay Sample Implementation

The following pseudo-code shows the implementation of the delay sample algorithm together with the spinbit reflection logic needed for its operation:

```
 1: spinBit ← 0                                          ▷ Connection related variables
 2: delayBit ← 0
 3: gotDelaySample ← False
 4: skipDelaySample ← False
 5: forceDelaySample ← False
 6: lastDelaySample ← 0                                  ▷ In time
 7: highestPktNum ← 0                                    ▷ The highest packet number seen by the peer
 8:
 9: procedure On Incoming Packet(hdr)                    ▷ hdr is the header of the incoming packet
10:     if hdr.isLongHeader then
11:         return                                      ▷ Continue only for short header
12:     end if
13:
14:     if hdr.pktNum > highestPktNum then              ▷ To ignore spurious edges
15:         if client perspective then
16:             if spinBit = hdr.spinBit then           ▷ If is a spin edge
17:                 if not gotDelaySample then          ▷ If the ended period did not get its DS
18:                     if skipDelaySample then         ▷ If this was an empty period
19:                         skipDelaySample ← False     ▷ Restart the recovery process
20:                         delayBit ← 1
21:                         forceDelaySample ← True
22:                     else                            ▷ If not, leave an empty period
23:                         skipDelaySample ← True
24:                     end if
25:                 else
26:                     gotDelaySample ← False
27:                 end if
28:             end if
29:             spinBit ← not hdr.spinBit               ▷ Client inverts
30:         else
31:             spinBit ← hdr.spinBit                   ▷ Server reflects
32:         end if
33:     end if
34:
35:     if hdr.delayBit = 1 then                        ▷ If this packet is a delay sample
36:         if client perspective then
37:             if hdr.spinBit ≠ spinBit then           ▷ If next packet has opposite spin bit
38:                 gotDelaySample ← True               ▷ This period did get its DS
39:                 delayBit ← 1                        ▷ Reflect DS
40:                 lastDelaySample ← time.Now()
41:             end if
42:         else if hdr.spinBit = spinBit then          ▷ If next packet has the same spin bit
43:             delayBit ← 1                            ▷ Reflect DS
44:             lastDelaySample ← time.Now()
45:         end if
46:     end if
47: end procedure
```

```
48:
49: procedure On Outgoing Packet(hdr)                ▷ hdr is the header of the outgoing packet
50:     if hdr.isLongHeader then
51:         return                                    ▷ Continue only for short header
52:     end if
53:
54:     hdr.spinBit ← spinBit                         ▷ Set spin bit and delay bit
55:     hdr.delayBit ← delayBit
56:
57:     if delayBit = 1 then                          ▷ If this packet is a DS
58:         delayBit ← 0                              ▷ Reset DS control variable
59:         if time.Now() − lastDelaySample > time.Millisecond then   ▷ If delayed
60:             if not forceDelaySample then          ▷ And not forced
61:                 hdr.delayBit ← 0                  ▷ Abort reflection
62:             else
63:                 forceDelaySample ← False
64:             end if
65:         end if
66:     end if
67: end procedure
```

Looking at the pseudo-code shown above, it can be concluded that the delay sample algorithm is quite efficient and does not require large resources in terms of memory and CPU. By carefully analyzing the code, we can see that almost all the code necessary to manage the generation and reflection of the delay sample is actually performed once per period (i.e. on a single packet).

With respect to the procedure for incoming packets, the block comprised between line 16 and line 28 is executed only when a spin edge is detected by the client (i.e. the server never executes this block). The same applies to the block between line 35 and line 46, again this time executed once per period if the packet is a delay sample. For the rest, the procedure performs the simple spin bit reflection/inversion if the edge is not reordered. As regards the procedure for outgoing packets, the block included between line 57 and line 66, even in this case, is executed only if the packet is a delay sample.

3.5 Use of the Delay Sample by a passive observer

Unlike what happens with the spin bit for which it is necessary to validate or at least heuristically evaluate the goodness of an edge, the delay sample can be used by an intermediate observer as a simple demarcator between a period and the following one eliminating the ambiguities on the calculation of the RTT found with the analysis of the spin-bit only.

The measurement types obtainable from the observation of the delay sample are exactly the same achievable with the spin bit only (with or without the VEC).

End-to-end RTT measurement

The delay sample generation process — shown in section 3.2 — ensures that only one packet marked with the delay bit set to 1 runs back and forth on the wire between two endpoints per round trip time. Therefore, in order to determine the end-to-end RTT measurement of a QUIC flow, an on-path passive observer can simply compute the time difference between two delay samples observed in a single direction (Figure 3.2). Note that a measurement, to be valid, must take into account the difference in time between the timestamps of two *consecutive* delay samples belonging to adjacent spin-bit periods. For this reason, an observer, in addition to intercepting and analyzing the packets containing the delay bit set to 1, must maintain awareness of each spin period in such a way as to be able to assign each delay sample to its period and, at the same time, identifying those periods that do not contain it (see section 3.5.2).

Half-RTT measurement

An on-path passive observer that is sniffing traffic in both directions — from client to server and from server to client — can also use the delay sample to measure "upstream" and "downstream" RTT components. Also known as the half-RTT measurement, it represents the component of the end-to-end RTT concerning the paths between the client and the observer (upstream), and the observer and the server (downstream). It does this by measuring the delay between a delay sample observed in the downstream direction and the one observed in the upstream direction, and vice versa (Figure 3.2). Also in this case, it should verify that the two delay samples belong to two adjacent periods, for the upstream component, or to the same period for the downstream component.

Intra-domain RTT measurement

Taking advantage of the half-RTT measurements it is also possible to calculate the intra-domain RTT which is the portion of the entire RTT used by a QUIC flow to traverse the network of a provider (or part of it). To achieve this result two observers, able to watch traffic in both directions, must be employed simultaneously at ingress and

egress of the network to be measured. At this point, to determine the delay between the two observers, it is enough to subtract the two computed upstream (or downstream) RTT components as shown in Figure 3.3.

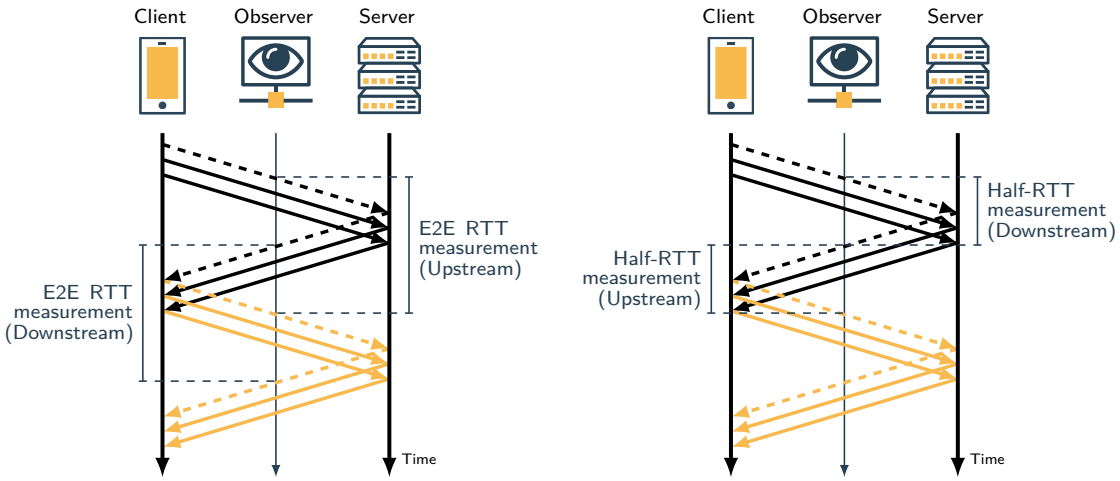


Figure 3.2. Practical example of how a passive observer can measure the end-to-end RTT and its components. Black and orange arrows for spin bit 0 and 1, respectively. Dashed arrows when the delay bit is set to 1.

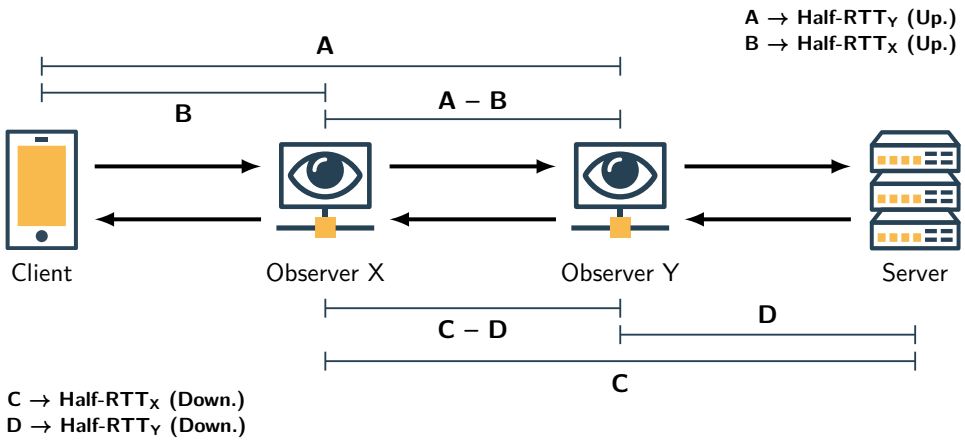


Figure 3.3. Practical example of how two passive observer, placed at ingress and egress of provider's network, can measure the Intra-domain RTT exploiting their end-to-end RTT components.

