POLITECNICO DI TORINO

Master's Degree in COMPUTER ENGINEERING

Master's Degree Thesis

IMPLEMENTATION OF PERFORMANCE MONITORING TECHNIQUES FOR EVALUATIONS ON AN EMULATED L4S NETWORK

Supervisors

Prof. RICCARDO SISTO

MASSIMO NILO

FABRIZIO MILAN

FABIO BULGARELLA

Candidate

MATTEO GUARNA

APRIL 2024

Abstract

As networks develop and permeate ever deeper many aspects of today's society, the need for performance monitoring techniques has become crucial in a wide variety of contexts. Among such methods figure Explicit Host-to-Network Flow Measurement techniques (EFM), which allow to extract network metrics from a data transmission without accessing the connection's endpoints.

The objective of this thesis is twofold: on one hand, it involves implementing Spin Bit and Delay Bit, two EFM techniques capable of measuring in-network latency; on the other, it aims to test these techniques on L4S, a state-of-the-art network suite for TCP.

This research introduces the first functioning implementation of these EFM techniques over TCP. As a result, it highlights both the Delay Bit's intrinsic advantages over the Spin Bit, and L4S's improvements over classic TCP, presenting various use-case scenarios tested on both physical and virtual platforms.

Summary

In the digital age networks have become so intertwined with every single aspect of our lives that understating their relevance has become impossible: it's therefore self-evident that developing more fitting and efficient ways to measure the networks' characteristics is nowadays critical in many situations. Host-to-Network Performance Monitoring techniques are a new family of algorithms specifically designed to gather network metrics in the most agile and least disruptive way possible, i.e., with no access to the endpoints, and without introducing any additional data on the network under scrutiny.

The scope of this research encompasses two similar algorithms, specifically tailored for latency evaluation: the Spin Bit and the Delay Bit. The main objective of the thesis is deploying (for the first time ever) these algorithms over the TCP/IP stack, assessing their validity, and comparing the two solutions to examine whether the Delay Bit's theoretical advantages over the Spin Bit hold true in real-life scenarios. Most of the work hereby presented relies in fact on the patching of the Linux kernel's network stack, in order to allow network hosts to deploy, within the network connection, the algorithm-defined markings which are then exploited by an on-network observer to extract the required metrics. Performance Monitoring techniques were initially developed to provide a built-in way of measuring QUIC traffic, which hides by design any information from external observers. Still, their many advantages make them a compelling alternative to traditional performance evaluation solutions.

The network environment's choice too fell on a state-of-the-art technology: in order to better assess and evaluate the efficacy of the selected monitoring techniques, they were in fact employed to compare the classic TCP protocol with L4S. L4S, which stands for Low-Loss, Low-Latency, Scalable throughput, is an innovative connection-oriented suite of protocols aiming at improving TCP's performance, ensuring, as the name suggest, low and consistent delay, and virtually zero losses, translating into a stable, reliable, connection, which is critical for many cutting-edge network applications, ranging from live streaming to cloud services. L4S achieved

the status of IETF standard in early 2023 and many companies have been working on their own custom implementation of this solution since.

L4S, as a suite of protocols, relies on two key components to provide the described features. First, a brand new TCP Congestion Control algorithm is employed on the endpoints, and it is both truly scalable and relying on an Accurate ECN mechanism. Truly scalable CC algorithms are likewise defined as they can increase their throughput independently from the RTT, while Accurate ECN means that the algorithm constantly retrieves information regarding the state of the network, and can adapt to the network's conditions to optimize the flow and avoid saturation.

The second feature on which L4S is based is a Dual Queue solution, that must be implemented on all routers that might experience congestion. The solution consists in having two queues instead of one, with the latter specifically reserved for L4S traffic. The router must therefore be able to tell L4S segments from classic TCP segments, and by dividing the flows, it is able to recognize whether the queues are filling: if that happens, which means that a congestion is likely to occur, the queue overwrites L4S's marking, warning the endpoints which are then able to tune their transmission rate to avoid queue saturation, i.e., packet drops. This is also instrumental in reducing the RTT and its fluctuations, as much of the delay in a connection is created when packets have to wait in long queues before being forwarded: maintaining a short queue on L4S's side does not impact the throughput, and allows for a lower and more consistent end-to-end network latency. The scheduler handling both queues is also designed to avoid favouring any of the two flows, thus maintaining for each connection the same resource share they would acquire if the mechanism were not in place.

