



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



Institut für
Technische Informatik und
Kommunikationsnetze

Piet De Vaere

Adding Passive Measurability to QUIC

Master Thesis MA-2017-16
September 2017 to April 2018

Tutor: Prof. Dr. Laurent Vanbever
Supervisor: Dr. Mirja Kühlewind
Supervisor: Brian Trammell
Supervisor: Tobias Bühler

Abstract

This work evaluates the addition of a latency “spin bit” to the QUIC protocol. The spin bit is set by the connection endpoints, and toggles once per RTT. Doing so allows on path observers to easily extract the RTT from a flow by measuring the duration between two spin bit transitions. Furthermore, up- and downstream delay can be measured separately as well. The functionality of the spin bit is evaluated on an emulated network. Under good network conditions the spin bit provides accurate and frequent RTT measurements. However, when network conditions worsen, the accuracy of the samples deteriorates. Various enhancements to counter these effects are proposed. One of them, the Valid Edge Counter, is a two bit extension to the basic spin bit that results in near perfect RTT measurements under all network and traffic conditions, albeit extreme network conditions can lead to high sample rejection rates.

Contents

1	Introduction	5
1.1	Motivation	5
1.2	Goals and overview	6
2	Background	7
2.1	QUIC	7
2.1.1	Header structure	7
2.2	MINQ	8
2.3	Passive measurement	9
2.4	Mininet, NetEm, Tcpdump and Tcpreplay	9
2.5	Gilbert-Elliot loss model	10
2.6	Vector Packet Processing (VPP)	10
3	Proposed methods	11
3.1	Handshake measurement	11
3.2	Explicit notification	11
3.3	Packet number echo	11
3.4	A <i>path</i> layer	12
3.5	Latency spin bit	12
4	The latency spin bit	13
4.1	Basic mechanism	13
4.2	Soundness	13
4.2.1	In the absence of reordering	14
4.2.2	In the presence of reordering	16
4.3	The observer’s perspective	20
4.3.1	Network delay measurement methods	20
4.3.2	The effects of network impairments and sparse traffic patterns	22
5	Observer flavours and spin enhancements	25
5.1	Spin bit observers	25
5.1.1	The basic observer	25
5.1.2	The packet number observer	26
5.1.3	The heuristic observers	26
5.2	Spin bit enhancements	27
5.2.1	A two bit spin value	27
5.2.2	The valid bit	28
5.2.3	The Valid Edge Counter (VEC)	29
6	Evaluation	33
6.1	Test setup	33
6.1.1	Network	33

6.1.2	Endpoint software	33
6.1.3	Observer	34
6.2	Results	35
6.2.1	Basic functionality	36
6.2.2	The influence of bursty traffic patterns	36
6.2.3	The influence of the packet scheduler	36
6.2.4	The influence of reordering	38
6.2.5	The influence of jitter	42
6.2.6	The influence of random loss	44
6.2.7	The influence of burst loss	46
7	Measuring other metrics	49
7.1	Alternative marking based loss measurement	49
7.2	An additional loss bit	49
7.3	Spin based reordering and loss measurement	49
7.4	VEC based loss and reordering measurement	50
8	Conclusion	51
	Bibliography	53
A	List of Acronyms	55
B	Additional plots	57

Chapter 1

Introduction

1.1 Motivation

Since its introduction in January 1980, TCP (Transmission Control Protocol) [1] has dominated the internet. And although the internet as a whole has changed dramatically over the last forty years, many of its core protocols — including TCP — have not seen major updates since their creation. In fact, the vast scale, complexity and economic interests of today’s internet have made it nearly impossible to introduce changes to its core protocols [2].

Therefore, many of the challenges that have arose over the past decades have been solved by applying clever hacks and bodes, rather than solving problems at their core. Not because the technical solutions were not there, but simply because coordinating their adoption has proven to be extremely troublesome. A classical example of this is the shortage of IPv4 (Internet Protocol Version 4) addresses, which has been solved by extensively applying NAT (Network Address Translation) throughout the internet — violating the fate sharing principle —, rather than adopting IPv6 (Internet Protocol Version 6).

Similar, but less well known, issues exists in the field of network manageability. As the internet’s traffic volumes and complexity has risen, so has the need for more advanced tools to manage it. An import part of this is being able to troubleshoot and monitor the network, which in turn requires operators to be able to perform measurements on the flows travelling over their network.

None of today’s prevailing network protocols were designed with this kind of measurability in mind, and as a result network operators have had to resort to hacks and bodes to perform measurement. These methods have to piggyback on information exposed for other purposes, often causing them to be overly complex, and producing only mediocre results.

An example of this is the passive measurement of the RTT (Round Trip Time) of TCP flows. As TCP has no explicit method of signalling RTTs to the network, a plethora of hacky RTT measurement methods have been developed. Some methods try to measure connection handshake RTT [3], others exploit the TCP timestamping option to link packets with the packets they trigger [4], and some even go as far as analysing packet inter-arrival times in the frequency domain [4, 5].

Although these methods have been able to serve the needs of network operators so far, the introduction of the QUIC (Quick UDP Internet Connections) transport protocol might soon change this. Contrary to TCP, and as can be seen in Figure 2.1, the vast majority of QUIC control information is encrypted, rendering acknowledgement or timestamp based RTT estimation methods useless. Furthermore, the use of pacing congestion controllers — such as Google’s BBR [6] — makes spectral analysis ineffective. If this situation is left as it is today, network operators could

soon face significant difficulties in keeping their networks operational, bringing the stability of the internet at risk.

However, this does not need to be the case. As QUIC is likely to be the first new transport layer protocol to see widespread adoption in almost forty years, it represents a rare opportunity to design a layer four protocol with explicit support for measurability. Doing this would not only ensure that network operators continue to have access to the information they need to debug their networks, but also makes it possible to do so while explicitly controlling which information gets exposed. Thus, allowing for a conscious trade-off between privacy and manageability.

1.2 Goals and overview

A network flow has three defining characteristics: delay, loss rate and throughput. As the throughput of a QUIC flow can trivially be measured by counting the number of (encrypted) bytes on the wire¹, the question remains how to measure the delay and loss rate. This thesis will focus on the former, leaving the later as future work.

Within the IETF (Internet Engineering Task Force), multiple methods for signalling RTTs to network elements have been proposed. A brief overview of these methods is provided in Chapter 3. As of yet, there is consensus that the use of a latency “spin bit” that toggles once per RTT is the most promising approach. However, as the spin bit is a new concept, it is still unclear how suitable it really is.

The goal of this work is to make an in depth analysis of the spin bit, and propose a measurement technique that

1. provides accurate measurements,
2. provides an observer with samples at reasonable intervals,
3. is robust to network impairments such as reordering, latency jitter and loss.
4. is straightforward to implement,
5. has a small header footprint,
6. is memory and processor efficient on the endpoints and observer,
7. does not expose additional, privacy sensitive information.

The theoretical analysis of the spinbit is done in Chapter 4. In Chapter 5 concrete observers and further enhancements to the spin bit are proposed. These observers are evaluated under emulation in Chapter 6. Finally, Chapter 7 gives a brief introduction on how the spin bit can be used to measure metrics different from delay.

¹The use of padding can complicate the measurement of application data, but from network management perspective it is desirable to also measure padding bytes.

Chapter 2

Background

2.1 QUIC

QUIC is a new transport layer protocol intended to serve as a replacement for the nearly forty years old TCP [1] as a transport protocol for HTTP (Hypertext Transfer Protocol). It was originally developed by Google [7], which then handed the development over to the IETF for standardization. Because this is still an ongoing effort, the QUIC specification is still a work in progress. Most of this thesis is based on revision 7 of the base specification drafts [8, 9, 10].

QUIC is a multiplexed protocol. This means that a single QUIC connection can carry multiple *streams*, each of which is a virtual, bidirectional, communication channel between the two connection endpoints. These streams do not suffer head-of-line blocking. That is, data loss on one stream does not affect the data transfer on other streams.

In order to be compatible with existing middleboxes and operating systems, QUIC packets¹ are transported in UDP (User Datagram Protocol) datagrams. Therefore, one could argue that QUIC is a layer 4.5 rather than a layer 4 protocol. In order to prevent middleboxes from mangling with packets, to prevent network ossification, and to protect user privacy, QUIC packets are fully authenticated, and largely encrypted, as can be seen in Figure 2.1.

A major implication of QUIC's protected payload, is that — by design — only very limited information is available in the publicly readable headers. While this is great for privacy, it makes the job of a network administrator significantly harder. After all, how can you diagnose a problem if you cannot observe what is happening on your network?

The next section discusses in detail what information is available in the public headers.

2.1.1 Header structure

The structure of an IP (Internet Protocol) packet carrying a QUIC packet is shown in Figure 2.1. QUIC packets have either a *long* (Figure 2.1(b)), or a *short* (Figure 2.1(a)) header. Typically, the long header format will be used during connection setup, after which only short headers are used to save bandwidth.

The IP and UDP headers are not protected, and can be read and modified by the network. The QUIC header is authenticated, but not encrypted and exposes the following information:

¹Fondly known as *quackets*.

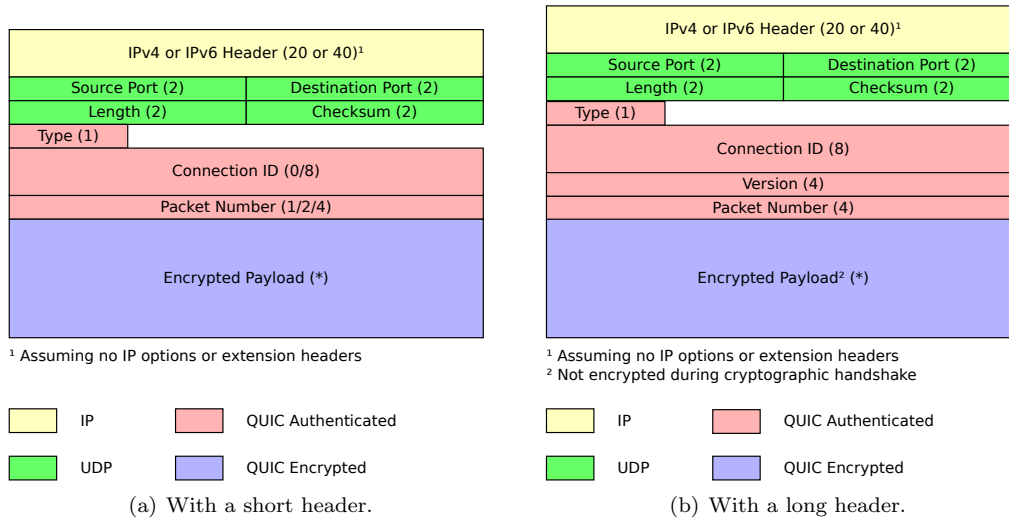


Figure 2.1: The structure of an IP packet carrying a QUIC packet. Field sizes are in bytes.

The type byte provides information on the structure of the remainder of the QUIC packet. It informs parsers whether a short or long header is used, whether the payload is encrypted or clear text, ...

The connection ID servers to uniquely identify a QUIC connection, and facilitates connection migration.

The version number identifies the version of the QUIC protocol in use.

The packet number identifies the QUIC packet. Both endpoints assign strictly increasing 8-byte packet numbers to all their outgoing QUIC packets, and include the least significant byte(s) in the QUIC header. Note that both endpoints use a separate packet number space, thus *each packet number may be used twice*: once in each direction of the connection. A retransmission is considered to be a new packet, and receives a new packet number.

Last in the packet is the QUIC payload. The term “payload” should be considered in the broad sense of the word, as not only application data, but also almost all QUIC control data is sent in this section. Because in the vast majority of transmissions the payload is fully encrypted, this data is not available to on path elements.

Although the current QUIC drafts specifies the packet number to be unencrypted, discussion is underway in the IETF to encrypt it as well. If this would happen, the packet number would likely be encrypted separately from the payload, and remain at a fixed position in the header.

2.2 MINQ

Minq (Minimal QUIC) is a minimal and partial implementation of the QUIC protocol. It is intended for testing and verification of the QUIC protocol stack, and evolves together with the QUIC specification. It is written in the Go programming language².

Minq is available under an MIT license at <https://github.com/ekr/minq>. A version modified for use during this thesis is available at <https://github.com/pietdevaere/minq>.

²<https://golang.org/>

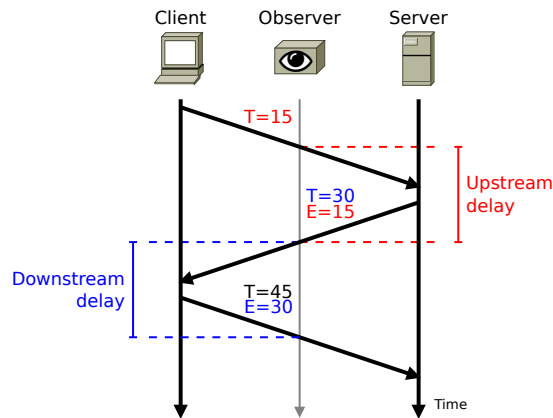


Figure 2.2: An example of passive measurement: by observing TCP timestamps (T) and their echos (E), a passive observer can measure the RTT of a flow.

2.3 Passive measurement

Passive measurement refers to the practise of measuring certain parameters of network flows simply by observing them. For example, a network operator might measure the RTT of a TCP flow by observing the sequence of timestamps and timestamp echos on that flow, as shown in Figure 2.2. In order for passive measurement to be possible, it is necessary that adequate information is exposed by the flow under measurement. Simply put, information that is not present, cannot be extracted. However, as the internet has historically been dominated by TCP traffic [11, 12], and because TCP transmits all its control information in clear text, network flows typically expose sufficient information to meaningfully measure their three defining characteristics: throughput, delay and loss rate.

With the advent of encrypted network protocols — such as QUIC — this situation is starting to change. As more and more network traffic is being encrypted, less and less information is exposed to the networks it traverses. This has the advantage that it protects user privacy and prevents packet meddling, but has the disadvantage that network operators lose valuable insight in what is happening on their networks. Depending on their security requirements, protocol designers may choose to resolve this by explicitly exposing some information (such as loss rate, RTT or throughput) to the network.

2.4 Mininet, NetEm, Tcpcdump and Tcpreplay

Mininet [13] is a network emulation tool. It uses features of the Linux kernel (e.g. processes and network namespaces) to create virtual hosts, links, and switches. By doing so, it can emulate entire networks on a single host.