3.5.1 Observer's algorithm

Given below is a formal summary of the functioning of the observer every time a delay sample is detected. A packet containing the delay bit set to 1:

- if it has the same spin bit value of the current period and no delay sample was detected in the previous period, then it can be used as a left edge (i.e., to start measuring an RTT sample), but not as a right edge (i.e., to complete and RTT measurement since the last edge). If the observation point is symmetric (i.e., it can see both upstream and downstream packets in the flow) and in the current period a delay sample was detected in the opposite direction (i.e., in the upstream direction), the packet can also be used to compute the downstream RTT component.
- if it has the same spin bit value of the current period and a delay sample was detected in the previous period, then it can be used at the same time as a left or right edge, and to compute RTT component in both directions.

3.5.2 The Waiting Interval

Like stated previously, every time an empty period is detected, the observer must restart the measurement process and consider the next delay sample that will come as the beginning of a new measure, then as a left edge. As a result, being able to assign the delay sample to the corresponding spin period becomes a crucial factor for the proper functioning of the entire algorithm.

Considering that the division into periods is realized by exploiting the spin bit square wave, it is easy to understand that the presence of spurious spin edges — caused by packet reordering — would inevitably lead the observer to overestimate the amount of periods actually present in the transmission. This results in a greater number of empty periods detected and the consequent decrease of the actual RTT samples achievable. Therefore, in order to maximize the performance of the whole algorithm, the observer must implement a mechanism to filter out spurious spin edges.

To face this problem the waiting interval has to be introduced. Basically, every time a spin bit edge is detected, the observer sets a time interval during which it rejects every potential spurious edge observed on the wire. While at the end of the interval it starts again to accept changes in the spin bit value. This guarantees proper protection against spurious edges in relation to the size of the interval itself. For instance, an interval of 5ms is able to filter out edges that have been reordered by a maximum of 5ms. Clearly, the mechanism does its job for intervals smaller than the RTT of the observed connection.

Chapter 4

Packet Loss Measurement

4.1 The general algorithm

In order to provide the QUIC protocol with a mechanism to passively evaluate the loss rate of a connection, Telecom Italia is proposing a new algorithm that, exploiting one of the two bit of the short header reserved for future purposes, would allow — in theory — passive measurability of the end-to-end round trip packet loss.

The loss bit, as called by Telecom Italia, is a single bit signal whose purpose is to mark a variable number of packets which are exchanged two times between the endpoints realizing a two round-trip reflection. The overall exchange comprises:

(a) by the client, the generation and consequent transmission to the server of a first train of packets having the loss bit set to 1;

(b) by the server, upon reception from the client of each one of the packets included in the first train, their reflection to the client by means of a respective second train of packets (i.e., of size equal to the first train received) having the loss bit set to 1;

(c) by the client, upon reception from the server of each one of the packets included in the second train, their reflection to the server by means of a respective third train of packets (i.e., of size equal to the second train received) having the loss bit set to 1; and

(d) by the server, upon reception from the client of each one of the packets included in the third train, their final reflection to the client by means of a respective fourth train of packets (i.e., of size equal to the third train received) having the packet loss field set to 1.

A passive on-path observer, placed on whatever direction, can trivially *count and compare* the number of marked packets seen during the two round-trip exchanges (i.e., the first and third or the second and the fourth trains of packets, depending on which

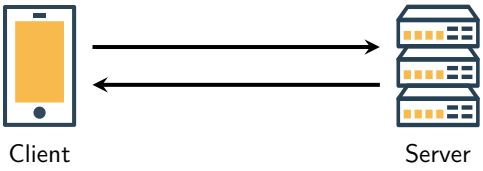
direction is observed) estimating the loss rate experienced by the connection. This process is repeated continuously so as to obtain more measurements as long as the endpoints exchange traffic.

Regarding its implementation within the QUIC protocol, the loss bit would be placed in the partially unencrypted (but authenticated) short header, in addition to the spin and delay bits, occupying the third and last bit reserved for future purposes. Accordingly, the proposed scheme of the first header byte, including both signal suggested by TiLAB, would be 01SDLKPP.

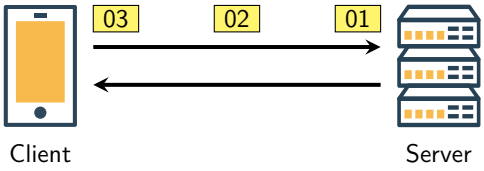
4.1.1 Problems to be addressed

The general algorithm shown above gives an idea of its underlying principles but is not enough to make the whole process working properly. First of all, we should consider the fact that packet rates in the two directions may be different. Therefore, the right number of packets to be marked has to be chosen in order to avoid their congestion on the slowest traffic direction. Secondly, a mechanism to individually identify each train of packets must be provided to enable the observer to distinguish between trains belonging to different rounds.

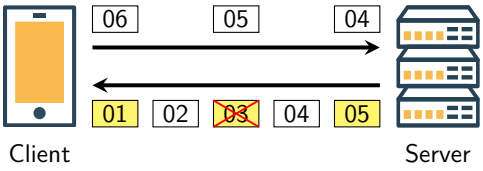
Next sections try to give a reasonable solution to these problems outlining, at the same time, a possible implementation of the algorithm.



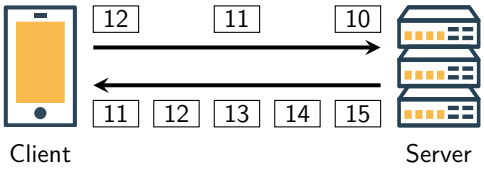
(a) No traffic at beginning.



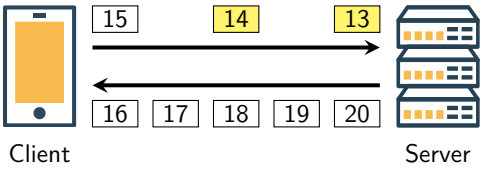
(b) The Client starts sending data and marks a set of packets setting their delay bit to 1.



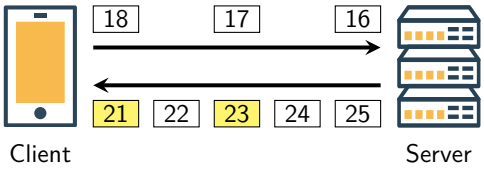
(c) The Server starts sending data and reflects marked packets received by the client.



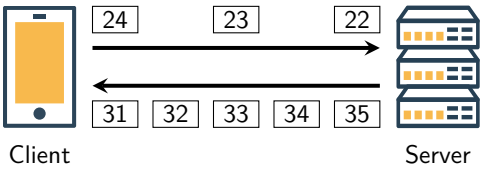
(d) The Client introduces a pause during which packets are sent with delay bit set to 0. The Server has no marked packets to be reflected.



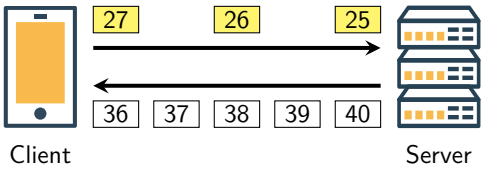
(e) The Client starts its reflection phase marking a new set of packets of the same size as those received (i.e., shortened of lost ones).



(f) The Server reflects marked packets received by the client.



(e) The Client introduces a pause during which packets are sent with delay bit set to 0. The Server has no marked packets to be reflected.



(f) The Client restarts the whole process to enable a new measurement.

Figure 4.1. Schematic representation of the loss bit mechanism. Packets are represented with packet numbers and different background colors: white for unmarked packets (delay bit set to 0) and yellow for marked ones (delay bit set to 1). Red crossed packet signals a loss.

4.2 Choosing the packets to be marked

In order to provide an good estimate of the round-trip packet loss which takes into account the different packet rates in the two directions, the measurement has to be be performed on the same number of packets in both directions. As a consequence, this number is inevitably equal to the amount of packets transited on the slowest direction. For example, if 100 packets are transmitted in one direction (e.g., downstream) during a certain marking period and 10 packets are transmitted in the opposite direction (e.g., upstream) during the same marking period, the measurement in the downstream direction will be performed on 10 packets only among the 100 transmitted ones.