The testing environment put in place for this research had thus to accommodate L4S both on the connection's endpoints and on the routers: luckily, a ready-made implementation of both a dual queue algorithm (with the dualpi2 protocol) and of a L4S-compatible congestion control algorithm (in the shape of the Prague CC protocol) has been developed on Linux for testing purposes by the research team which worked on the IETF document. The first part of the whole research consists in crafting and validating said environment. Actually, two different solution are provided: a fully-virtualized network solely relying on Linux Network Namespace's virtualization primitives, which can be deployed on any Linux machine, and a more complex, although still very straightforward, physical network, comprising Linux servers filling in as both the endpoints and the network devices, as the Dual Queue implementation was already available on Linux, while actual routers would have required a custom implementation.

Both test plants were designed with an identical layout: four endpoints are put into two different LANs connected by a router. A network impairment is also used to emulate real-life conditions on both test plants, like delays, reordering, jitter, and losses. A congestion is put into place by managing the throughput, so that the upstream flow on the router exceeds the downstream flow, allowing the hosts upstream to generate more traffic than the router is able to drain. By doing so, hosts have to adjust to the downlink channel's capacity, thus requiring congestion control. The two host pairs across the router generate each its own flow: one over classic TCP with Cubic as Congestion Control protocol, and the latter an L4S-compatible flow with Prague. By doing that the Dual Queue has to handle each connection separately during the congestion, and the metrics acquired through performance monitoring provide a means of comparing the connections' behaviour, and to assess whether L4S is behaving as expected, and if it provides any sizable advantages over classic TCP.

Once the network environments were ready and validated, the most significant part of the work could begin, i.e., patching the Linux kernel to implement the Spin Bit and the Delay Bit. First though, it is necessary to understand how both these algorithms work. The Spin Bit's logic is very simple: the idea to mark a bit inside each packet either with the value 0 or 1, according to the following rules: one of the endpoints (the one starting the connection) starts sending packets all with the same marking, and the latter endpoints simply reflects them with the same value. As ACKs and other packets start coming back, the first endpoint knows that it must mark all packets with the opposite value to the last one received, resulting in a square wave which transitions once per RTT. This allows an on-network observer to extract the end-to-end latency by just looking at the transitions. The only issue is that in case reordering occurs, false transitions are created, resulting in false measurements. This is the exact reason why the Delay Bit was implemented by the TIM's working group for research and innovation: the delay Bit replaces the square waves with pulse waves, meaning only one packet is marked to one at the beginning and reflected between the endpoints indefinitely, with the observer taking measurements with each pulse. This solution is resilient to reordering as even rearranging the packet containing the sample set to 1 does not change the RTT measure by a meaningful amount, but is, at least in theory, susceptible to losses, as if the marked sample is lost, the measure is lost too. Also, the endpoints need a timer to be able to tell whether the marked bit was lost, and generate a new one. Still, theoretically the risk of this to happen is very low as the marked packet is one for each RTT, so the chance of losing exactly that packet is extremely low provided loss ratios which are compatible with any real network connection, albeit unstable.

The first choice towards implementing these algorithms was finding a suitable place inside the TCP/IP stack which could be used to carry the marking. The most obvious solution was that which was in fact adopted, i.e., using one of the reserved bits inside the TCP header. This choice proved fortuitous, as it's clear that both techniques require some connection-wide state variables to be stored inside the kernel: during the implementation phase, both the marking action and the control logic could therefore be implemented inside the TCP modules of the Linux kernel, at layer four of the OSI model, avoiding inter-layer communication which is in fact much more complex. It was in fact sufficient to define some state variables, introducing a parsing mechanism to extract the incoming value and update the state variable accordingly, and then an algorithm to mark the reserved bit accordingly. This workflow was basically identical for both the Spin Bit and the Delay Bit, even if in the latter case, some adjustments were needed to avoid segmentation happening on the lower levels of the network stack from introducing multiple samples on the same RTT.