Mininet uses NetEm [14] to emulate network impairments such as delay, reordering, jitter or loss. Delay is emulated by holding back all packets on an interface for a fixed amount of time. When jitter is added, the delay for each packet is sampled from a bounded normal distribution, rather than set to a fixed value. Therefore, jitter can also lead to packet reordering. However, packet reordering can also be explicitly specified. When this is done, a fixed fraction of packets will be forwarded without any delay at all. Loss can be emulated too. Either by randomly dropping packets with a set probability, or by using the Gilbert-Elliot loss model.

Tcpdump³ can be used to capture traffic from a network interface. It does so by using the Pcap (Packet Capture) libraries. In order to replay this traffic at a later time, Tcpreplay⁴ can be used.

2.5 Gilbert-Elliot loss model

Modern communication channels often exhibit bursty loss patterns. In order to simulate such loss patterns, the Gilbert-Elliot loss model [15, 16] can be used. The Gilbert-Elliot model is based on a two state Markov chain, as shown in Figure 2.3. When the Good state is active, packets are lost with a low probability l . When the Bad state is active, packets are lost with a high probability h . For every packet, the probability of transitioning from the Good to Bad state is g , and in the other direction the probability is b . In other words, the durations of good reception and loss periods follow a geometric distribution with parameters g and b respectively. This means that good reception and loss periods have expected lengths of $1/g$ and $1/b$ respectively.

When $l = 0$, this model is referred to as the Gilbert model, when $l = 0$ and $h = 1$, the model is known as the *simple* Gilbert model.

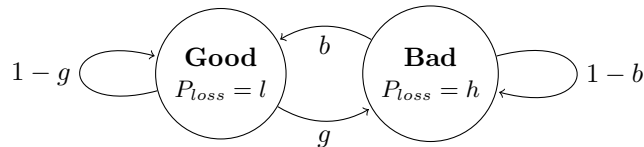


Figure 2.3: The Markov chain of the Gilbert-Elliot loss model.

2.6 Vector Packet Processing (VPP)

VPP (Vector Packet Processing)⁵ is a software framework for packet processing created by the Fast Data Project (FD.io). It processes packets by grouping them in *vectors*, and then passing these vectors through a processing graph. By doing so, the instructions for every graph node need to be loaded in to the instruction cache only once per vector, instead of once per packet. This leads to claimed speed improvements of up to two orders of magnitudes over other technologies.

³<https://www.tcpdump.org>

⁴<https://tcpreplay.appneta.com/>

⁵<https://fd.io/technology/#vpp>

Chapter 3

Proposed methods

A plethora of methods can be used to expose a flow's RTT to the network. This section provides a brief overview of five methods that have been proposed at the IETF, and the reason that they have been dismissed or are still under discussion. These methods are: handshake measurement, explicit inclusion, packet number echo, the *path* layer, and the latency spin bit.

3.1 Handshake measurement

QUIC handshakes can be recognized as such by observers. Because they follow a predefined pattern, it is easy to determine which handshake packets belong together. This makes it relatively straightforward to extract RTT information from them. However, as a handshake only occurs at the beginning of a connection, handshake measurement does not allow for RTTs to be tracked over time. Furthermore, during the handshake the protocol executes one-off tasks (e.g. establishing the crypto context for the connection) that will be included in the handshake RTT measurement.

3.2 Explicit notification

This is the most straightforward way to inform network elements of a flow's RTT: by placing a field in the header that contains the RTT as measured by the sending endpoint. However, this method also comes with a clear limitation: it is not possible to measure the delay upstream and downstream of an observation point separately. This greatly limits the use of delay information, as it does not facilitate network operators in determining whether bottlenecks are located in, or outside of their own network.

3.3 Packet number echo

The key reason that ACK (acknowledgement) or timestamp based TCP RTT estimation works, is that both ACK and timestamp analysis allows an observer to link packets with the response they elicit. As QUIC packet numbers are unique, this can also be achieved by adding a packet number echo to the QUIC packet header. Noteworthy is that this method is also usable with encrypted

packet numbers, by considering the packet number as a *packet association token* rather than a sequence number¹.

This approach has two major downsides: firstly, in order to measure the full RTT, both directions of a flow need to be observed. Secondly, echoing back a packet number requires up to four bytes of header space. This can potentially lead to significant overhead on a QUIC connection.

3.4 A *path* layer

PLUS (Path Layer UDP Substrate) is protocol that runs on a proposed new network layer that is specifically aimed at facilitating communication between network endpoints and the network itself [17]. In its proposed form, the PLUS header contains a packet number and packet number echo field. Furthermore, it also provides other information, such as explicit notification of a connection closure. However, from a pure RTT measurement perspective, it has the same properties as a plain packet number echo, with the additional downside of having a larger header footprint.

3.5 Latency spin bit

The latency spin bit — or spin bit for short — is a bit placed in the QUIC header, that is toggled once per RTT. How this is accomplished is explained in Section 4.1. Because the signal requires only one bit, it can be placed in the QUIC type byte. Therefore the spin bit adds effectively zero overhead to the QUIC header size, which makes it an attractive option. However, many of the spin bit's properties are still unclear. For example: how does it perform under loss, or how accurate are the estimates it provides? This thesis aims to answer those questions, providing an in depth analysis of the spin bit's characteristics. This information can be used by the IETF to make an informed decision about the inclusion of the spin bit in QUIC.

¹This does require that both the encrypted packet number and the encrypted packet number echo are placed at a well known location in the header, and that the original packet number crypto text is echoed back, rather than a re-encrypted copy of the clear text packet number.

Chapter 4

The latency spin bit

4.1 Basic mechanism

The latency spin bit is — as the name implies — a single bit signal that can be used by unprivileged, passive observers to measure the RTT of a network flow [18]. The spin bit is placed in the unencrypted (but authenticated) QUIC header, and is set by the endpoints as follows:

The server sets the spin bit to the same value of the spin bit in the packet with the largest packet number it has thus far received from the client.

The client sets the spin bit to the opposite value of the spin bit in the packet with the largest packet number it has thus far received from the server. If no packet has been received from the server yet, the spin bit is set to ‘0’.

Note that because the client will always initiate the connection, it is not necessary to specify a default spin bit value for the server. Furthermore, the endpoints always have access to the packet numbers, even when they are encrypted during transmission.

When the client and server follow the above rules, each of them will generate one *spin bit transition* per RTT. By monitoring packets for these transitions, passive observers can measure the network RTT.

4.2 Soundness

The desired property of the latency spin bit is that each endpoint will generate one (and only one) spin bit transition per RTT. In order to formally show the soundness of the latency spin bit, some abstractions are made.

The network is abstracted to two queues. An *upload* queue that is written to by the client and read from by the server, and an *download* queue that is written to by the server and read from by the client. The endpoints write packets to these queues. A packet consists of two elements: a *packet number*¹ and a *spin bit*. They are represented as a box with a packet number and a spin indicated by an arrow (spin ‘0’ is down, spin ‘1’ is up) as follows: $\boxed{42 \downarrow}$. Packets that are in either of the two network queues are said to be *in transit*.

¹It is assumed that the packet numbers do not wrap around. In QUIC this does happen, but because the packet number is used as a decryption nonce, this has to be unambiguously clear to the endpoints. Therefore the arguments in this section still hold.

In order to simplify the argumentation, a synchronous execution model is considered. However, the arguments remain valid in an asynchronous model as well. Each endpoint does the following every cycle:

1. If possible, it reads one packet from its incoming queue.
2. It processes the packet and decides on the value of the spin bit for the next packet it generates. This happens as specified in Section 4.1.
3. It sends out one packet to its outgoing queue. The server only does this after it has read the first packet from the client.

The queues are also synchronous. For illustrative purposes they can hold five elements, and during each cycle all elements move one element towards the end of the queue. Only elements at the end of the queue can be read by endpoints.

4.2.1 In the absence of reordering

Soundness is first proven in the absence of in network reordering. Under this assumption, the network queues are simple FIFO (First In, First Out) queues.

In order to proof soundness, the notion of a *spin edge* is introduced:

Definition 1 (spin edge): When a packet has a spin value that is different from the spin value of the packet in front of it in the queue, this packet is said to transport a *spin edge*, or *spin bit transition*.

It is now proven that at any point in time, *at most* one spin edge is in transit.

Theorem 1: At any point in time, there is at most one spin edge in transit.

Proof:

1. Before the client initiates the connection, all network queues are empty (Figure 4.1(a)).
2. When the client starts sending, it transmits only packets with spin ‘0’, so no spin edges are in transit (Figure 4.1(b)).
3. Once the server receives the first packet from the client, it will start sending packets with spin ‘0’, so still no spin edges are in transit (Figure 4.1(c)).
4. When the first packet from the server reaches the client, the client wil start transmitting packets with spin ‘1’. At this point, one spin edge is in transit (Figure 4.1(d)).
5. In order for the client to generate another spin edge, it must transmit a packet with spin ‘0’. This will only happen when the client receives a packets with spin ‘1’, and the server will only transmit a packet with spin ‘1’ after it received one.

Just before the server receives the first packet with spin ‘1’, the upload queue will contain only packets with spin ‘1’, and the download queue only packets with spin ‘0’. Thus, no spin edges are in transit at this point (Figure 4.1(e)).

6. After the server transmits the first packet with spin ‘1’, the upload queue wil contain only packets with spin ‘1’, and one spin edge will be in transit in the download queue (Figure 4.1(f)).
7. Just before the client reads this packet, both the download and upload queue contain only packets with spin ‘1’, so at this point no spin edges are in transit (Figure 4.1(g)).

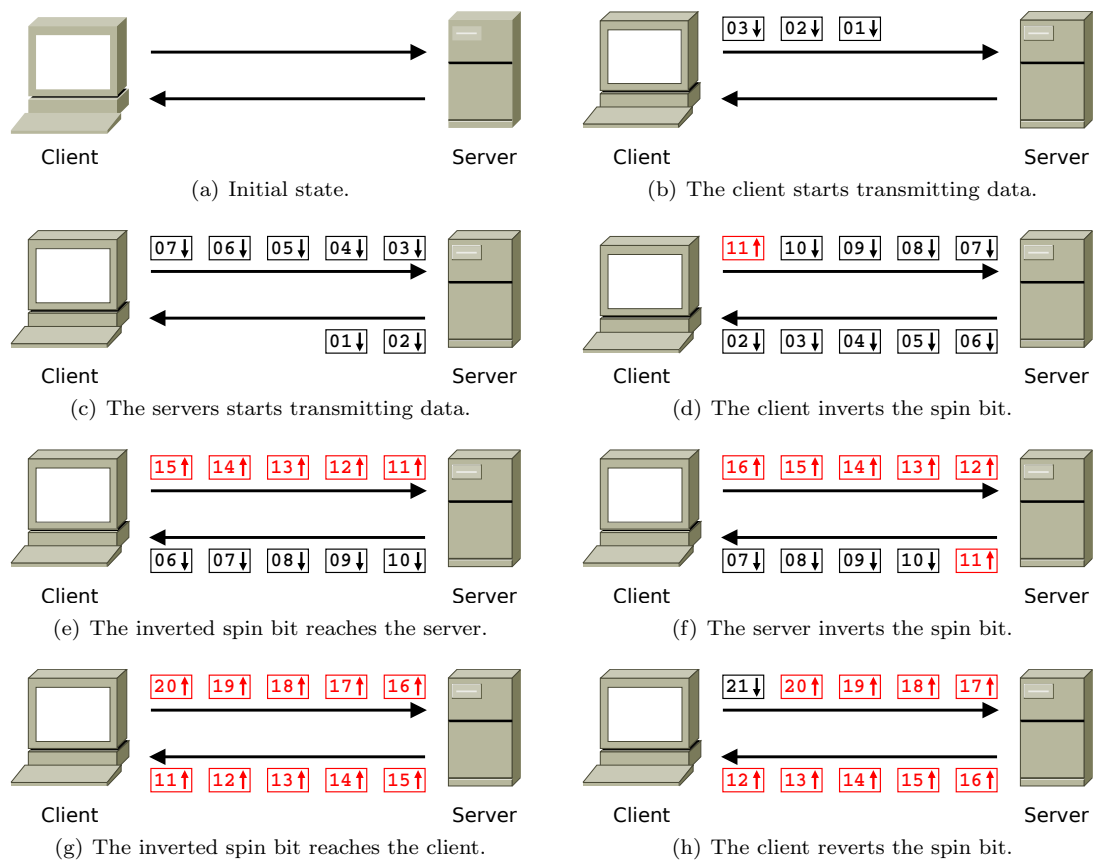


Figure 4.1: Schematic representation of the spin bit mechanism in the absence of reordering. Packets are represented as rectangles with a packet number and spin represented by an arrow.

8. When the client receives this packet, it will generate a packet with spin '0', placing a new spin edge in transit (Figure 4.1(h)). At this point, only one spin edge is in transit. This situation is analogous to the situation in step 4. Thus, by induction, there is always at most one spin edge in transit.

□

From this, it can be proven that both the server and client generate only one spin edge per RTT.

Theorem 2: The client generates one spin edge per RTT.

Proof: The client will (only) generate a spin edge after it received one from the server. The server will (only) generate a spin edge after it received one from the client.

When the client generates a packet **A** that carries a spin edge, Theorem 1 states that there can be no other spin edges in transit. Once **A** is read by the server, the server will generate a packet **B** that carries a spin edge in the download queue. This must again be the only spin edge in either of the network queues. Only when **B** has traversed the download queue and is read by the client, the client will again generate a spin edge.

Thus, after the client generated a spin edge, it will (only) generate the next spin edge after the first spin edge has traversed the upload queue, and the packet then generated by the server traversed the download queue. By definition, this takes one RTT.

□

Theorem 3: The server generates one spin edge per RTT.

Proof: The proof is analogue to the proof of Theorem 2.

□

4.2.2 In the presence of reordering

Now that soundness without reordering has been established, this can be extended to soundness under network reordering conditions. Again, the desired property is that both the client and server generate one spin edge per RTT.

The most important observation to proof this property, is that (when following the rules from Section 4.1) the endpoints will ignore the spin value of any packet that is read after a packet with a higher packet number has already been received. Thus, for any packet, if there is a packet with a higher packet number in front of it in the queue, or if such a packet has already been read by the receiving endpoint, its spin value is said to be *void*.

In order to reflect this, the definition of a spin edge is slightly changed:

Definition 1' (spin edge): When a packet contains a non void spin value, and this spin value is different from that of the nearest packet in front of it in the queue with a non void spin value, this packet is said to transport a spin edge or spin bit transition.

Thus, the following sequence contains a spin edge: $\boxed{03 \uparrow} \boxed{02 \downarrow} \boxed{01 \downarrow}$, and so does $\boxed{03 \uparrow} \boxed{01 \downarrow}$. But the next sequence does not: $\boxed{01 \uparrow} \boxed{03 \downarrow} \boxed{02 \downarrow}$.

Continued on the next page.