The above problem can be easily addressed by a method wherein the two endpoints of a communication exchange marked packets interleaved with unmarked packets. By doing so, three different scenarios can be foreseen: the first one in which the slower endpoint sends a contiguous train of marked packets as opposed to the faster node which sends marked packets interleaved with other unmarked. The second scenario in which both endpoints transmit at the same rate and presumably have both all marked packets. Finally, the third in which, due to variable transmission rates, both endpoints transmit the two types of packets. As a result, depending on the scenario analyzed, the accuracy of the loss rate estimation varies accordingly.

From an implementation point of view, this result can be achieved by introducing a counter that literally counts the incoming packets aligning the outgoing ones to this value. Basically, the counter is incremented every time a packet arrives and decremented when a marked packet is transmitted. Since the creation of the initial train of marked packets is carried out by the client, the management and use of this counter is also assigned to it, which in fact "calculates" the correct number of packets to be marked each time. Note that to avoid congestion in the opposite endpoint (i.e., the server) in case its transmission rate suddenly decreases, it is necessary to calibrate the counter in order to limit the number of marked packet that can be sent by the client.

Consider, for example, the case in which client and server begin to transmit at a rate of 10 and 100 packets per marking period, respectively. When the client transmits the third marked packet, it has accumulated an average of 30 incoming packets; accordingly, the counter shows a value of 27 (i.e., 30 packets received minus 3 sent). At this point, if suddenly and for any reason the two transmission rates were reversed, the client would end up with 27 tokens to be used to transmit marked packets that would inevitably congest the server that could no longer reflect them in the opposite direction because of its new transmission rate.

To address this issue, the counter must be limited at the top so that its value can not exceed a certain safety threshold. In the most conservative case, one could think of using a boolean variable so that if multiple packets arrive before sending a marked packet in the opposite direction, only this one is sent; thus, the next one has to wait for

an additional incoming packet that trigger its transmission. However, an upper limit of 2 or 3 is probably more appropriate as it makes possible to absorb any small variations in the transmission frequency — which could lead to the formation of marking holes in the slow direction reducing the number of packets usable to estimate the loss rate — while protecting the server from possible congestion during reflection in case transmission rates change significantly.

4.3 Identify the different trains of packets

In the introductory explanation, to give a precise idea of how a passive observer can practically produce a packet loss measurement, four trains of packets were mentioned. However, figuratively speaking, it should be noted that actually a single client generated train of packets travels back and forth between the endpoints[1]. As a consequence, without taking the appropriate measures, this train would be seen by a hypothetical observer as a contiguous flow of indistinguishable marked packet (interleaved with others unmarked).

To solve this problem, having no other signaling bits available, the only usable methodology is to introduce a temporal pause during which no marked packets are forwarded so as to separate the different sequences of packets in an unequivocal way. Since the measurement by the observer is performed among marked packets that transit in the two consecutive rounds in the same direction (i.e., those seen in the first round-trip compared to those transited in the second round-trip), the client can simply isolate them introducing a double pause: one after the end of the generation phase and the other after the end of the reflection phase (i.e., between the second and the third train and between the fourth and the first train, according to the notation used in the first section).

The pause length is a crucial factor for the proper functioning of the algorithm: it should be large enough to enable the observer to easily identify the train of packets belonging to the different phases of the process but not excessively large to significantly decrease the ratio of marked packets to the total of those transited during the duration of the entire algorithm. In this case, the RTT of the connection could be a good solution because it can be easily determined by both the client and the observer — exploiting for example the delay sample or the spin bit — and it is theoretically big enough to absorb potential delays in reflection caused by transmission rate variation at the endpoints. However, to simplify the observer implementation and to prevent with greater confidence delays in reflection, a version of the algorithm that uses a long pause twice the RTT will also be tested.

[1] Obviously, these are not the same exact packets but an equal number of packet (except for those lost) marked each time in each direction.

4.4 The complete algorithm

Taking note of all the considerations made in the previous sections, a complete algorithm can be expressed as follows. Similar to what was done with the delay sample mechanism, even for the loss-bit algorithm the complexity clearly hangs towards the client.

4.4.1 How client marks packets

Client's tasks can be summarized into two phases: generation and reflection. The generation phase is in charge of generating at the correct rate a train of marked packets exchanged between the endpoints. While, the reflection phase re-proposes in transmission the same number of packets (clearly shortened of lost ones) realizing a second round-trip which enables passive measurability of the end-to-end round-trip loss rate.

Client part of the algorithm can be expressed in six consecutive points. Version A of the algorithm involves the use of a pause equal to the single RTT and the introduction of some tricks to optimize performance in those contexts in which delays in reflection may arise. Version B instead uses a pause equal to twice the RTT.

1a. the client starts generating marked packets and, after an interval equal to $RTT/2$[2], it listens to the reception channel waiting for the first marked packet reflected by the server. The client also maintains a generation counter that is incremented every time a marked packet arrives and decremented when another one is forwarded. If this counter reaches the zero value, the generation process is paused (i.e., outgoing packets are transmitted unmarked) and resumes as soon as its value returns positive;

1b. the client starts generating marked packets listening, at the same time, to the reception channel waiting for the first marked packet reflected by the server. The client also maintains a generation counter that is incremented every time a marked packet arrives and decremented when another one is forwarded. If this counter reaches the zero value, the generation process is paused (i.e., outgoing packets are transmitted unmarked) and resumes as soon as its value returns positive;

2. when the first marked packet return back to the client, this stops generating marked packets and compute the RTT of the connection as the difference in time between the timestamp of the first generated marked packet and the one just received. In addition, the generation counter is set back to 0 while the reflection counter is initialized to 1 and incremented for each marked packet received;

[2]Since the reflection phase does not necessarily last as long as an RTT but varies in relation to the traffic actually present in the two directions, this waiting semi-period introduces a good safety margin that should prevent a delayed reflected marked packet to trigger the end of the generation phase (i.e., the second point of the algorithm).

3a. using the RTT just calculated, the client introduces a first pause of equal duration during which the outgoing packets are forwarded with the loss bit set to 0. Incoming marked packets still increment the reflection counter;

3b. the client introduces a first pause of duration twice the RTT just calculated during which the outgoing packets are forwarded with the loss bit set to 0. Incoming marked packets still increment the reflection counter;

4a. then, the client starts reflecting marked packets — realizing the second round-trip transmission — and, after an interval equal to $RTT/2$[3], it stops incrementing the reflection counter on marked packets reception;

4b. then, the client stops incrementing the reflection counter and starts reflecting marked packets realizing the second round-trip transmission;

5. when the first reflected marked packet return back to the client, this compute the updated RTT of the connection as the difference in time between the timestamp of the first reflected marked packet and the one just received. The reflection process continues until the reflection counter is zeroed;

6a. finally, the client introduces a second pause of duration equal to the newly computed RTT during which the outgoing packets are forwarded with the loss bit set to 0. The whole process restart going back to the first point.

6b. finally, the client introduces a second pause of duration twice the newly computed RTT during which the outgoing packets are forwarded with the loss bit set to 0. The whole process restart going back to the first point.

4.4.2 How server marks packets

As seen in the general algorithm section, the server is responsible for reflecting every incoming train of packets sent by the client in the opposite direction. Basically, this can be done by maintaining a reflection counter incremented every time a marked packet arrives and decremented when another one is transmitted. It should be noted that, as already mentioned, the server does not maintain any status regarding the transmission rate control of marked packets. This means that whenever any packet has to be sent out, the server check the reflection counter to determine whether the packet need to be marked or not. Nothing more than that. This same behaviour is applied for both algorithm variants.

[3]Even in this case, a safety interval is introduced to enable the detection and consequent reflection (by the client) of those marked packets delayed during server reflection.