Theorem 1, 2 and 3 can now be proven under reordering.

Theorem 1’: At any point in time, at most one spin edge is in transit.

Proof:

1. Initially, both queues are empty, so no spin edges are in transit (Figure 4.2(a)).
2. When the client starts transmitting, it only sends packets with spin ‘0’, so no spin edges are in transit. When reordering occurs, no spin edges are created (Figure 4.2(b)).
3. Once the server receives the first packet from the client, it will start transmitting packets with spin ‘0’, so no spin edges are in transit. Also here reordering will not lead to the generation of spin edges (Figure 4.2(c)).
4. Once the first packet from the server reaches the client, the client will start transmitting packets with spin ‘1’. At this point one spin edge is in transit (Figure 4.2(d)).
5. When at any point, a packet gets reordered from a position behind the spin edge to a position in front of it, the original spin edge is void, and a new one is created (Figures 4.2(e) and 4.2(f)). Further packet reordering might again move the position of the spin edge, but will not lead to the generation of a new spin edge without another edge being voided. Thus, at any point, at most one spin edge is in transit.
6. Once the packet carrying the valid spin edge is received by the server, the spin bits of all packets with a lower packet number than the received packet become permanently void. They can have no more influence on the behaviour of the endpoints, and can be ignored during further analysis. The sender will now start generating packets with spin ‘1’. Because all packets generated by the client after packet carrying the spin edge, must have spin ‘1’, all packets with a non void spin value in the upload queue must have spin ‘1’, and thus no spin edges can be present in the upload queue. So again, one spin edge is in transit (Figure 4.2(g)).
7. As for the upload link, reordering cannot generate a second spin edge on the download link.
8. Once the spin edge has been received by the client, the client starts generating packets with spin ‘0’. Because all packets generated by the server after it generated the packet carrying the spin edge have spin ‘1’, all packets with non void spin values on the download link must have spin ‘1’. At this point only one spin edge is in transit. This situation is analogous to the situation in step 4. Thus, by induction there is always at most one spin edge in transit.

□

The validity of Theorem 2 and 3 under reordering follows from the validity of Theorem 1’.

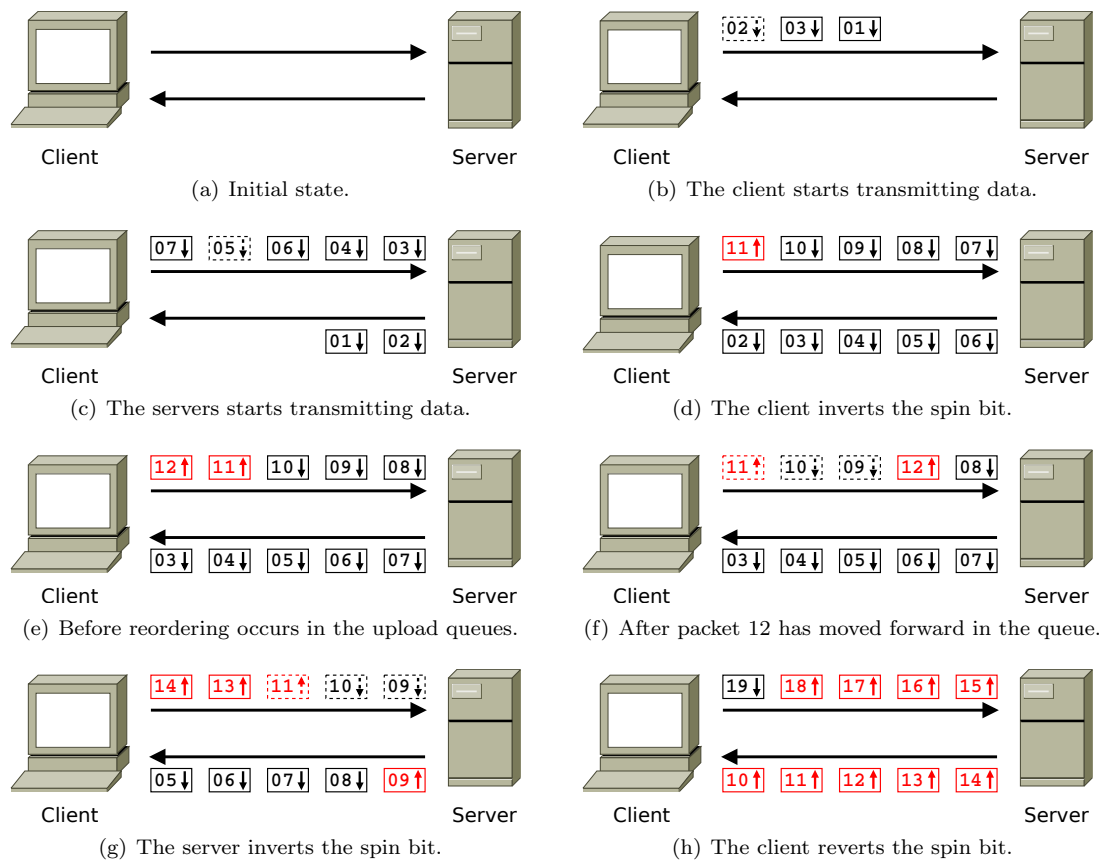


Figure 4.2: Schematic representation of the spin bit mechanism under reordering network conditions. Packets are represented as rectangles with a packet number and spin represented by an arrow. Packets with void spin values are drawn with dotted lines.

4.3 The observer's perspective

Contrary to the endpoints, an on path observer does not modify the value of the spin bits. In fact, doing so would break the header authentication, rendering the packet corrupted. However, by solely observing the values of the spin bits on a flow's packets, an on path observer can measure the (partial) RTT of that flow, as is discussed in Section 4.3.1. Unfortunately, impairments on the network or the absence of packets can reduce the accuracy of these measurements. This is discussed in Section 4.3.2.

4.3.1 Network delay measurement methods

Full-RTT measurement

By observing the spin values carried in a flow's packets, a passive, unprivileged observer can determine the RTT of that flow. In order to perform such a full-RTT measurement, only one of the directions of the flow needs to be observed. The observer then measures the duration between two spin bit transitions on the packets it observes. As each endpoint generates one spin bit transition per RTT (see Section 4.2), this measurement corresponds directly to the flow's RTT. This measurement is shown in Figure 4.3(a).

As this measurement can be done in either flow direction, an observer that can observe both directions of a flow can extract two RTT measurements per RTT. However, as is shown in Figure 4.3(b), these samples are *not* independent. This is because the spin edges used to perform up- and downstream measurement travel in the same packets.

Up- and downstream delay measurement

When both directions of a flow can be observed, an observer can also measure the delay up- and downstream in the network separately. These delays are referred to as half-RTTs. As shown in Figure 4.4, the observer measures them by timing the duration between spin bit transitions in different flow directions, rather than considering both flow directions separately. These measurements are fully independent. This can be useful for network operators to determine whether a problem exists in or outside of their own network.

Cooperative measurements

When multiple observers are cooperating, the delay of specific parts of the network can be measured. For example, the delay between two observers can be measured by subtracting two half-RTT measurements from those observers, as shown in Figure 4.5. Note that the resulting delay is the sum of the up- and downstream delays between those observers. When the clocks of the cooperating observers are accurately synchronized, it is also possible to measure the up- or downstream path delay separately, by recording the absolute times of the spin bit transitions and subtracting these. While this does not necessarily require the spin bit — the arrival times of arbitrary packets could be stored —, it allows observers to easily agree on which packets to sample.

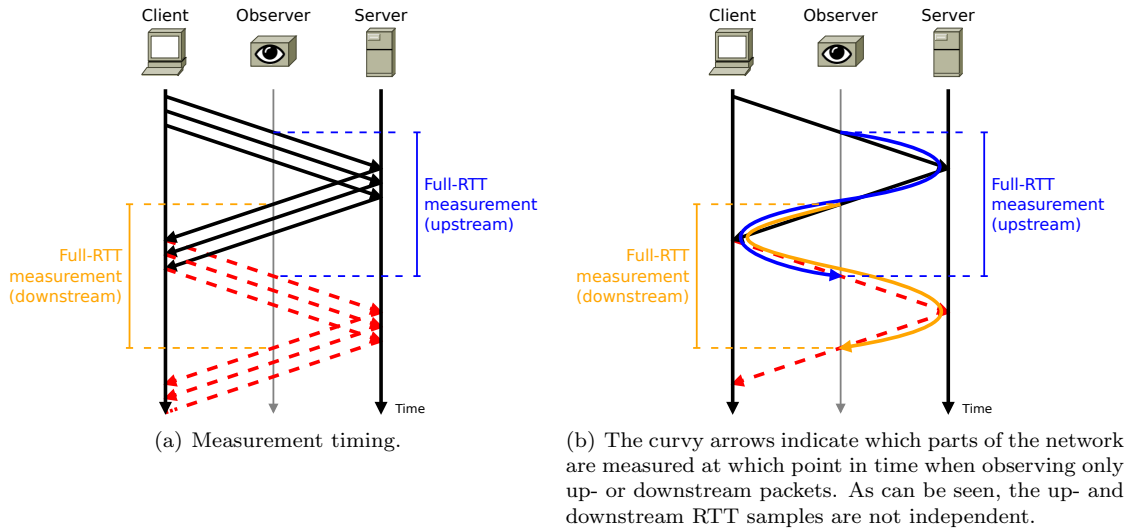


Figure 4.3: Schematic representation of a full-RTT measurement by an on path observer. Different line styles represent different spin values carried by the packet.

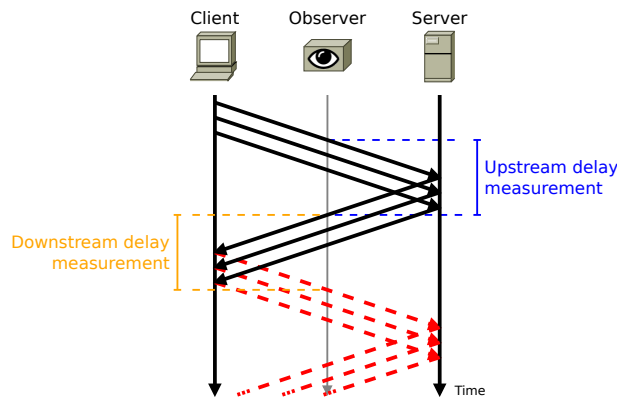


Figure 4.4: Schematic representation of up- and downstream half-RTT measurements by an on path observer. Different line styles represent different spin values carried by the packet.

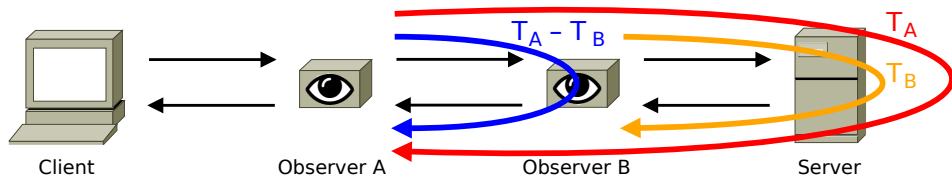


Figure 4.5: Schematic representation of a cooperative measurement of the delay of a section of the network.

4.3.2 The effects of network impairments and sparse traffic patterns

When the network exhibits impairments, or when the observed flows have (very) low data rates, the fidelity of the spin signal can be reduced. This section will discuss the influence these network and traffic conditions have from a theoretical perspective, while Chapter 6 shows the effects on an emulated network.

The effect of reordering

As shown in Figures 4.6(b) and 4.6(c), packet reordering can — but does not necessarily — influence the RTTs measured by observing the spin bit. When this is the case depends on whether or not the reordering created a new spin edge. If this is not the case, the observed spin signal does not change, and therefore neither does the observed RTT. However, if a new spin edge is introduced, two new — generally short — RTT estimates are added, and the original RTT estimates are shortened as well. In summary, instead of making two correct RTT estimates, an observer now makes four incorrect estimates.

It is noteworthy that an observer with access to the packet serial numbers, could trivially detect the reordering event and reject the incorrect RTT samples. However, when the packet number is encrypted, this is possible.

The effect of loss

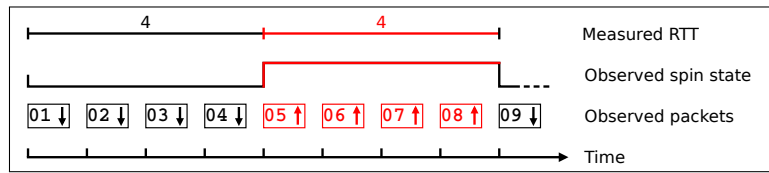
Similarly to reordering, loss can — but does not necessarily — influence an observer's RTT measurements. This is illustrated in Figures 4.6(d) and 4.6(e). Losing a packet that does not carry a spin edge does not influence the spin signal, and therefore does not influence the RTT estimates. However, when a packet carrying a spin edge is lost, two RTT estimates are offsetted by one interpacket time, each in a different direction. The severity of these errors is highly dependent on the traffic pattern: when the flow carries a lot of data, packets are likely to be spaced closely together in time, making the interpacket time — and thus the error — small. If, however, a flow carries sparse traffic, the interpacket times are likely to be larger, thus leading to larger errors. This is especially true when not one, but a burst of packets is lost. Because of the prevalence of loss based congestion controllers, flows on lossy networks typically have small congestion window sizes, exacerbating this effect.

Contrary to packet reordering, loss cannot be reliably detected by observing packet numbers, as QUIC endpoints “may intentionally skip packet numbers to introduce entropy into the connection” [8].

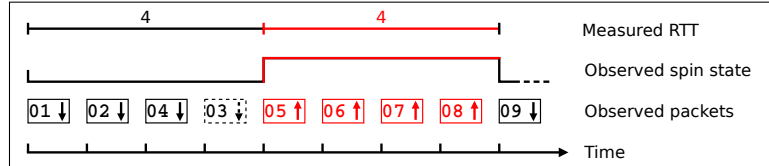
The effect of sparse traffic patterns

When an endpoint is unable to immediately send out a new packet after it receives a packet carrying a spin edge, an error is introduced into the observer's RTT observations. This situation typically occurs when the endpoint's congestion window does not allow for it to send data (e.g. because loss recovery is ongoing), or when the endpoint has no data to send. The later is illustrated in Figure 4.7. Concretely, the time the endpoint has to wait between receiving the spin edge from the remote endpoint, and generating one himself is added to the observer's RTT measurements. The spin edge generated by the server is then called a *delayed* edge. Such an edge is shown in Figure 5.1(b).