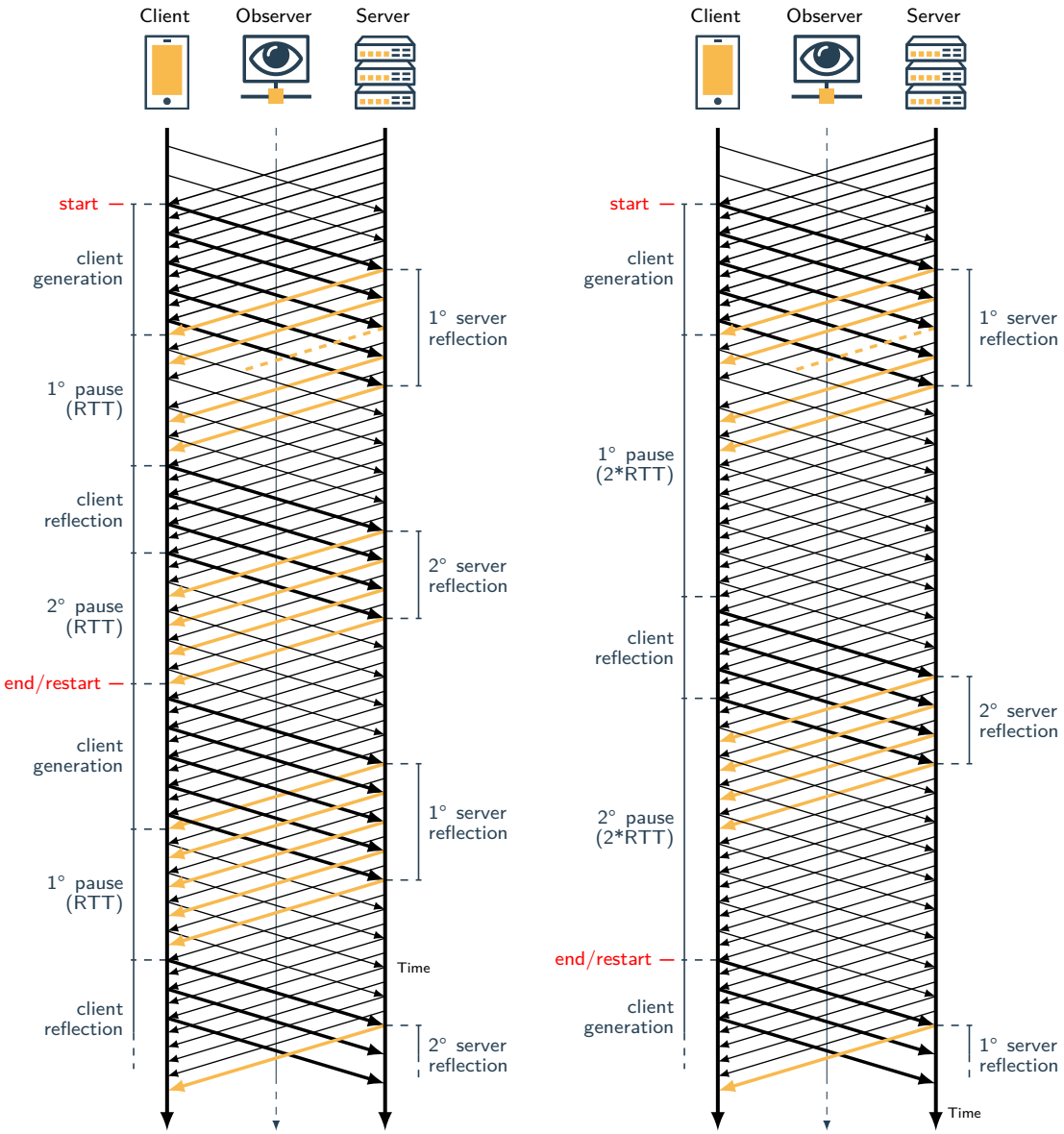


Figure 4.2. Timing schema of the loss bit algorithm. Packets are represented by arrows, thick and normal respectively for marked and unmarked. Different colors are used to distinguish between upstream and downstream marked packets. Dashed arrows for lost packets. The server transmits at twice the speed of the client.

4.5 Loss Bit Implementation

The following pseudocode shows a possible implementation — not necessarily an efficient one — of the single loss bit algorithm (variant A):

```
 1: lossPause ← True                                          ▷ Connection related variables
 2: reflectionPhase ← False
 3: genPktCounter ← 0
 4: rflPktCounter ← 0
 5: firstRflPktRcvd ← False
 6: firstPktTime ← 0                                          ▷ In time
 7: lossPauseEndTime ← 0
 8: halfRttEndTime ← 0
 9: lossRttTime ← 0
10:
11: procedure On Incoming Packet(hdr)                         ▷ hdr is the header of the incoming packet
12:     if client perspective then                            ▷ Handle loss bit on client context
13:         if hdr.lossBit = 1 then
14:             if lossPause then
15:                 if reflectionPhase then
16:                     rflPktCounter ← rflPktCounter + 1
17:                 end if
18:             else if not reflectionPhase then
19:                 now := time.Now();
20:                 if now > halfRttEndTime then
21:                     genPktCounter ← 0
22:                     rflPktCounter ← 1
23:                     reflectionPhase ← True
24:                     lossPause ← True
25:                     lossRttTime ← now − firstPktTime
26:                     lossPauseEndTime ← now + lossRttTime
27:                     halfRttEndTime ← lossPauseEndTime + lossRttTime / 2
28:                 end if
29:             else
30:                 now := time.Now();
31:                 if now <= halfRttEndTime then
32:                     rflPktCounter ← rflPktCounter + 1
33:                 else if not firstRflPktRcvd then
34:                     firstRflPktRcvd ← True
35:                     lossRttTime ← now − firstPktTime
36:                 end if
37:
38:                 if rflPktCounter == 0 then
39:                     reflectionPhase ← False
40:                     lossPause ← True
41:                     lossPauseEndTime ← now + lossRttTime
42:                     halfRttEndTime ← lossPauseEndTime + lossRttTime / 2
43:                 end if
44:             end if
45:         end if
46:
47:         if not reflectionPhase then
```

```
48:             if genPktCounter < 2 then
49:                 genPktCounter ← genPktCounter + 1
50:             end if
51:         end if
52:
53:     else if hdr.lossBit = 1 then                    ▷ Handle loss bit on server context
54:         rflPktCounter ← rflPktCounter + 1
55:     end if
56: end procedure
57:
58: procedure On Outgoing Packet(hdr)                   ▷ hdr is the header of the outgoing packet
59:     hdr.lossBit ← 0                                 ▷ loss bit not set by default
60:
61:     if client perspective then                      ▷ Handle loss bit on client context
62:         if lossPause then
63:             now := time.Now();
64:             if now > lossPauseEndTime then
65:                 firstPktTime ← now
66:                 lossPause ← False
67:                 if reflectionPhase then
68:                     firstRflPktRcvd ← False
69:                     rflPktCounter ← rflPktCounter − 1
70:                 end if
71:                 hdr.lossBit ← 1
72:             end if
73:         else
74:             if reflectionPhase then
75:                 if rflPktCounter > 0 then
76:                     rflPktCounter ← rflPktCounter − 1
77:                     hdr.lossBit ← 1
78:                 end if
79:             else if genPktCounter > 0 then
80:                 genPktCounter ← genPktCounter − 1
81:                 hdr.lossBit ← 1
82:             end if
83:         end if
84:
85:     else if rflPktCounter > 0 then                  ▷ Handle loss bit on server context
86:         rflPktCounter ← rflPktCounter − 1
87:         hdr.lossBit ← 1
88:     end if
89: end procedure
```

As expected, there is a clear difference difference in quantity of code destined for the two endpoints: while the client is in charge of managing the entire generation logic including pauses management, the server merely reflects marked packets as they arrive without carrying out any other operation that would weigh down its work. Regarding the procedure for incoming packets, almost all the codes is executed only when the received packet contains the loss bit set to 1.

4.6 Use of the Loss Bit by a passive observer

An on-path passive observer, looking at the loss bit value of each packet, can estimate the end-to-end round trip loss rate of a QUIC connection. Since a train of marked packets travels back and forth two times between client and server, the observer placed on whatever direction can count them and finally compare the number of packets seen during the first round with the number of packets seen during the second round.

To achieve this result, the observer need to maintain at least two counters: one for the first train of generated packets and the other for the second train of reflected packets. In addition, to correctly detect pauses that separate the different trains of packets, it must maintain the timestamp of the last seen marked packet so as to be able to compute the time elapsed between two consecutive marked packets. Finally, it has also to maintain the state of the algorithm (i.e., if observed packets belong to the generation or the reflection phase).

In details, when a packet containing a loss bit is observed, the time elapsed from the last seen marked packet is computed:

- if it is less than $RTT/2$ (for the A variant of the algorithm) or less than RTT (for the B variant), the packet belong to the current train of packet then the current counter is incremented;
- if it is greater than $RTT/2$ (for the A variant of the algorithm) or greater than RTT (for the B variant), a pause has been detected and the packet belong to a new train of packets. At this point, if the algorithm is in the reflection phase, the two counters are compared to determine the number of packets lost. Finally, the value of the current counter is copied into the previous counter, the current counter is set back to 1 and the state of the algorithm is inverted.

Note that when a pause is detected, if the current counter is greater than the previous one, then surely the just ended train of packets has been produced by the client during its generation phase. Accordingly, the observer can use this information to update its internal state of the algorithm just in case a de-synchronization occurred.

Since the measurement in question is of end-to-end round trip type, as already seen for the delay measurement, even for the packet loss it is possible to calculate the loss rate components between Observer and Client and between Observer and Server. It follows that the intra-domain measurement can also be performed using these components in the same way seen in the previous chapter.

Chapter 5

Software Observer

Together with the implementation of the two algorithms introduced in the previous chapters, a software observer has been developed to evaluate their functioning and performance. The main purpose of the observer is to analyze the traffic flow produced by a QUIC connection by measuring the round-trip delay and the loss rate.