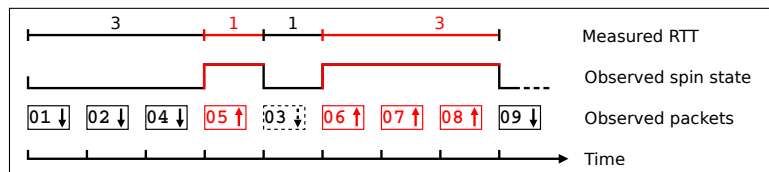
This situation can be especially problematic, because the resulting measurement may have no correlation with the flows RTT at all, and the observer is unable to detect when this is the case.



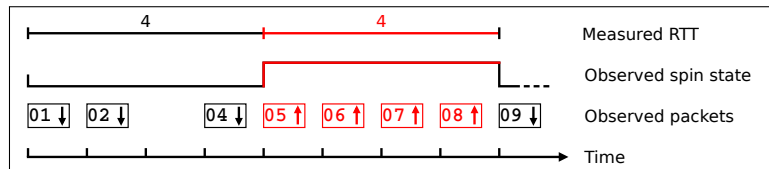
(a) Under ideal network conditions the observer correctly measures the RTT.



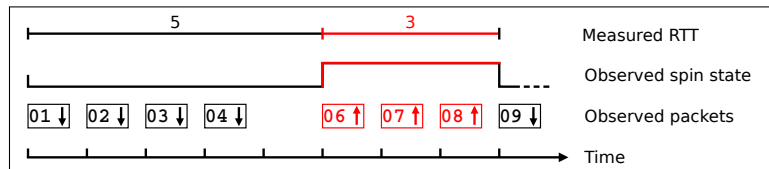
(b) Some cases of reordering do not affect the measured RTT values.



(c) Reordering packets over a spin edge leads to multiple incorrect RTT readings.



(d) Some cases of loss do not affect the measured RTT values.



(e) Losing a packet that carries a spin edge leads to two incorrect RTT readings.

Figure 4.6: Schematic representation of how an on path observer can measure the RTT of a network flow by only observing the spin value, and the effect of loss and reordering on these measurements. The observer monitors only one direction of a flow with a RTT of 4 time ticks. Reordered packets are drawn with dotted edges.

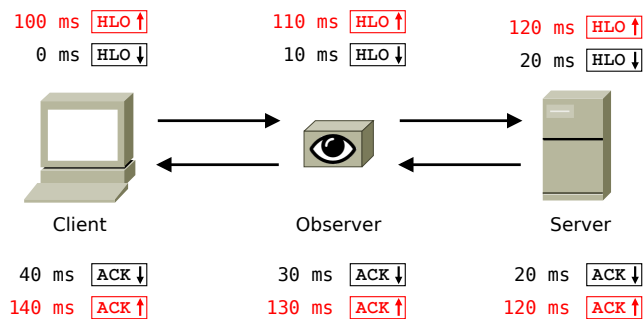


Figure 4.7: Schematic representation of a heartbeat traffic pattern and its effect on the spin bit. Every 100 ms the client sends a Hello message (HLO) to the server, who acknowledges receipt (ACK). Every link has a 10 ms delay, resulting in a 40 ms RTT. However, the spin bit toggles every 100 ms.

Chapter 5

Observer flavours and spin enhancements

Up to this point, mainly one type of spin bit observer has been considered: a basic observer that directly measures a flow’s RTT from the spin bit waveform. However, there is no need for observers to restrict themselves to this. In this section, a number of different observers are proposed, each coming with their advantages and disadvantages. Furthermore, a number of enhancements to the spin bit are introduced. Later in this work, in Chapter 6, these observers are implemented and their performance is analysed with the help of an emulated network.

These implementations are used in this section to compare the lines of code and required state for each observer. It should be noted that the given numbers are *upper bounds* for each observer’s complexity, and that further optimizations might still be possible. However, by comparing real implementations written in the same code style, a fair comparison between the observers is achieved. Lines of code are counted for a bidirectional analyser without header parsing, 5-tuple classification or state-variable declaration. State is also considered for an observer analysing both flow directions, again excluding storing the 5-tuple. The impact the spin signal has on the endpoints is not considered, as it has trivial complexity compared to the remainder of the QUIC protocol.

Furthermore this section also theoretically compares how well analysers can deal with the effects of network impairments or sparse traffic. The results are summarized in Table 5.1.

5.1 Spin bit observers

5.1.1 The basic observer

The basic observer classifies every packet in to a flow, extracts the spin bit from the QUIC header and measures the duration between transitions in each direction of the flow (see Figure 4.6(a)). This way, the observer can get two RTT samples (one in each direction) per flow per RTT. Note however that these two samples are not independent, as can be seen in Figure 4.3(b).

Implementing this observer can be done in only 19 lines of code, and requires only direct comparisons between the stored state and the packet data. With a per flow memory footprint of 34 bytes, this observer also requires the least amount of state. However, the basic observer has no tolerance to reordering: a single out of order packet can cause the observer to register two additional spin bit transitions, resulting in four false RTT readings, as shown in Figure 4.6(c). The basic observer can also not filter out the effects of loss, nor detect delayed edges.

Table 5.1: A comparison of observer types.

Observer	Complexity		Tolerance to		
	Lines of code	State/flow [B]	Reordering	Loss	Delayed edges
One bit signals					
Basic	19	34	None	None	None
PN*	21	42	$2^8, 2^{16}$ or 2^{32} packets	None	None
Static heur.	25	34	1 ms	None	None
Dyn. heur.	48	182	$\frac{1}{10} \cdot \text{RTT}$	None	None
Two bit signals					
Two bit spin	19	34	3 RTTs	None	None
Valid edge	26	36	Full	Some	Full [†]
Three bit signals					
VEC	25	34	Full	Full	Full

* Only a single bit signal when the QUIC packet number is sent unencrypted.

† Only when both directions of a flow are observed, otherwise only some tolerance.

5.1.2 The packet number observer

The packet number observer (or PN observer) is similar to the basic observer, but implements reordering detection based on the packet number. Similarly to the endpoints, this observer will ignore all packets with a packet number smaller than that of the largest already observed packet (per flow direction). Depending on the exact header format used, packet numbers are 8, 16 or 32 bits long. Thus, reordering of up to $2^8, 2^{16}$ or 2^{32} packets can be tolerated.

Implementation wise this observer is still simple, clocking in at 21 lines of code. However, the need to read a second field from the QUIC header makes this approach somewhat less elegant. Furthermore, in comparison to the basic observer, an additional 32-bit packet number (worst case) needs to be stored for each flow direction. This results in a slightly higher memory footprint of 42 bytes per bidirectional flow. However, in exchange for these additional bits, reordering of up to 2^{32} packets can be tolerated. Unfortunately, no loss or delayed edges can be tolerated.

5.1.3 The heuristic observers

In order to reject erroneous spin bit transitions created by out of order packets, it is also possible to use heuristics. Because exploring the entire space of heuristic observers is out of scope for this work, only two simple heuristics are considered. This should suffice to give some basic insight in to the applicability of heuristics in general.

Considered are:

- A *static* rejection heuristic, that rejects spin bit transitions if they would result in a RTT measurement lower than a preset threshold. In this work, a threshold value of 1 ms is used.
- A *dynamic* rejection heuristic, that rejects spin bit transitions if they would result in a RTT measurement lower than a preset fraction of the moving minimum of the RTT samples. In this work, the rejection threshold is placed at one tenth of the minimum of the last ten samples. In order to be robust against sudden, large, RTT decreases, a sample is also accepted if it is the fifth sample in a row being rejected.

Of these two methods, the static heuristic observer is the simplest to implement, and at 25 lines of code has a code footprint that is only slightly higher than the basic observer. No additional state is stored, so the memory footprint remains at 34 bytes, the same as for the basic observer.

The dynamic heuristic observer is significantly more complex. At 48 lines of code, it has more than double the code complexity of the basic observer. Furthermore, because it needs various counters and a circular buffer to store historic RTTs, it requires a whopping 182 bytes of memory per flow. This makes the dynamic heuristic observer the most complex observer considered in this work.

The reordering tolerance of the static heuristic is 1 ms, while for the dynamic heuristic it is one tenth of the RTT. Neither heuristic provides tolerance to loss or delayed edges.

5.2 Spin bit enhancements

Thus far, the analysis has assumed that the spin signal consists of only one bit. However, adding additional bits to a spin signal can improve it significantly. This section discusses possible bits that can be used to augment the spin signal, together with the observers used to analyse them.

5.2.1 A two bit spin value

In order to detect packet reordering, it is possible to use a two bit spin signal. Instead of inverting the spin signal, the client would set the outgoing spin signal as follows:

$$\text{outgoing spin} = (\text{incoming spin} + 1) \pmod{4}$$

In other words, instead of counting up in \mathbb{Z}_2 , this signal counts up in \mathbb{Z}_4 .

This does not significantly increase the observers complexity, as still only 19 lines of code and 34 bytes of state per flow are required.

However, by ignoring all spin values that do not equal the next expected spin, observers can effectively filter out packets that are reordered up to 3 RTTs. Unfortunately, still no tolerance to loss or delayed edges is achieved.

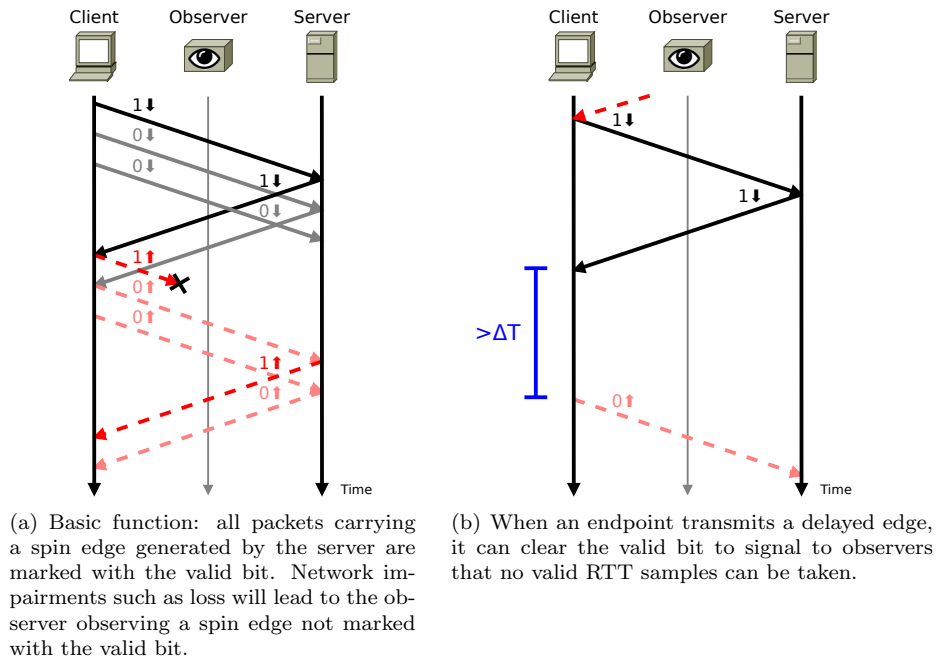


Figure 5.1: Schematic overview of the functionality of the valid bit. The valid bit (as digit) and spin (as arrow) are indicated for each packet. Different line styles represent different spin bits.

5.2.2 The valid bit

Another way to augment the spin signal is to use an additional *valid* bit that is set by the endpoint as follows:

The valid bit of a packet is set to '1' only if that packet carries a transition in the spin signal *and* that packet was generated within ΔT time units after receiving the incoming packet that caused the transition in the outgoing spin signal.

In this work, ΔT is set to 1 ms. The functionality of the valid bit is illustrated in Figure 5.1.

Analysing the valid bit somewhat increases the complexity of the observer, making it require 26 lines of code, and 36 bytes of state per flow. However, this additional complexity brings tolerance to:

Reordering: an observer can check that each spin transition it sees is accompanied by a set *valid* bit. This will not be the case for additional spin edges generated by reordering.

Loss (partially): losing a packet carrying a valid bit *before* it passes the observer will lead to the observer not seeing a valid spin transition, causing it to not take an RTT sample. However, losing a packet carrying a valid bit *after* it passes the observer will not be detected, but can still influence the RTT measurement.

Delayed edges: an endpoint will explicitly signal delayed edges by setting the valid bit to '0'. This is only true when both directions of a flow are observed, otherwise delayed edges in the unobserved part of the flow are not detectable.

5.2.3 The Valid Edge Counter (VEC)

As mentioned above, the valid bit does not allow an observer to detect loss that occurs *after* a packet has passed the observer. However, this loss can still influence the timing of the spin bit. Therefore, it would be desirable for the observer to be able to detect such loss, and reject RTT samples as necessary.

This can be achieved by naturally expanding the valid bit to a two bit Valid Edge Counter, or VEC, that is set by the endpoints as follows:

- By default, the VEC is set to ‘00’.
- When a delayed edge in the spin signal is transmitted, the VEC is set to ‘01’.
- Whenever an edge in the spin signal is transmitted in time, the VEC is set to the value of the VEC that accompanied the last incoming spin bit transition plus one, holding at three.

Or, in pseudocode:

```

1: last.spin ← 0
2: last.vec ← 0
3: last.pn ← 0
4: last.time ← 0
5: last.new ← False
6:
7: procedure ON INCOMING PACKET(packet)
8:   if packet.pn > last.pn and packet.spin ≠ last.spin then
9:     last.spin ← packet.spin           ▷ Store spin, VEC, PN and time of transition
10:    last.vec ← packet.vec
11:    last.pn ← packet.pn
12:    last.time ← time.now()
13:    last.new ← True
14:   end if
15: end procedure
16:
17: procedure ON OUTGOING PACKET(packet)
18:   if endpoint is client then           ▷ Client flips spin, server echos back
19:     packet.spin ← not last.spin
20:   else
21:     packet.spin ← last.spin
22:   end if
23:
24:   if last.new then
25:     if time.now() - last.time < ΔT then           ▷ If edge and not delayed
26:       packet.vec ← min(last.vec + 1, 3)           ▷ Increase VEC
27:     else                                           ▷ If edge and delayed
28:       packet.vec ← 1                               ▷ set VEC to ‘01’
29:     end if
30:   else
31:     packet.vec ← 0                               ▷ In all other cases, VEC is ‘00’
32:   end if
33:
34:   last.new ← False
35: end procedure

```

The functionality of the VEC is illustrated in Figure 5.2. Implementing an observer for the VEC requires 25 lines of code, and has the same memory requirements as the basic observer: 34 bytes per flow.

Now, whenever a VEC observer sees a packet with a VEC value of ‘01’ or higher, it knows that this packet contains a spin edge generated by an endpoint, and that it can thus use this packet as the starting point of an RTT measurement. Similarly, when a packet with a VEC value of ‘10’ or higher is observed, the observer knows that this packet can be used to terminate a half-RTT measurement. Finally, a packet with a VEC value of ‘11’ can be used to terminate a full-RTT measurement. This is true because a VEC value of ‘11’ assures that a packet carries a valid spin transition, and that the previous two spin transitions have passed through the network without being lost, delayed or severely reordered, as that would have caused the VEC to fall back down to zero. A similar argument can be made for a VEC value of ‘10’. These results are summarized in Table 5.2.