The observer was developed using the C++ programming language which, thanks to its efficiency, guarantees performance that is clearly superior to other programming languages with a high level of abstraction. To speed up the implementation process, the PcapPluPlus[1] library was used in support. PcapPlusPlus is a multi-platform C++ network sniffing and packet parsing framework which is meant to be lightweight, efficient and easy to use. In our case, the observer runs in a Linux environment and for this reason, PcapPluPlus relies on the libpcap system library to capture traffic.

Quic Packet Analyzer, as it has been called, allows the analysis of different types of marked traffic. Firstly, it can compute the RTT and the half-RTT of a QUIC flow exploiting exclusively the information exposed by the spin bit. Secondly, it can improve the same measurements using the additional information exposed by the delay bit and the Valid Edge Counter. Finally, it can compute the loss rate of a QUIC connection by looking at the loss bit. In all cases, it is possible to select the transmission direction (i.e., upstream or downstream) for which to perform the measurements or choose to monitor both if symmetric computation is needed. Figure 5.1 shows a list of all supported marking methods.

QuicPA is designed to work in two different ways. The first involves the analysis of a capture file previously made using a network sniffing tool such as Tcpdump. The second modality instead allows the observer to directly sniff and analyse packets from a configured network interface in real time. In this last case, packets are captured asynchronously, meaning that the whole capture process is done in a dedicated thread

[1] https://github.com/seladb/PcapPlusPlus

	Spin Bit	Delay Bit	VEC	Loss Bit
01SRRKPP	✓			
01SDRKPP	✓	✓		
01SVVKPP	✓		✓	
0SDVVKPP	✓	✓	✓	
01SLRKPP	✓			✓

Table 5.1. Short header marking methods supported by the software observer.

different from the one in which the analysis takes place.

Packet analysis is performed sequentially. Whenever a packet is captured (or taken from the capture file) a callback function is executed and, based on the selected marking method, a data structure containing the essential information of the packet is generated. This includes the direction (whether the packet is client or server generated), the timestamp, the spin bit value and, optionally, the delay bit, the VEC and the loss bit values (extracted from the first byte of the QUIC short header). Each produced packet-info structure is then appended to the analysis queue ready for the next phase.

The analysis phase involves the processing of every single packet-info structure contained in the queue. According to the selected marking method, each dequeued packet-info structure is computed by different modules (each of which uses the respective observer computation rules discussed in the previous chapters), one for each marking method, thus producing the RTT and loss rate samples. All the measurements, as soon as they are produced, are then saved in different files organized by marking method, observation direction (i.e., upstream and downstream) and type of measurement (RTT, half-RTT, loss rate).

The observer also maintains additional data structures through which it produces useful statistics: with respect to the delay measurement, it stores the number of sampled periods and the average, maximum and minimum RTT (and half-RTT) found for each marking method; with respect to the loss measurement, it stores instead the number of packets generated, reflected and lost during the entire connection, and the percentage of marked packets in relation to the total packets transited per direction. Also these statistics are saved on a dedicated file.

Chapter 6

Evaluation

Using the software observer discussed in the previous chapter, the delay bit and loss bit algorithm have been tested to verify their functioning under different network conditions.

6.1 Testing environment

The whole evaluation has been performed by means of Mininet[1], a network emulation tool that using specific Linux Kernel features can emulate — on a single host — a complete network topology composed of virtual hosts, switches and links interconnecting them. Mininet internally uses NetEm[2] to introduce network impairments such as delay, jitter, reordering and loss. Delay is emulated by queueing and holding back all outgoing packets towards a specific interface for a fixed amount of time. Jitter, instead, can be introduced varying the delay of each packet (e.g., using a fixed delay of 10ms and a jitter of 1ms the resulting delay is sampled from the normal distributed interval $[9 - 11]$). However, being NetEm forward queues based on time, differently from what expected, jitter can lead to heavy packet reordering too. To emulate controlled reordering, a fixed fraction of packets can be delayed of a further amount of time (i.e., leaving the delay of the remaining part unchanged). Finally, loss is emulated by randomly dropping packets with a certain probability.

The emulated network used to perform all tests is shown in Figure 6.1. In realizing this specific topology, the same approach — with the same reasons behind it — used by Piet De Vaere in his "Adding Passive Measurability to QUIC"[9] was followed. It consists of seven nodes: two hosts, client and server, four switches and the observer, which actually is a switch too[3].

[1] http://mininet.org

[2] https://wiki.linuxfoundation.org/networking/netem

[3] Since in Mininet each node is a simple shell running on a linux network namespace, switches are

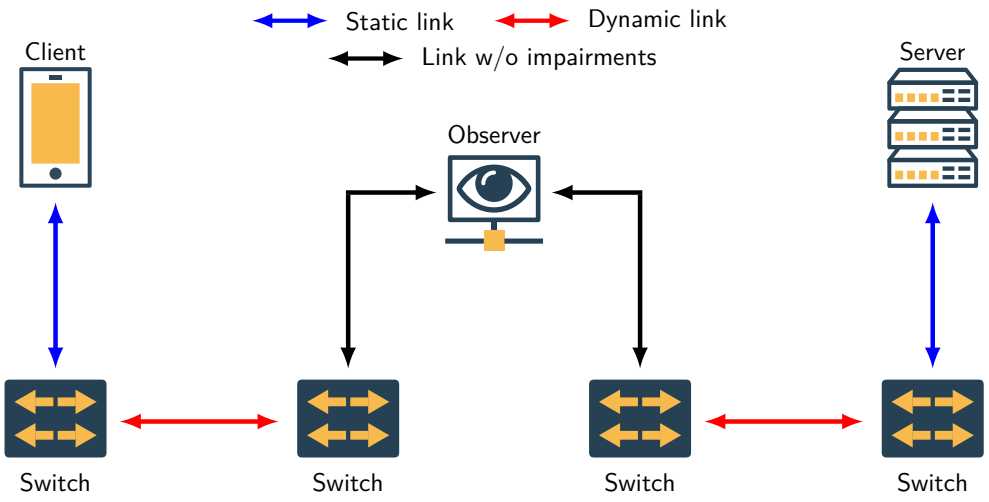


Figure 6.1. Network topology emulated using Mininet.

This number of switches is chosen for two reasons. Firstly, link impairments are applied on the interfaces at the edges of the link itself and, in particular, on Linux kernel outgoing queues of connected hosts. Since packet capture is done at the interface driver after NetEm traffic control manipulation, links directly attached to the observer should not be impaired in order not to alter captured traffic which would inevitably differ from the observer point of view. Secondly, NetEm does not support rules nesting thus making it impossible to introduce packet reordering alongside a fixed amount of delay on the same link[9].

Therefore, two impaired links are used on both sides of the observer enabling at the same time the introduction of all kind of impairments made available by Mininet. In detail, static links are in charge of delaying packets for a fixed amount of time while dynamic links introduce, from time to time, reordering and loss according to the test in progress. It should be noted that each configured impairment is actually applied twice per link, once for the upstream channel and another one for the downstream channel. For instance, setting a delay of 5ms for static links and a loss rate value of 2% for dynamic links results in an overall round-trip delay of 20ms and an overall loss rate of 8%.

nothing but nodes that run an OpenFlow switch instance. Then, in the case of the observer, the corresponding node is a switch that simultaneously performs packet capture.

6.1.1 Methodology

All tests performed require a certain amount of data being exchanged between client and server. These ones run a custom version of QuicGo modified to introduce delay bit and loss bit functionalities. Moreover, the server implements a simple HTTP server which enables the client to download a desired amount of data exploiting the QUIC protocol. Instead, the observer node runs Tcpdump[4] to capture traffic and write it to disk. Then, produced traffic files are analyzed using the software observer exposed on Chapter 5.

The emulated network is configured to introduce a base fixed delay of 40ms (i.e., static links have a configured delay of 10ms). Dynamic links configuration is specified on each performed test. Unless otherwise noted, all links are bandwidth limited to 1 Gbps. Charts are based on 200 Megabyte transmission.

Note that in order to obtain results as close as possible the ones obtainable on a real scenario, all tests are performed using the default settings of QuicGo. This means that no changes have been done on congestion control algorithm of QUIC protocol, then the transmission window size is free to change without any limitation according to experienced network conditions.

[4]http://www.tcpdump.org/

6.2 Delay bit evaluation

To test the operation of the delay bit algorithm, it was decided to compare the measurements produced by it with those obtained using the VEC algorithm which produces consistent samples in every network conditions. In this case, QuicGo has been modified to mark at the same time packets with both mechanisms. Being the number of bits available in the header less than the four required to simultaneously implement spin bit, delay bit and VEC, the second most significant bit of the first byte — always set to one and therefore useless in an experimental context — was sacrificed for the purpose. By doing this, it was possible to compare the two algorithms and their behaviour under the same network conditions.