Because of these assurances, the VEC informs the observer with full certainty about when it can take a valid RTT sample. Even when only a single direction of a flow can be observed. This means that an observer using the VEC can — in theory — perfectly filter out the effects of reordering, loss and delayed edges.

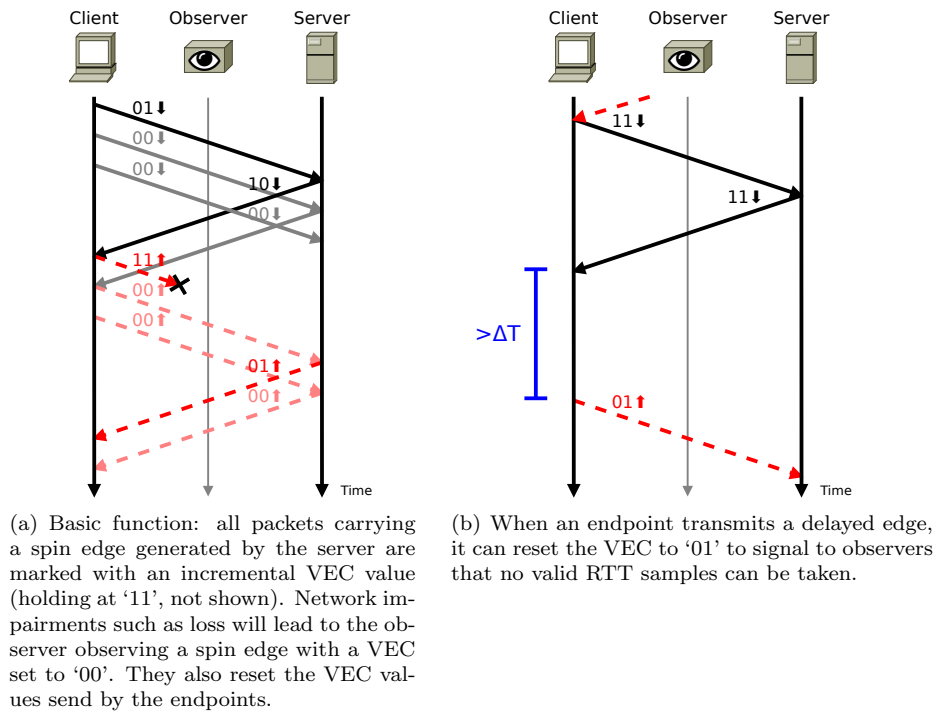


Figure 5.2: Schematic overview of the functionality of the VEC. The VEC bits (as digits) and spin (as arrow) are indicated for each packet. Different line styles represent different spin bits.

Table 5.2: Overview of what packets marked with each VEC codepoint can be used for.

Codepoint				Usecase
00	01	10	11	
	✓	✓	✓	Start of RTT measurement
		✓	✓	End of half-RTT measurement
			✓	End of full-RTT measurement

Chapter 6

Evaluation

In order to validate the quality and robustness of each of the observers introduced in Chapter 5, all of them have been implemented and extensively benchmarked on an emulated network.

6.1 Test setup

6.1.1 Network

The evaluation of the spin bit and various observers has been performed on a network emulated using Mininet and NetEm. The structure of this network is shown in Figure 6.1. It consists of a client, a server and five switches, one of which is the observer. Although on first sight it might seem that the network used is overly complicated, and that one switch — the observer — would suffice, this is actually not the case. This is due to two reasons.

Firstly, Linux traffic control is implemented in the output queues of the Linux kernel, *before* the packets are forwarded to the device driver [19]. However, Pcap intercepts packet at the device driver [20]. This means that the network sniffer will have a different view of the network link than the simulated host — the observer — on which it is running. In order to avoid this, the observer must run on a host that has traffic control disabled.

Secondly, NetEm is fairly limited in the type of network conditions it can simulate. It is for example not possible to specify the rule “All packets are delayed for 10ms, and 5% of packets must be held back an additional 1ms to create reordering” in one NetEm rule. As NetEm does not support nesting (i.e. having a packet processed by multiple NetEm instances sequentially)¹, two hosts, and thus two links, are used instead. *Static* links are used to simulate the network’s basic parameters. In this case this means delaying packets for 10ms. To add further imperfections to the network, *dynamic* links are introduced. In the example this link delays 5% of packets for 1ms to simulate reordering. In order to simulate dynamic network characteristics, the parameters of the dynamic links are changed on the fly, hence their name.

6.1.2 Endpoint software

The endpoints run Pinq (Piet’s Minq), a custom version of Minq (see Section 2.2). Pinq mainly differs from vanilla Minq in that it adds a measurement byte to the QUIC header. Within this byte, individual bits are allocated to carry the spin, edge valid and VEC signals. Note that the use

¹see <https://lists.linux-foundation.org/pipermail/netem/2006-0ctober/001012.html>

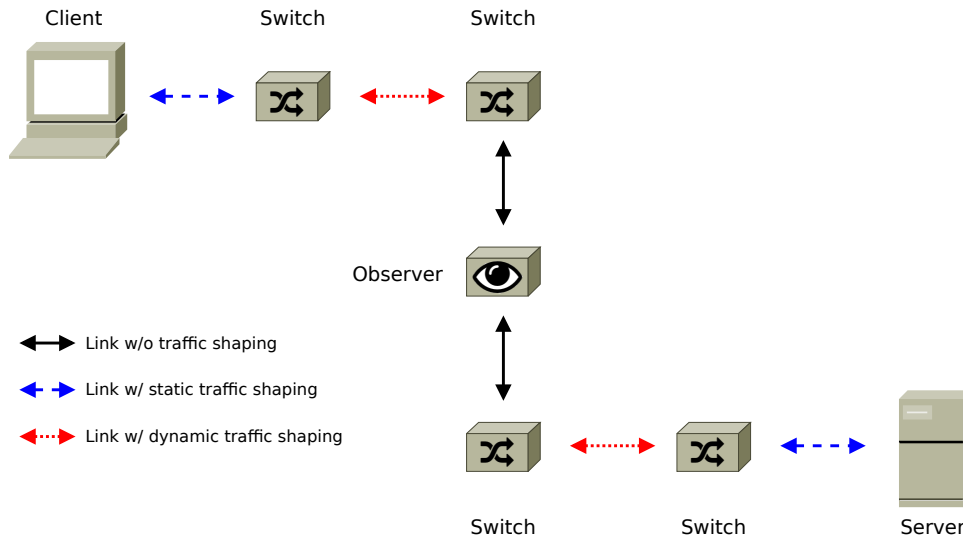


Figure 6.1: Schematic representation of the emulated network used in all experiments.

of a dedicated measurement byte is not necessary, and that these bits could be placed anywhere in the QUIC header.

Because there are concerns within the IETF that the spin bit might leak privileged information from the QUIC state machine, the measurement field logic is implemented in almost complete isolation from the remainder of the QUIC code. The only interaction points between the measurement code and other code are:

- When a new connection is initiated, `newMeasurementData(role)` is called, passing only whether the endpoint is the client or server.
- For every received packet `incomingMeasurementTasks(hdr)` is called, passing only the public QUIC header.
- For every transmitted packet, `measurement(hdrData.encode())` is called. This function takes no arguments, and returns the measurement byte as an unsigned integer.

Thus, besides the timing of these calls, all information supplied to the measurement code is also available to any observer on the connection's path. In fact, the only reason that the measurement code cannot be implemented as a shim layer between the QUIC and UDP code is because the measurement field should be authenticated by the QUIC code. Adding an interface to QUIC to add data to its authenticated header would lift this restriction. An example of such an interface is proposed by PLUS [17].

In order to be able to run the desired experiments, a number of other changes had to be made to Minq. These included debugging, adding loss detection and congestion control, profiling and more. Almost all of these changes have been upstreamed back to Minq.

6.1.3 Observer

The observer uses Tcpcdump to capture the network traffic and write it to disk. These traffic traces are later replayed to a VPP instance using Tcpreplay. A custom module implementing all the observers introduced in Chapter 5 is loaded in to the VPP instance, which reports the observed RTTs in real time.

The separation between data generation (using Mininet) and data analysis (using VPP) is made

to ensure that experiments are reproducible, and that the same packet traces can be replayed to different versions of the observer implementations.

6.2 Results

Using the setup described earlier in this chapter, a number of experiments were performed to validate and evaluate the latency spin bit.

In each of these experiments, the client and server both calculate the RTT of the connection using TCP's exponentially decaying smoothing algorithm [21]. The resulting client estimate is used as ground truth for application experienced RTT.

When the error of an RTT measurement is considered, it is compared to the linearly interpolated client estimate. A positive error means that the observer overestimates the RTT. Note however that the term *error* should be interpreted carefully, as RTT measurements taken at different points in the network might result in different — correct — RTT values. This is illustrated in Figure 6.4 and further explained in Section 6.2.3.

Unless otherwise noted, the network setup shown in Figure 6.1 is used. The client is configured to continuously upload data to the server. The server does not transmit data. The links with dynamic traffic shaping are set to have a delay of 10 ms each, resulting in a base RTT of 40 ms. When evaluating network impairments, the links with dynamic traffic shaping are used to add these impairments to the network. Note that every packet passes an impaired link *twice*. Once before, and once after being observed. All links are bandwidth limited to 100 Mbps. All ECDFs (Empirical Cumulative Distribution Functions) are based on 60 seconds of data.

6.2.1 Basic functionality

To validate the basic functionality of the spin bit, a network with a base delay of 40 ms is emulated. Then, around the 80 s mark of the emulation, the network’s delay is doubled to 80 ms. Figure 6.2 shows the evolution of the server’s, client’s and observer’s estimates of the RTT over time.

Two interesting observations can be made from this graph. Firstly, as is to be expected, the spin bit estimates closely follow the *application experienced* RTT, rather than the link RTT.

Secondly, in this scenario the spin bit observer reports finer grained and more accurate RTT information than the server. This is because when data is flowing overwhelmingly in one flow direction, the receiving endpoint will not receive frequent ACKs from the sending endpoint. Because QUIC endpoints need to receive acknowledgements to estimate the connection’s RTT, it can only do so occasionally². Combining this with the exponential smoothing algorithm used by QUIC, results in the server’s RTT significantly lagging the link. This behavior is not unique to TCP, as the RFC (Request For Comments) on TCP RTT estimation also specifies that RTT samples should only be taken when new data is being acknowledged [22].

What this means is that the spin bit cannot only be useful for on path observers, but — in some traffic scenarios — also endpoints can use the spin bit to obtain better RTT estimates. For example, in order to dimension the buffer of a video stream.

6.2.2 The influence of bursty traffic patterns

As explained in Section 4.3.2, sparse traffic on a link can cause the spin bit to toggle at a frequency different from the flow’s RTT. In Sections 5.2.2 and 5.2.3 it is proposed that marking delayed edges as invalid makes it possible for an observer to detect and reject faulty RTT samples.

In order to validate this, a client is configured to upload data to a server while alternating two traffic patters. That is, it continuously sends single packet Hello’s to the server at 100 Hz (as in Figure 4.7), combined with bursts of 1 MB of data every 5 seconds.

An excerpt from the resulting RTT trace is shown in Figure 6.3. It can be seen that while the basic analyser incorrectly reports an RTT of 100 ms between the data bursts, the valid edge observer is able to filter out these samples completely. The same is true for the VEC observer (not shown in figure).

6.2.3 The influence of the packet scheduler

As can be seen in Figure 6.2, the spin bit RTT estimates closely follow the application experienced RTT. However, there is still a noticeable difference between the two. The reason for the difference between the client and observer estimates is threefold. Firstly, the client can take much more frequent samples, that are then exponentially smoothed. The observers can only take two samples per RTT, which it does not smooth. Secondly, the client and the observer measure the delay of different parts of the network at different times. As shown in Figure 6.4, when the network (buffer) delays are continuously varying, this results in different RTT readings. Thirdly, the endpoints do not measure the local delay between receiving and sending a packet, while the observer does. This is also shown in Figure 6.4.

All three of these causes are influenced by the packet scheduler: when there is more traffic in the flow, the endpoints will get more frequent RTT updates, and the depth of the network buffers — which is a major factor in the application RTT — will vary more. Furthermore, the packet

²In this scenario, whenever the server issues flow control credit, which is acknowledged by the client.

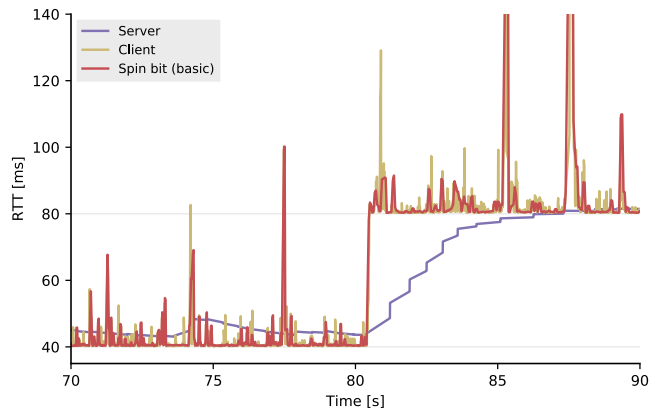


Figure 6.2: RTT estimates from the server, client and basic observer over time. Endpoints use window based congestion control. The network’s RTT initially is 40 ms and later changes to 80 ms.

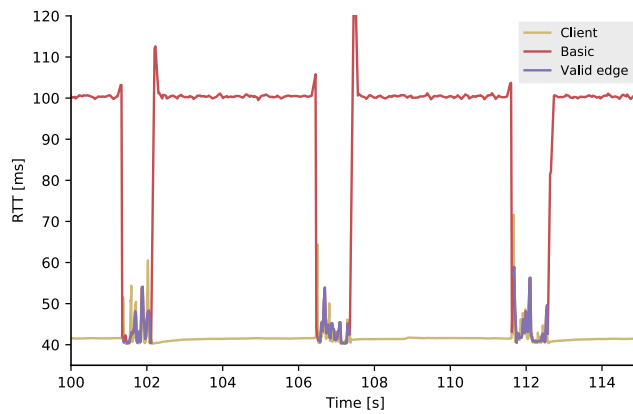


Figure 6.3: RTT estimates from the client, basic observer and valid edge observer over time when alternating the heartbeat traffic pattern with bursty data transmissions. The link RTT is 40 ms, every 100 ms a heartbeat message is sent.