6.2.1 Functionality without impairments

The first test proposed is a simple continuous transmission between client and server of 200 Megabyte of data. The emulated network does not present any impairments configured on dynamic links. Therefore, packets are just delayed of 40ms.

As can be seen from Figure 6.2, without reordering and loss applied all proposed methods produces almost the same measurements. By default, in fact, the delay bit algorithm marks as delay sample the first packet of the spin-period. This means that, in the absence of impairments, spin-edge, delay bit and VEC are always carried by the first packet of each period. As a consequence, RTT samples are computed taking into account the same edges for each measurement.

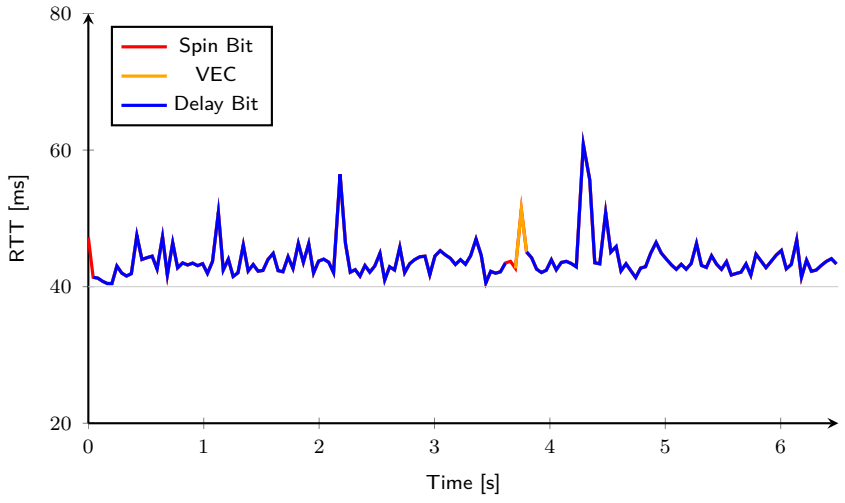


Figure 6.2. Chart showing computed RTT measurements using spin bit, delay bit and VEC observer without introducing network impairments. The network RTT is set to 40ms.

	Spin Bit		Delay Bit		VEC	
	Up	Down	Up	Down	Up	Down
RTT Samples	149	149	145	145	147	147
Avg. RTT (ms)	43.83	43.85	43.76	43.77	43.81	43.83
Min. RTT (ms)	40.46	40.38	40.46	40.38	40.46	40.38
Max. RTT (ms)	60.83	60.72	60.83	60.72	60.83	60.72
H-RTT Samples	149	149	145	147	147	149
Avg. H-RTT (ms)	21.55	22.29	21.50	22.26	21.51	22.28
Min. H-RTT (ms)	20.18	20.08	20.18	20.08	20.18	20.08
Max. H-RTT (ms)	34.36	40.49	34.36	40.49	34.36	40.49

Table 6.1. RTT statistics of 200MB transmission without network impairments.

Looking at the Table 6.1, it is possible to appreciate that the data collected by the three measurement systems are mostly the same. The only difference that stands out is the number of samples taken by each algorithm. The spin bit mechanism produces exactly one sample per application period: essentially it does not care about application or flow control limited traffic. In contrast, VEC and delay bit algorithms show less produced RTT samples because they skip the measurement every time a "hole" in traffic flow is detected. In this specific case, VEC produces 147 samples that is two samples less than those computed observing the spin bit only; whereas the delay sample produces 145 samples. This difference is to be attributed to the signaling mechanism embodied in the two algorithms. While the Valid Edge Counter is set back to 1 each time an edge reflection is delayed, the delay bit algorithm leaves an entire empty period in which no delay sample is produced. Moreover, while the VEC can be reset by both endpoints with immediate signaling to the observer, in the case of the delay bit, the empty period is inserted exclusively by the client which necessarily must first detect a failure in reflection of the delay sample in the event this one is caused by the server.

For these reasons, every time a delay sample is delayed — and therefore not reflected — the impact in terms of lost samples is on average twice the one perceived by the VEC algorithm in the same conditions (i.e., when an edge is delayed).

6.2.2 The effects of random loss

Being the RTT value determined by measuring the difference in time between two consecutive delay samples, packet loss affects the number of achievable measurements only when a delay sample is lost and it is therefore necessary to restart the generation process.

To evaluate the effects of random loss on delay bit functionality, dynamic links are configured to drop a specified percentage of packets. This percentage is varied from 0 to 5 per link to obtain an overall network loss rate comprised between 0% and 20%. As stated previously, congestion control is enabled and not limited.

Figure 6.3 shows how the computed average RTT value varies under different loss conditions. As expected, the solely spin bit is not enough to correctly calculate the RTT value of the network. The determined value increases with increasing loss rate due to the greater influence exerted by the QUIC recovery process which causes the entire application to slow down. The same problem cannot be found using the delay bit and the VEC algorithms as both use a time threshold in reflection. Their produced measurements are consistent with the configured network delay and perfectly comparable. This proves the reliability of the delay bit in the presence of even important losses.

The graph shown in Figure 6.4 instead offers an overview of the number of RTT samples obtainable using the two algorithms. Here two observations can be made.

Firstly, the Valid Edge Counter observer produces more valid RTT samples compared to the delay bit observer. Also in this case, this difference is to be attributed to the recovery process of the delay sample. When a delay sample loss is detected by the client (the only one in charge of performing this task), this one introduces an empty period in which no delay sample is transmitted. On the contrary, the VEC takes half round-trip to detect the loss of an edge (i.e., both endpoints can detect losses) and instantly restart the algorithm generating a new valid edge with VEC equal to 1.

Secondly, the upstream observer produces slightly more samples than the downstream counterpart. Since the client is downloading data from the server, the amount of packets transmitted in the upstream direction is significantly less than those sent in the opposite direction. For this reason, the client is more often led to generate traffic holes — which cause lacks of delay-sample/valid-edge reflection — than the server, especially in this case when the application is limited by QUIC recovery mechanism. As a consequence, the observer placed in the upstream direction will complete a greater number of measurements as the measurement process is restarted mostly by the client.

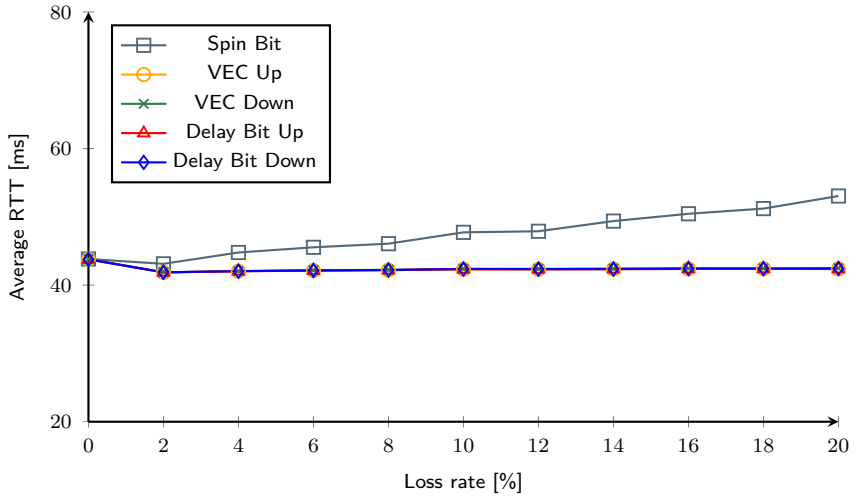


Figure 6.3. Average RTT value pace determined by each observer considering different random loss rates.

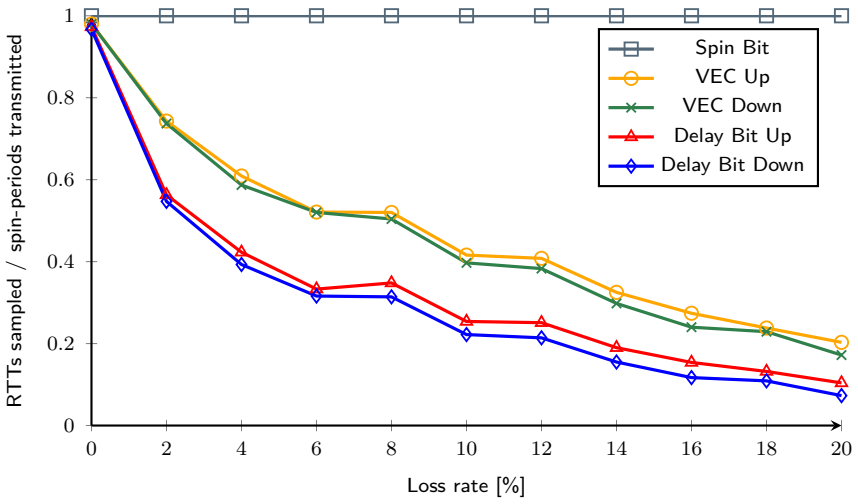


Figure 6.4. Number of RTT samples taken by each observer (normalized to the amount of spin-periods produced by the endpoints) considering different random loss rates.