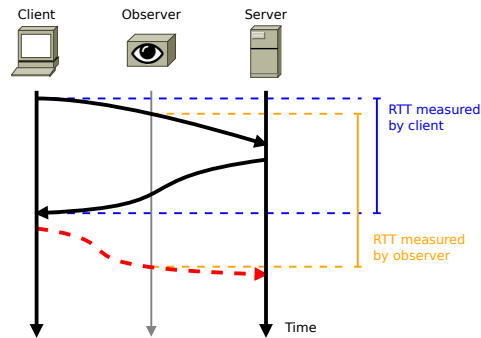


Figure 6.4: Schematic representation illustrating how measuring the RTT at different point in a dynamic network will yield different results.

scheduler determines when a packet gets sent, and therefore also the delay between receiving and transmitting a packet.

Figure 6.5 shows how using a different packet scheduling technique influences the error between the client and observer RTT estimates. Three packet schedulers are considered:

- A classic, TCP style, window based congestion controller (CC).
- A window based scheduler that uses a static window size of 50 packets.
- A fixed rate packet pacer, pacing at 1.5 MBps.

The static window size and pacer rate are chosen to result in similar data rates. Figure 6.6 also shows the influence of varying the window size of a static window scheduler.

The curves of the window based schedulers (static and dynamic) are largely similar and relatively symmetric. As stated above, when a flow has a higher data rate, the depth of the network buffers, and therefore the RTT, will vary more. Figure 6.6 shows this effect in action: when increasing the window size (`cwnd`) of the QUIC endpoints, the error between the client and observer estimates increases too.

However, using a fixed rate pacer results in a significantly different error distribution. This distribution is different in two ways. Firstly, the ECDF is steeper. This is attributed to the fact that the pacing scheduler attempts to spread out packets evenly in time, while the window based techniques typically transmit short bursts of data. As a consequence of this, using the pacing scheduler will result in constant length buffer queues (and therefore RTTs), while the bursty sending behaviour of the window based schedulers will result in varying buffer queue lengths, and therefore varying RTTs.

Secondly, the ECDF of the pacing scheduler is asymmetric: the median error is positive. More concretely, the ECDF shows a long and sharp increase between the 0 and 1 ms mark. This effect has two origins: the Linux process scheduler and the packet pacer itself.

The Linux process scheduler is responsible for scheduling the Minc threads. When a packet carrying a spin edge arrives at an endpoint when it has no sending credit, the Minc thread will go to sleep. Now the endpoint has to wait until Minc is woken up by the kernel before it can transmit an outgoing spin edge. Unfortunately, the kernel will only check which threads need to be woken once an internal timer (the so called “jiffy” timer) fires. For modern Linux distributions this defaults to once every millisecond. This delay is added to the spin bit measurement, but not to the client RTT estimate.

The packet scheduler is set to pace at 1.5 MBps. This corresponds to sending roughly one packet per millisecond. When a packet carrying a spin edge arrives at an endpoint when it has no sending credit, it can take up to 1 ms before enough credit to send the next packet has accumulated. Just like the delay from the Linux scheduler, this delay is added to the spin bit measurement, but not to the client RTT estimate.

6.2.4 The influence of reordering

As stated in Section 4.3.2, packet reordering will lead to the basic observer making too many, and too short RTT measurements. Most of the observers described in Chapter 5 are designed to have at least some tolerance to this effect.

To verify this, reordering is introduced in to the emulated network. Each dynamic link holds back most packets back by 1 ms, but forwards a set fraction of packets immediately. Each packet passes two dynamically shaped links (one before, and one after being observed), and might therefore be reordered up to two times. In order to suppress the effects of the congestion controller (i.e. variations in the window size), the endpoints use a static window size of 60 packets.

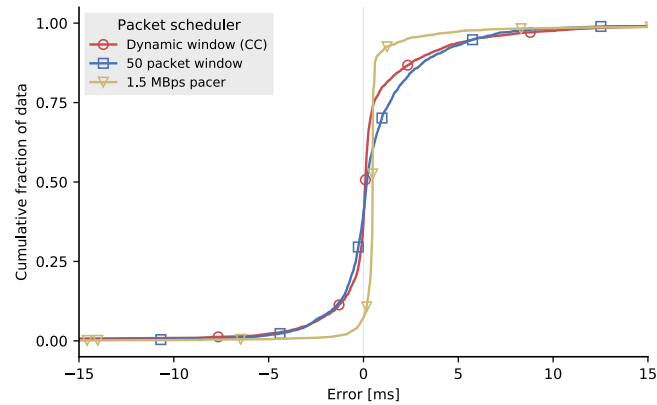


Figure 6.5: Effect of the packet scheduler on the deviations in spin bit based RTT measurements. The observed RTT is compared to the client estimate, which is considered to be ground truth. Positive errors correspond to overestimated RTTs.

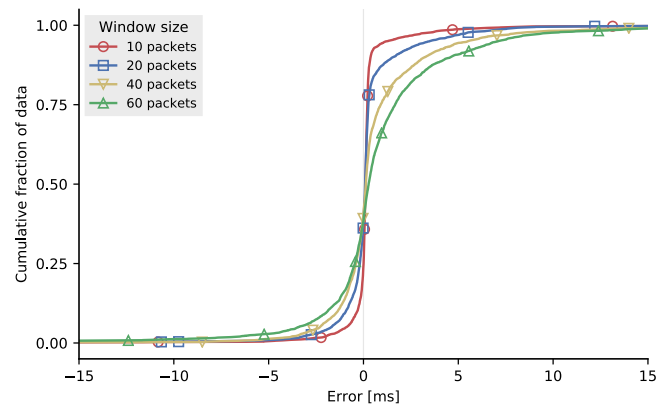
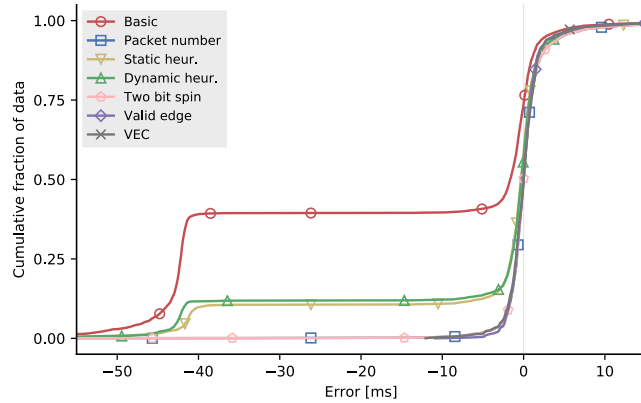
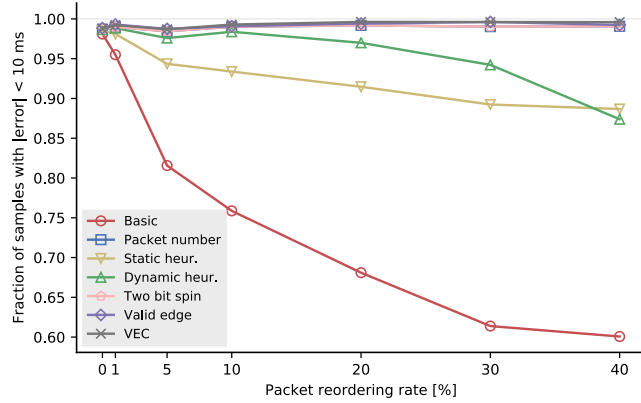


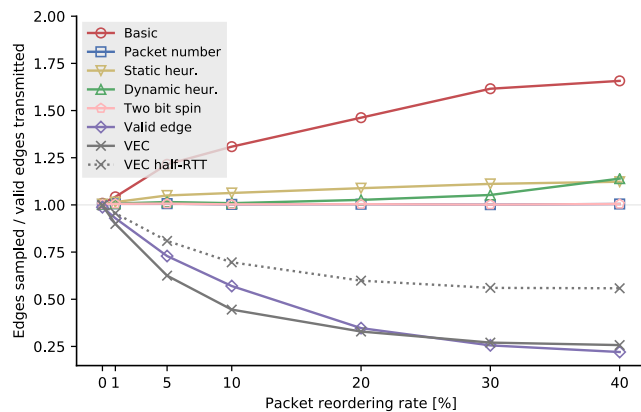
Figure 6.6: Effect of the congestion window size on the deviations in spin bit based RTT measurements. The observed RTT is compared to the client estimate, which is considered to be ground truth. Positive errors correspond to overestimated RTTs.



(a) ECDF showing observer error at a 40% reordering rate.



(b) Observer error for various reordering rates.



(c) Number of samples taken by each observer (normalized to the amount of valid edges transmitted by the endpoints) for various reordering rates. A plot with absolute numbers is shown in Figure B.1.

Figure 6.7: Graphs showing the effects of reordering on the results of various observers. Reordering rates express the percentage of packets that is reordered by 1 ms. Positive errors correspond to overestimated RTTs.

The results for different grades of reordering are shown in Figure 6.7 as follows:

Figure 6.7(a) shows an ECDF of the error in the observed RTTs (compared to the client RTT estimates) for the various observers at a 40% reordering rate. This rate is chosen as it most clearly demonstrates the effects reordering has.

Figure 6.7(b) shows how the error in the observed RTTs evolves over various reordering rates.

Figure 6.7(c) shows how many RTT samples are taken by each observer for various levels of reordering. This number is normalized to the number of valid spin edges (i.e. packets with the valid edge bit set) that are generated by both the endpoints. A version of this plot with absolute numbers can be found in Appendix B.

In the ECDF it can be seen that the vast majority of incorrect samples have an error of about -40 ms. This corresponds to one RTT, and therefore indicates that a number of near zero RTTs are observed. This is exactly what is expected from the theoretical analysis in Section 4.3.2, because when a packet is reordered over the spin edge (but remains close to that edge), three closely spaced together transitions in the spin signal are generated. These lead to the observation of two very short, incorrect RTTs (see Figure 4.6(c)). Figure 6.7(c) further confirms this by showing that the basic observer samples many invalid edges.

Figure 6.7(b) shows that the dynamic heuristic outperforms the static heuristic at the lower reordering rates, but that its accuracy starts declining rapidly for the more severe impairments. This is because once the dynamic heuristic accepts an incorrect sample above its rejection threshold (e.g. at $0.2 \cdot RTT$) the rejection threshold is lowered further (in this example to $0.02 \cdot RTT$), which is detrimental for its error rate.

The other methods (PN, two bit spin, valid edge and VEC) all almost perfectly filter out the reordering effects. However, the difference between them is clearly visible in Figure 6.7(c): The PN and two bit spin observers keep sampling once per RTT (possibly accepting small errors), while the valid edge and VEC observers reject up to 80% of the valid edges.

The valid edge observer rejects this many samples because it must see three valid edges in a row to perform a measurement:

- At the start of the measurement interval.
- On the return packet, to ensure that the returning spin edge was transmitted timely.
- On the packet it ends the RTT measurement on, again to ensure that the spin edge was not delayed.

Interleaved invalid edges cannot be ignored — as this would lead to false readings under loss —, forcing the valid edge observer to abort its current measurement when it detects one.

The VEC suffers a similar issue, as reordering can lead to the endpoints observing a spin edge with a VEC set to '00'. This causes them to reset the VEC to '01', again preventing the observer from taking an RTT sample.

It is worth nothing that both the valid edge and VEC observers would be able to take more samples if half-RTTs are measured. In such case the valid edge observer requires only two consecutive edges marked with the valid bit, and the VEC observer can sample the RTT whenever it sees a packet with VEC '10' or '11'. The number of half-RTT samples that could be taken by a VEC observer is shown in Figure 6.8(c)³.

³Number computed by counting the number of '10' and '11' VEC codepoints seen by the observer. The half-RTT observer itself has not been implemented.

6.2.5 The influence of jitter

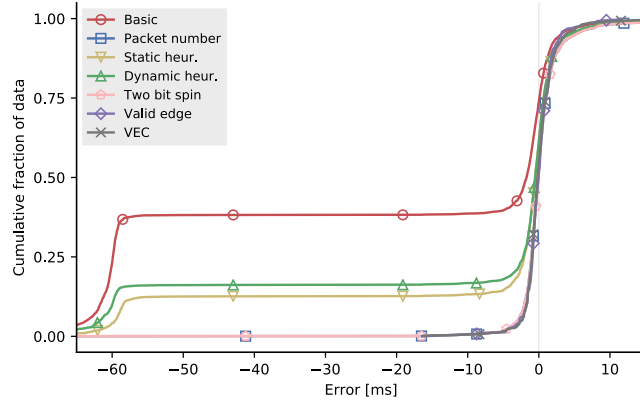
Next packet jitter is considered. This represents an extreme case of reordering: the dynamic links now hold back every packet by an average of 5 ms, plus or minus a jitter constant. The packet delays are chosen from a bounded normal distribution spanning this interval.

Figure 6.8 shows the effect various jitter rates have on the spin observers. The ECDF for 1 ms of jitter is shown, because at higher jitter rates the number of RTT samples using the valid edge and VEC observers drops so quickly to draw clear ECDFs. As the applied jitter is an extreme case of reordering, the effects seen on the observers are similar to — but more severe than — the effects seen in Figure 6.7.

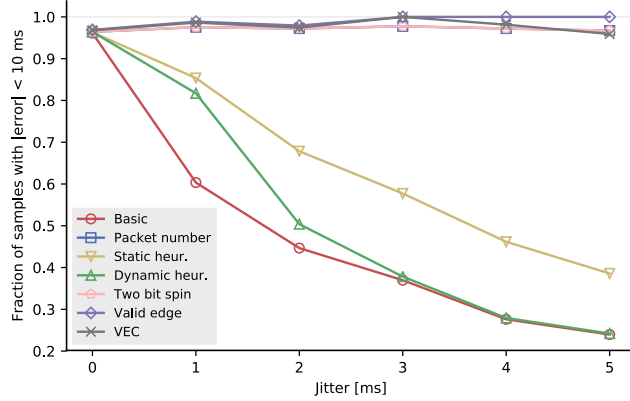
It can be seen that — contrary to the effects under milder forms of reordering — at all the jitter rates observed, the dynamic heuristic performs worse than the static heuristic. Moreover, for jitter above 3 ms, the dynamic heuristic observer starts behaving almost identical to the basic observer. That is, under these circumstances the rejection threshold of the dynamic heuristic becomes too small to have any meaningful effect.

The number of samples taken by the valid edge and VEC observers decreases rapidly when the jitter rate increases, down 0.4% and 2%, respectively. However, the samples that *are* taken, are relatively accurate, confirming that these methods provide the guaranty that when a sample is taken, it is accurate.

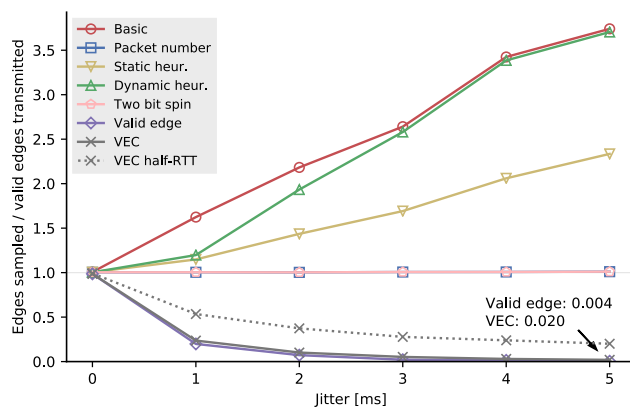
Interestingly, the packet number and two bit spin observers keep performing well under any amount of jitter. Both in terms of the error in the RTT samples, as in terms of the number of samples taken.



(a) ECDF showing observer error at 1 ms jitter.



(b) Observer error for various jitter rates.



(c) Number of samples taken by each observer (normalized to the amount of valid edges transmitted by the endpoints) for various jitter rates. A plot with absolute numbers is shown in Figure B.2.

Figure 6.8: Graphs showing the effects of jitter on the results of various observers. Jitter rates express the amount of jitter on a 5 ms delay link. Positive errors correspond to overestimated RTTs.

6.2.6 The influence of random loss

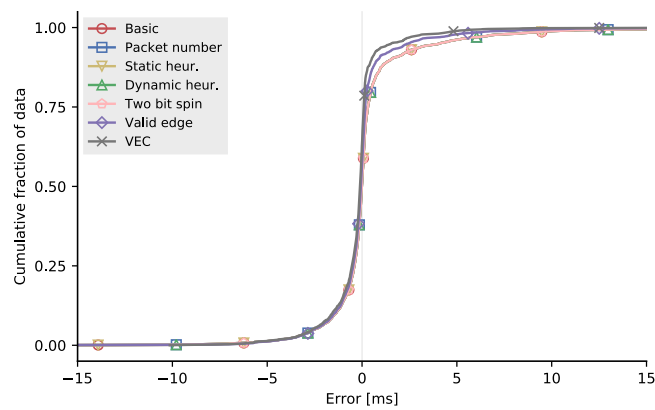
To evaluate the effect loss has on the spin bit, two network conditions are considered: random and burst loss.

Random loss is evaluated by configuring the dynamic links to drop a fixed percentage of randomly selected packets. As explained in Section 4.3.2, lossy links lead to larger errors for flows with sparser traffic. Therefore, the endpoint packet schedulers are configured to use a relatively small, fixed window of 20 packets. This is a realistic scenario because most congestion controllers would react to loss by reducing the window size. Figure 6.9 shows the emulation results.

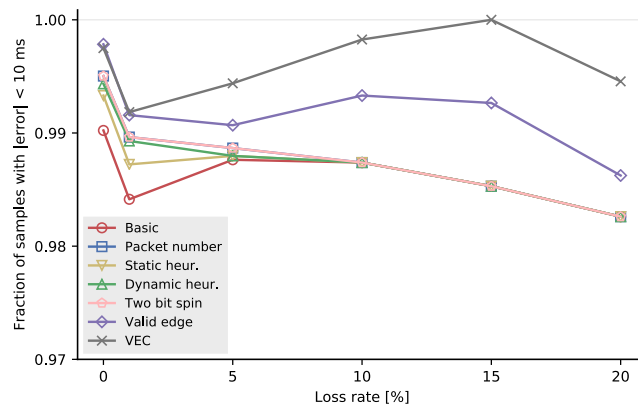
A first observation to be made, is that the effect of random loss on the accuracy of the RTT estimates is fairly limited. Even under extreme (20%) random loss conditions. This can be explained as follows: as illustrated in Figure 4.6(e), when a packet carrying a spin edge is lost, no spin edge is detected until the next packet with the new spin is observed. The error in the RTT reading will thus be dependent on the number of consecutive packets that are lost. That is, the length of the loss burst. When packets are lost with a probability r , the length of the loss bursts follows a geometrical distribution with parameter $1 - r$, which has an expected value of $(1 - r)^{-1}$.

Thus, even at a 20% loss rate ($r = 0.2$) the expected number of consecutive lost packets is only $(1 - r)^{-1} = 0.8^{-1} = 1.25$. Furthermore, because a (self-clocking) window based congestion controller is used, the packets are sent out in bursts, leading to small inter packet times, and thus thus leading to small errors.

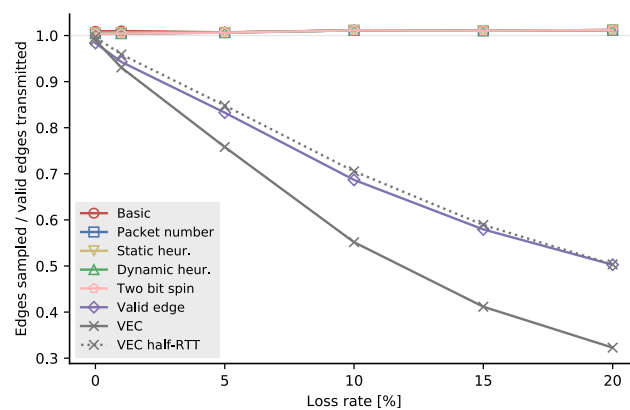
It can also be seen that the VEC observer produces more accurate RTT readings than the valid edge observer. This is because the later cannot detect when a spin edge gets lost after it has been observed, while the former can.



(a) ECDF showing observer error at 20% random loss.



(b) Observer error for various random loss rates.



(c) Number of samples taken by each observer (normalized to the amount of valid edges transmitted by the endpoints) for various random loss rates. A plot with absolute numbers is shown in Figure B.3.

Figure 6.9: Graphs showing the effects of random loss on the results of various observers. Positive errors correspond to overestimated RTTs.

6.2.7 The influence of burst loss

As discussed above, the expected error induced by a loss event is dependent on the amount of consecutive packets that are lost. Therefore, the effect of bursty loss is further evaluated in this section.

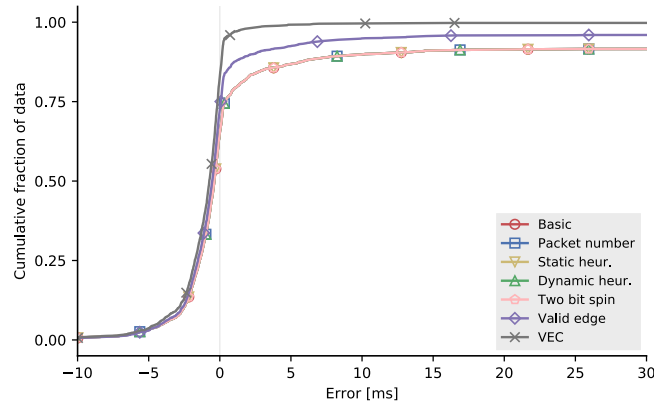
The links with dynamic traffic shaping are now configured to drop packets following the simple Gilbert loss model, as introduced in Section 2.5. The lengths of the good reception periods are set to follow a $Geo(0.01)$ distribution, resulting in an expected length of 100 packets. The lengths of the loss bursts also follow a geometrical distribution, the parameter of which is varied over the experiments. The resulting expected burst lengths vary from 0 to 20 packets. The emulations are carried out twice. Once the endpoints are set to have a fixed window size of 20 packets (the same value as used for the analysis of random loss), and once they are set to send at a fixed, paced, data rate of 0.6 MBps. This pacing rate is chosen because it results in a similar data rate as the window based packet scheduler. Pacing at 0.6 MBps results in sending one packet approximately every 2 ms.

Figures 6.10 and 6.11 show the results for the fixed window size and fixed pacing rate, respectively. Besides the standard differences expected between a window based and pacing packet scheduler (see Section 6.2.3), the clearest difference is that the ECDF of the run with the static window size has a much longer, flatter, tail. This is because of a fundamental difference between the static window and fixed rate packet schedulers: when a full flight of data is lost, the window based scheduler has to wait until a RTO (Retransmission Timeout) occurs before it can continue sending, while the fixed rate pacer continues sending without interruption. The time the window based scheduler waits for the RTO event is added to the observers RTT measurements. As the duration of an RTO is a multiple of the network's RTT, this results in the flat tail seen in Figure 6.10(a).

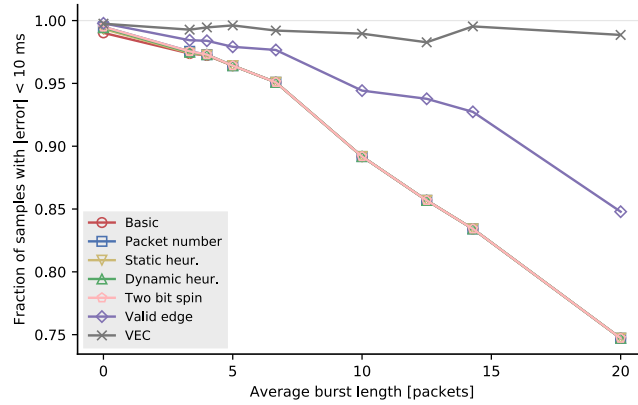
On the other hand, the ECDF of the experiment with the constant rate pacer shows clear 2 ms steps. The duration of these steps corresponds the inter packet time, which is set by the pacer. Each consecutive lost packet adds one inter packet time to the RTT error. Therefore, the RTT error increases in discrete steps, as can be seen in Figure 6.11(a).

Note that the first step is significantly larger than the later steps. This is because here another effect is also at play: as described in Section 6.2.3 spin edges might be delayed because the sender has insufficient sending credit when it receives a spin edge from the remote endpoint. Because the sender will clear the valid bit on packets carrying delayed spin edges, the valid edge observer can filter out this effect. Indeed, the height of the steps in the ECDF of the valid edge observer decays multiplicatively. This corresponds to the geometrical distributions followed by the lengths of the loss bursts.

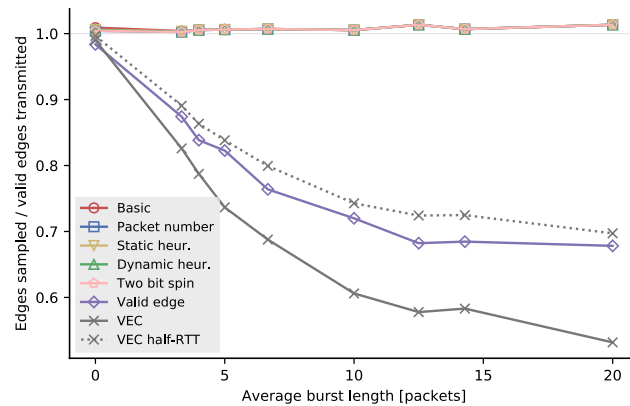
Similarly to the situation with random loss, only the valid edge and VEC observers can reject (some of) the incorrect RTT samples. The valid edge observer can only do this when the loss occurred *before* the observer. The VEC observer can also detect the loss of a packet carrying a spin edge after it has been observed.



(a) ECDF showing observer error under burst loss with an expected burst length of 10 packets.

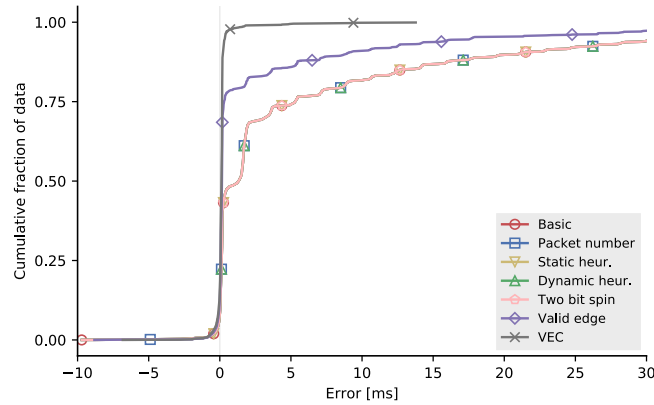


(b) Observer error for varying average burst lengths.

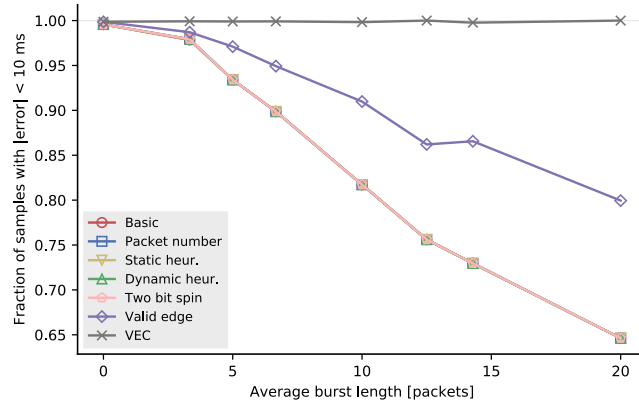


(c) Number of samples taken by each observer (normalized to the amount of valid edges transmitted by the endpoints) for varying average burst lengths. A plot with absolute numbers is shown in Figure B.4.

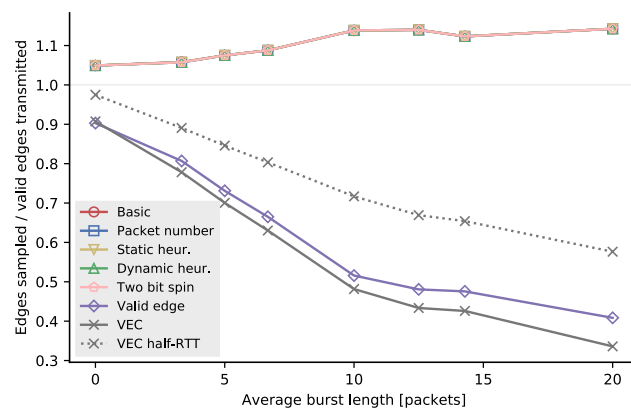
Figure 6.10: Graphs showing the effects of burst loss on the results of various observers, when a **fixed window** packet scheduler is used. Loss is simulated using the simple Gilbert model (see Section 2.5). Positive errors correspond to overestimated RTTs.



(a) ECDF showing observer error under burst loss with an expected burst length of 10 packets.



(b) Observer error for varying average burst lengths.



(c) Number of samples taken by the observer (normalized to the amount of valid edges transmitted by the endpoints) for varying average burst lengths. A plot with absolute numbers is shown in Figure B.5.

Figure 6.11: Graphs showing the effects of burst loss on the results of various observers, when a **fixed rate** packet scheduler is used. Loss is simulated using the simple Gilbert model (see Section 2.5). Positive errors correspond to overestimated RTTs.