6.2.3 The effects of reordering

As explained in Section 2.5.1, the spin bit alone in the presence of reordering can lead to the generation of really small fake periods and to the contraction of those adjacent. The VEC, as already demonstrated[9], allows the observer to reject spurious edges an detect lost edges. However, in case of reordering, its behaviour is highly restrictive to the point of discarding any edge that has been rearranged. In the case of the delay bit, being the measurement performed on a single packet (i.e., the delay sample) which is bounced between the two endpoints, no measurement are rejected in case of reordering.

To compare these two approaches, reordering is introduced into the emulated network. Each dynamic link forwards most packets immediately but hold back a set percentage of packets for 1 ms to emulate reordering. Each packet traverses two dynamic links, one before and one after the observer, and might therefore be reordered up to two times per direction.

Figure 6.5 shows the effects of different reordering rates on the computed average RTT value. First of all, the spin bit curves show how reordering seriously affects its reliability. Although on the upstream direction the average measurements are not too far from the real RTT value of the network, in the downstream direction the results worsen dramatically. This because in the upstream direction are forwarded fewer packets than in the downstream direction. For this reason, the distance between packets is higher while the probability of being reordered is lower.

Looking at the results produced by the other two algorithms, the VEC remains faithful to the real delay experienced by the network. The delay bit instead produces measurements whose RTT value tends to slightly increase as reordering rate increases. This is due to the fact that no measurement is discarded in case of packet reordering. As a consequence, with high reordering rates, the computed average RTT value is aligned to the base network delay plus the reorder delay (1 ms) introduced into the network. This approach guarantees results closer to the actual delay experienced by network users. On the contrary, discarding the measurements in case of reordering — as VEC does — undoubtedly produces measurements more faithful to the base network delay, but at the same time more distant from the experienced one.

Regarding the amount of samples achievable by the different observers, Figure 6.6 shows a significant overview. As expected, the spin bit observer produces more samples than those actually transmitted (also in this case the gap is greater for the downstream channel for the same reasons explained above). On the contrary, as it should be, the number of samples decreases significantly for both delay bit and VEC observers. The delay bit performs without any doubt better than the VEC as it guarantees a good number of samples even in conditions of heavy reordering (more than 25%). However, it is interesting to note that for small reordering values the difference between the two systems is minimal. This behavior is undoubtedly to be attributed to the continuous

transmission delays introduced by the congestion control which, experiencing reordering, tends to vary the transmission window size with consequent slowing down of the whole application; this leads inevitably to the introduction of traffic holes. In the presence of these ones, it has already been shown that the delay bit is slower in restarting the measurement process.

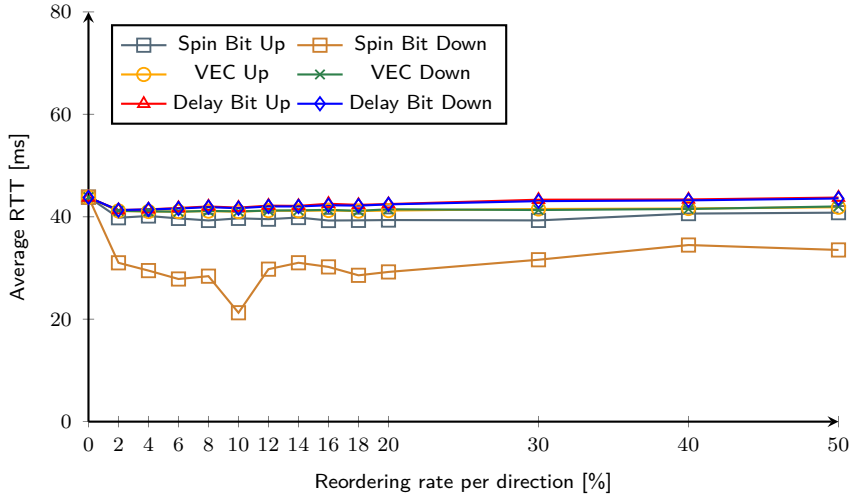


Figure 6.5. Average RTT value pace determined by each observer considering different reordering rates.

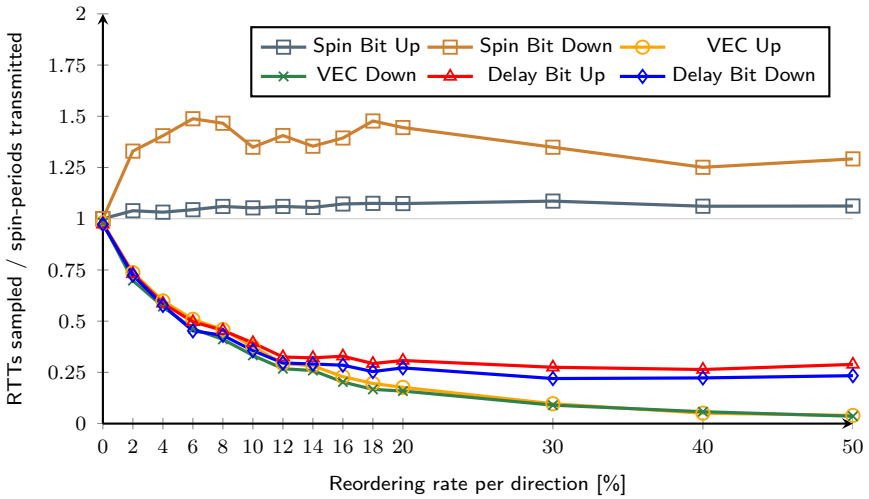


Figure 6.6. Number of RTT samples taken by each observer (normalized to the amount of spin-periods actually produced by the endpoints) considering different reordering rates.

6.3 Loss bit evaluation

Being the loss bit algorithm more complex to implement and debug, a simpler analysis was conducted. Also in this case, QuicGo has been modified to introduce its support alongside the spin bit signal used as the timing component of the algorithm (see Section 4.3). First of all, the resulting behaviour has been compared with the expected one just to verify the correctness of the implementation. Then, different tests have been performed in the presence of different random loss rate.

Note that the only scenario tested is the one where the client is downloading data from the server. Consequently, the transmission rates of the endpoints are kept constant, or at least they are not changed intentionally. The only factor that has influence on them is the congestion control which, detecting losses, varies the size of the transmission windows and therefore the transmission rates. The complete evaluation of all possible usage scenarios — especially those in which transmission rates are changed constantly — is left for future works.

6.3.1 First variant (A)

The A variant of the loss bit algorithm showed different problems during the testing phase. The single pause of RTT duration — introduced to separate different trains of packets — not always is correctly detected by the observer. In fact, when congestion control gets into action, it produces small variations in spin bit periods which causes in some cases packets belonging to the same train to be spaced more than half a RTT, the same amount of time used by the observer to detect a pause. This issue can be probably addressed introducing some sort of heuristic into the observer. However, it was preferred to shift the attention on the B variant of the algorithm that having a pause twice the RTT should not present this problem.

6.3.2 Second variant (B)

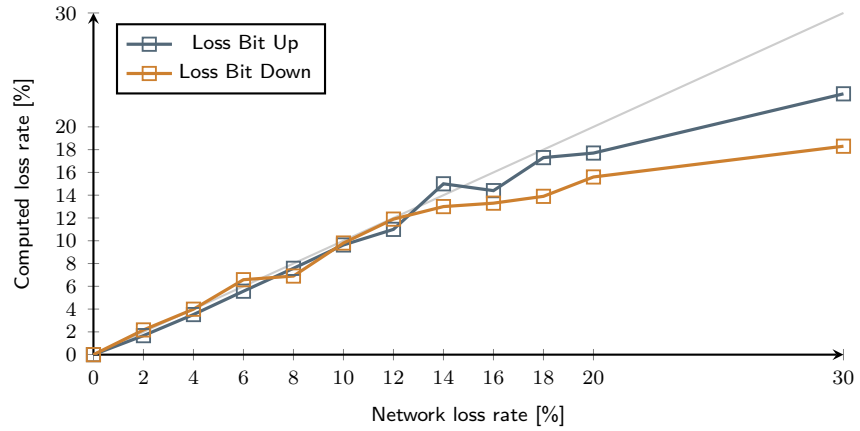
Contrary to the A variant of the loss bit algorithm, the B variant did not show the problems associated with the observer's pause detection. In this case the introduced pause is equal to twice the RTT computed by the client. Evidently, in the conditions highlighted above, this value is sufficient to guarantee a correct management of the pauses even in the presence of modest variations of the transmission rate.