Chapter 7

Measuring other metrics

The focus of the design of the spin signal has been on signalling a flow’s RTT to the network. However, the spin signal can also be useful to measure other metrics. This section gives a brief overview of possible alternative use cases. These have not been explored yet, but give interesting directions for future work. Most of these methods are not particularly useful for single shot measurements, but aggregated over time or per link they can be much more powerful.

7.1 Alternative marking based loss measurement

The spin bit can be seen as a self clocking variation of the alternate marking method described in RFC 8321 [23]. Therefore, measurement techniques proposed in RFC 8321 are also applicable to flows with a spin bit. These techniques rely on a flow of packets being split up in to *blocks*. RFC 8321 accomplishes this by alternatively marking groups of packets with a ‘1’ or a ‘0’ bit, which is exactly what the spin bit does! The most interesting technique in RFC 8321 measures loss between two observers by having each observer count the number of packets between two spin bit transitions. When the downstream observer counts less packets, loss must have occurred.

7.2 An additional loss bit

The spin bit can also be used to synchronize other measurements. For example, an additional *loss* bit can be added to the QUIC header. This bit is set by an endpoint when it detects a loss event, and cleared when it generates a spin edge transition. This does not only inform observers *if* loss has occurred, but aggregate information about *when* the loss bit is set also provides information about the loss rate. When the loss rate is high, the loss bit is likely to be set soon after the spin bit has transitioned. For lower loss rates, the loss bit is more likely to be set later in the block of packets with the same spin.

7.3 Spin based reordering and loss measurement

When packets are reordered over a spin edge, rapid transitions in the spin signal are generated. These are a clear indication that reordering has occurred. By measuring the number of packets between the start and end of “noisy” periods in the spin signal, the amount of reordering can be estimated. When the endpoint generated spin edge is marked with a valid bit or VEC, this method becomes more powerful.

7.4 VEC based loss and reordering measurement

When an endpoint generated edge is lost or reordered, this will cause the VEC to return to '00'. Observing how often this happens — perhaps combined with other proposed methods — can give insight in loss and reordering rates.

Chapter 8

Conclusion

This work investigates how QUIC can be modified to allow network elements to easily extract the RTT from a QUIC flow. More specifically, the addition of a latency “spin bit” — a single bit signal that toggles once per RTT — is evaluated. Because the spin bit toggles once per RTT, on path observers can directly measure a flow’s RTT by timing the duration between two spin bit transitions. Furthermore, when both directions of a flow are observed, the up- and downstream delays can be measured individually.

The spin bit has been added to a QUIC implementation and evaluated on an emulated network. It was found that the spin bit performs well under good network conditions, but that its accuracy deteriorates when reordering or burst loss occurs on the network. In order to mitigate these effects, two simple heuristics were applied to the resulting RTT estimates. Unfortunately, this had only limited success. However, observing the spin bit together with the QUIC packet number can successfully filter out the effects of reordering. But not those of loss.

When the QUIC packet numbers are encrypted, similar results can be obtained by using a two bit spin value (i.e. an incrementing counter) instead. Alternatively, a *valid* bit that marks every endpoint generated transition in the spin signal can be used. This results in similarly accurate RTT readings, but can lead to the rejection of a significant fraction of samples.

In order to add tolerance to burst loss, the valid bit can be expanded to a two bit signal: the VEC (Valid Edge Counter). The VEC counts the number of half-RTTs during which the spin signal has not been affected by loss or reordering. An observer can use this information to decide when accurate RTT samples can be taken.

When using the valid edge or VEC signals, endpoints can also signal when they *know* that the spin bit does not provide accurate RTT information. For example, because there is insufficient traffic on the flow. When endpoints do this, using the VEC results in near perfect RTT readings in all network and traffic conditions. However, when using the VEC, extreme network or traffic conditions might lead to very high sample rejection rates.

It can be concluded that the spin bit works. Under good network conditions it provides frequent, accurate RTT readings, and when network conditions worsen, the use of various enhancements allows for a trade-off to be made between accuracy and rejection ratio.

Bibliography

- [1] Information Sciences Institute, University of Southern California, “DoD standard Transmission Control Protocol,” RFC 761, Jan. 1980. [Online]. Available: <https://rfc-editor.org/rfc/rfc761.txt>
- [2] M. Handley, “Why the internet only just works,” *BT Technology Journal*, vol. 24, no. 3, pp. 119–129, 2006.
- [3] H. Jiang and C. Dovrolis, “Passive estimation of tcp round-trip times,” *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 3, pp. 75–88, Jul. 2002. [Online]. Available: <http://doi.acm.org/10.1145/571697.571725>
- [4] B. Veal, K. Li, and D. Lowenthal, “New methods for passive estimation of tcp round-trip times,” in *Passive and Active Network Measurement*, C. Dovrolis, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 121–134.
- [5] D. Carra, K. Avrachenkov, S. Alouf, A. Blanc, P. Nain, and G. Post, “Passive online rtt estimation for flow-aware routers using one-way traffic,” in *NETWORKING 2010*, M. Crovella, L. M. Feeney, D. Rubenstein, and S. V. Raghavan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 109–121.
- [6] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, “Bbr: Congestion-based congestion control,” *ACM Queue*, vol. 14, September-October, pp. 20 – 53, 2016. [Online]. Available: <http://queue.acm.org/detail.cfm?id=3022184>
- [7] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi, “The quic transport protocol: Design and internet-scale deployment,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM ’17. New York, NY, USA: ACM, 2017, pp. 183–196. [Online]. Available: <http://doi.acm.org/10.1145/3098822.3098842>
- [8] J. Iyengar and M. Thomson, “QUIC: A UDP-Based Multiplexed and Secure Transport,” Internet Engineering Task Force, Internet-Draft draft-ietf-quic-transport-07, work in Progress. [Online]. Available: <https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-07>
- [9] J. Iyengar and I. Swett, “QUIC Loss Detection and Congestion Control,” Internet Engineering Task Force, Internet-Draft draft-ietf-quic-recovery-07, work in Progress. [Online]. Available: <https://datatracker.ietf.org/doc/html/draft-ietf-quic-recovery-07>
- [10] M. Thomson and S. Turner, “Using Transport Layer Security (TLS) to Secure QUIC,” Internet Engineering Task Force, Internet-Draft draft-ietf-quic-tls-07, work in Progress. [Online]. Available: <https://datatracker.ietf.org/doc/html/draft-ietf-quic-tls-07>
- [11] K. Thompson, G. J. Miller, and R. Wilder, “Wide-area internet traffic patterns and characteristics,” *IEEE Network*, vol. 11, no. 6, pp. 10–23, Nov 1997.

-
- [12] D. Lee, B. E. Carpenter, and N. Brownlee, “Observations of udp to tcp ratio and port numbers,” pp. 99–104, May 2010.
- [13] B. Lantz, B. Heller, and N. McKeown, “A network in a laptop: Rapid prototyping for software-defined networks,” in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: ACM, 2010, pp. 19:1–19:6. [Online]. Available: <http://doi.acm.org/10.1145/1868447.1868466>
- [14] S. Hemminger *et al.*, “Network emulation with netem,” in *Linux conf au*, 2005, pp. 18–23.
- [15] E. N. Gilbert, “Capacity of a burst-noise channel,” *The Bell System Technical Journal*, vol. 39, no. 5, pp. 1253–1265, Sept 1960.
- [16] E. O. Elliott, “Estimates of error rates for codes on burst-noise channels,” *The Bell System Technical Journal*, vol. 42, no. 5, pp. 1977–1997, Sept 1963.
- [17] M. Kuehlewind, T. Buehler, B. Trammell, S. Neuhaus, R. Muenstenr, and G. Fairhurst, “A path layer for the internet: Enabling network operations on encrypted protocols,” 2017.
- [18] B. Trammell, P. De Vaere, R. Even, G. Fioccola, T. Fossati, M. Ihtar, A. Morton, and S. Emile, “The Addition of a Spin Bit to the QUIC Transport Protocol,” Internet Engineering Task Force, Internet-Draft draft-trammell-quic-spin-01, Dec. 2017, work in Progress. [Online]. Available: <https://datatracker.ietf.org/doc/html/draft-trammell-quic-spin-01>
- [19] W. Almesberger, “Linux network traffic control – implementation overview,” *5th Annual Linux Expo*, pp. 153–164, 1999.
- [20] S. McCanne and V. Jacobson, “The bsd packet filter: A new architecture for user-level packet capture.” in *USENIX winter*, vol. 93, 1993.
- [21] M. Sargent, J. Chu, D. V. Paxson, and M. Allman, “Computing TCP’s Retransmission Timer,” RFC 6298, Jun. 2011. [Online]. Available: <https://rfc-editor.org/rfc/rfc6298.txt>
- [22] D. Borman, R. T. Braden, V. Jacobson, and R. Scheffenegger, “TCP Extensions for High Performance,” RFC 7323, Sep. 2014. [Online]. Available: <https://rfc-editor.org/rfc/rfc7323.txt>
- [23] G. Fioccola, A. Capello, M. Cociglio, L. Castaldelli, M. Chen, L. Zheng, G. Mirsky, and T. Mizrahi, “Alternate-Marking Method for Passive and Hybrid Performance Monitoring,” RFC 8321, Jan. 2018. [Online]. Available: <https://rfc-editor.org/rfc/rfc8321.txt>

Appendix A

List of Acronyms

Acronyms

ACK	acknowledgement
ECDF	Empirical Cumulative Distribution Function
FIFO	First In, First Out
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
Minq	Minimal QUIC
NAT	Network Address Translation
NetEm	Network Emulator
Pcap	Packet Capture
Pinq	Piet's Minq
PLUS	Path Layer UDP Substrate
QUIC	Quick UDP Internet Connections
RFC	Request For Comments
RTO	Retransmission Timeout
RTT	Round Trip Time
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VEC	Valid Edge Counter
VPP	Vector Packet Processing

Appendix B

Additional plots

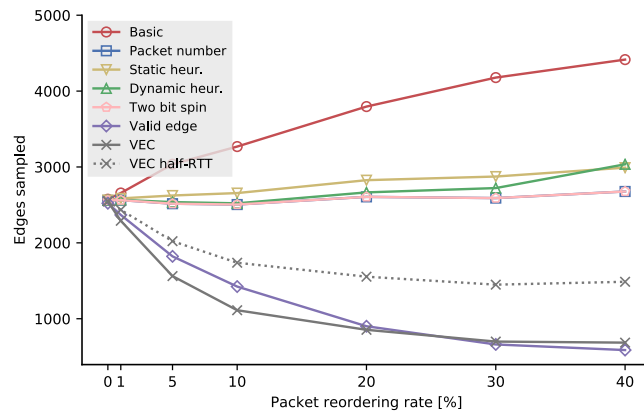


Figure B.1: Supplement to Figure 6.7. Absolute number of samples taken by each observer for various reordering rates.

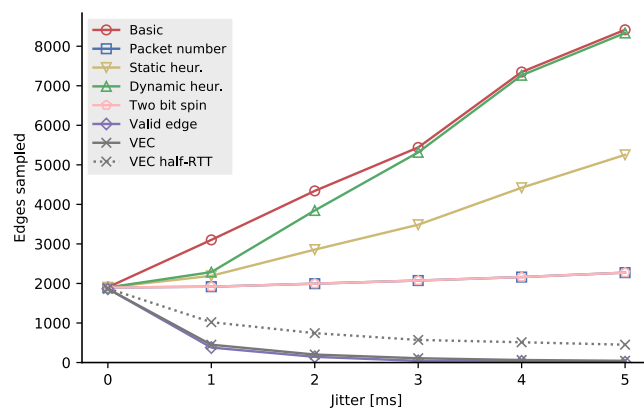


Figure B.2: Supplement to Figure 6.8. Absolute number of samples taken by each observer for various jitter rates.

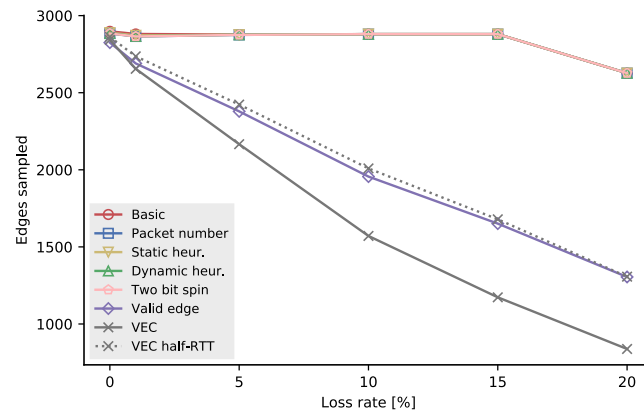


Figure B.3: Supplement to Figure 6.9. Absolute number of samples taken by each observer for various random loss rates.

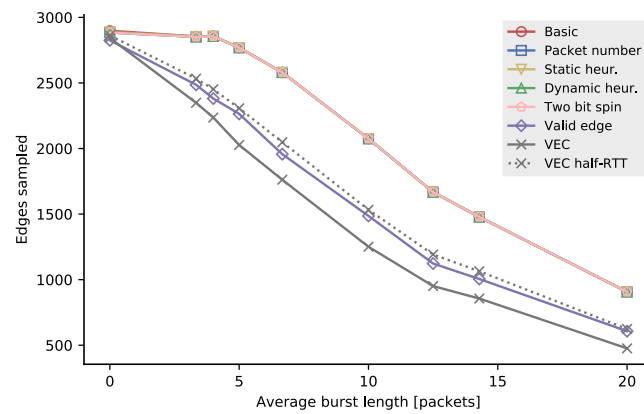


Figure B.4: Supplement to Figure 6.10. Absolute number of samples taken by each observer for varying average burst lengths, with a fixed window size.

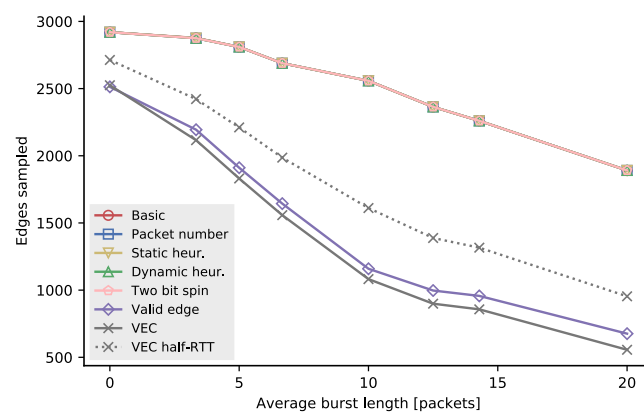


Figure B.5: Supplement to Figure 6.11. Absolute number of samples taken by each observer for varying average burst lengths, with a fixed pacing rate.