To test the loss bit algorithm in its B variant, dynamic links are configured to drop a specified percentage of packets. Throughout all tests, this percentage is changed to obtain an overall loss rate experienced by the network in the range comprised between 0% and 30%.

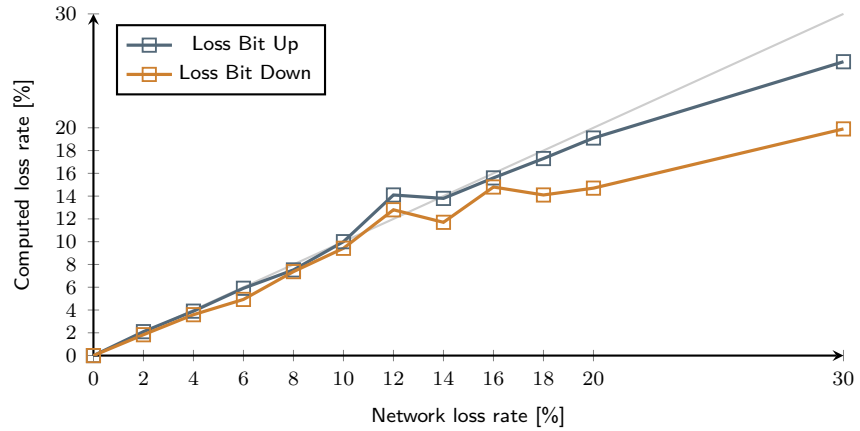
In order to perform the statistical computation of the loss rate, an entire flow of data

is analyzed (from beginning to end). Afterwards, the number of lost packets is determined by subtracting from the total of packets marked during all generation phases those marked during all reflection phases. Finally, the loss percentage is determined by considering lost packets in relation to marked packets associated with the generation phase.

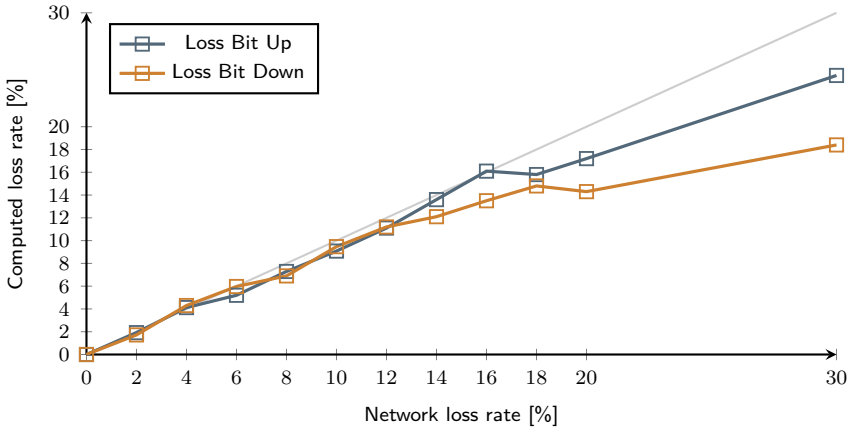
Figure 6.7 shows the statistical computed loss rate at different network loss rates for transmissions of size equal to 20, 50 and 100 megabytes. As shown in the graph, for values below 12% of loss rate, the algorithm produces positive feedbacks. The values are more or less in line with the loss actually introduced into the network (for all three tests performed). For values over 12%, the algorithm loses its effectiveness computing loss rates lower than expected.



(a) 20 Megabyte download



(b) 50 Megabyte download



(c) 100 Megabyte download

Figure 6.7. Computed loss rate determined by loss bit observer considering different configured overall network loss rates.

Chapter 7

Conclusion

This work aimed to provide the QUIC protocol with a complete support for performance measurement. Specifically, two algorithms have been analyzed. Firstly, we focused on the implementation and analysis of the delay bit, an additional delay signal theorized with the intent of improving the already present spin bit signal. Secondly, an initial analysis was performed on the use of a so called loss bit whose purpose is to enable a passive observer to compute statistical measurements of the loss rate experienced by the connection.

The two algorithms have been implemented on QuicGo (a QUIC protocol implementation) an evaluated on an emulated network using Mininet. It has been shown that both solutions, although not very simple from the implementation point of view, are completely sustainable considering also the fact that almost all of the computation is charged to the client.

The delay bit has been tested in different network conditions. The results produced by the related observer has been compared with those computed using the Valid Edge Counter. In the absence of network impairments, both solutions produce an excellent amount of RTT samples with a slight advantage on the VEC side as it can count on a rapid restart of the measurement process in case of traffic holes (determined by application limited traffic).

When losses are introduced, the amount of sample registered by the observers quickly deteriorates. The reasons behind this behavior are mainly two. On the one hand, we have the losses of delay-samples/valid-edges triggered by impaired links. On the other hand, the congestion control that causes the entire transmission to slow down with the consequent introduction of delays in the reflection of delay-samples/valid-edges. Both of these events lead the two solutions to frequently restart their measurement process, operation for which the VEC proves to be clearly faster (the delay bit takes on average a period and a half, while the VEC only half a period). As a consequence, the delay bit produces a lower quantity of RTT samples compared to the counterpart.

In contrast, by introducing reordering, a different trend was found. Although the VEC observer discards measurements every time an edge is reordered, with low reordering rates the two solutions are comparable and perform similarly. In the presence of high reordering rates, instead, the VEC shows a very high sample rejection rate, behavior not found in the delay bit observer which still continues to produce a good quantity of samples.

For the reasons explained above, it can be concluded that the delay bit works. It produces a good amount of RTT samples in almost every network conditions and, above all, it uses just one bit.

Regarding the loss bit algorithm, a simpler preliminary analysis has been conducted. Variant B of the algorithm clearly shows that a single bit can be used to compute a statistical estimate of the loss rate. In particular, loss rates below 12% are more or less correctly estimated. For higher values, instead, the algorithm tends to underestimate the actual loss rate experienced by the channel. Obviously, this approach needs to be extensively tested in different network scenario especially those in which endpoints vary constantly their transmission rates so as to confirm unequivocally its functioning.

Bibliography

[1] Iyengar J. and Thomson M., "QUIC: A UDP-Based Multiplexed and Secure Transport", IETF, Internet-Draft draft-ietf-quic-transport-17, Work in Progress, December 2018. [Online] Available at: https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-17

[2] Trammell B. and Kühlewind M., "The QUIC Latency Spin Bit", IETF, Internet-Draft draft-ietf-quic-spin-exp-01, Work in Progress, October 2018. [Online] Available at: https://datatracker.ietf.org/doc/html/draft-ietf-quic-spin-exp-01

[3] Trammell B. and De Vaere P. and Even R. and Fioccola G. and Fossati T. and Ihlar M. and Morton A. and Emile S., "Adding Explicit Passive Measurability of Two-Way Latency to the QUIC Transport Protocol", IETF, Internet-Draft draft-trammell-quic-spin-03, Expired and Archived, May 2018. [Online] Available at: https://datatracker.ietf.org/doc/html/draft-trammell-quic-spin-03

[4] Handley M., "Why the internet only just works", BT Technology Journal, Vol. 24 No. 3, p. 119-126, July 2006. [Online] Available at: https://doi.org/10.1007/s10550-006-0084-z

[5] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC Editor, RFC 8446, August 2018. [Online] Available at: https://rfc-editor.org/rfc/rfc8446.txt

[6] Thomson M. and Turner S., "Using TLS to Secure QUIC", IETF, Internet-Draft draft-ietf-quic-tls-17, Work in Progress, December 2018. [Online] Available at: https://datatracker.ietf.org/doc/html/draft-ietf-quic-tls-17

[7] Borman D., Braden B., Jacobson V. and Scheffenegger R., "TCP Extensions for High Performance", RFC Editor, RFC 7323, September 2014. [Online] Available at: https://rfc-editor.org/rfc/rfc7323.txt

[8] Fioccola G. and Capello A. and Cociglio M. and Castaldelli L. and Chen M. and Zheng L. and Mirsky G. and Mizrahi T., "Alternate-Marking Method for Passive and Hybrid Performance Monitoring", RFC Editor, RFC 8321, January 2018. [Online] Available at: https://rfc-editor.org/rfc/rfc8321.txt

[9] Vaere, Piet De and Mirja Kühlewind., "Adding Passive Measurability to QUIC", 2018. [Online] Available at: https://pub.tik.ee.ethz.ch/students/2017-HS/MA-2017-16.pdf

[10] Cociglio M. and Fioccola G. and Bulgarella F. and Sisto R., "New Spin bit enabled measurements with one or two more bits", IETF, Internet-Draft draft-cfb-ippm-spinbit-new-measurements-00, Work in Progress, February 2019. [Online] Available at: https://datatracker.ietf.org/doc/html/draft-cfb-ippm-spinbit-new-measurements-00