The last piece of work consisted in deploying an on-network observer capable of analysing the marking on the segment, and correctly extracting the metrics. This time though, instead of developing a new custom solution, it was possible to rely on Spindump, an open-source application which, among its capabilities, supports Spin and Delay Bit parsing for QUIC traffic, and patching it to make it compatible with TCP.

Once the whole testing environment was complete and validated, it was finally possible to actually follow through with the experimentation. First, the Delay Bit and the Spin Bit were compared with a host-based monitoring technique, which acted as the benchmark for the analysis: the results confirmed the Delay Bit's resilience to network reordering, whereas the Spin Bit proved unusable, and its reliability even when losses were introduced. All L4S's analysis were therefore conducted using the Delay Bit to gather latency information: by changing the network conditions through the impairment, it was possible to obtain a wide array of scenarios which were instrumental both to analyse the nuances in the suite's behaviour, and to assess its resilience in the most diverse situations. As a result, L4S proved superior to classic TCP in almost every aspect and network conditions, in full accordance with both its design and all expectations.

In conclusion, this experiment reached its main goal of deploying a working implementation of two Host-to-Network EFM techniques, and comparing them in a simulation of real-life applications. Furthermore, these results were achieved while testing a new powerful connection-oriented technology which is collecting much praise and attention from many vendors in the field. Lastly, both the only currently available virtual L4S-compatible environment and the first Linux patch implementing Spin Bit and Delay Bit are now publicly available for researching and further developments.

Acknowledgements

ACKNOWLEDGMENTS

to Sole, my dearest one
whom words alone cannot express how grateful I am for her being the best part of my life

to my mother
who always represented for me a model of persistence, abnegation, and true ethics

to my father
who, despite often diverging views, always tried to instill into me a sense of vision and direction greater than his own

to my sisters
who always supported their big brother, even when I probably didn't deserve it

to my family
who always gave me everything that really counts

to my friends at the university
who welcomed me within their group, and without whom my struggles in my course of studies would have increased tenfold

to, Fabio, Fabrizio, and Massimo
who really gave me all the help I needed in order to succeed in this research

Table of Contents

List of Tables

List of Figures

Acronyms

AccECN	Accurate ECN
AIMD	Additive Increase Multiplicative Decrease
AQM	Active Queue Management
BBR	Bottleneck Bandwidth and Round-trip Propagation Time
CC	Congestion Control
CLI	Command Line Interface
CWND	Congestion Window
DCTCP	DataCenter TCP
ECN	Explicit Congestion Notification
EFM	Explicit Host-to-Network Flow Measurement

LAN	Local Area Network
MTU	Maximum Transmission Unit
TCP	Transport Control Protocol
VLAN	Virtual LAN
WAN	wide Area Network

Chapter 1

Introduction

1.1 Performance monitoring and L4S

As networks continue to grow in complexity and significance within the digital landscape, ensuring optimal performance and reliability has become an ever-present challenge. Network operators, researchers, and service providers are seeking ways to enhance the quality and performance of their networks to meet the demands of today's data-intensive applications. Performance monitoring techniques play a crucial role in this context, offering valuable insights into the network behaviours, efficiency, and responsiveness.

The primary objective of this thesis is the implementation of some performance monitoring techniques on a L4S environment, in order to further validate their employment and showcase their effectiveness within an established protocol like TCP, but also in the context of a state-of-the-art testing ground. L4S was in fact selected as the subject for this testing activity due to its innovative nature, its relevance, and its ability to improve the performance of a traffic flow over the network, thus providing the opportunity to compare flows with clear-cut behaviours and assess the performance techniques' prowess even in complex and diverse conditions.

1.2 The objectives of this research

This research was conducted in collaboration with the TIM Group at the Telecom Italia Lab in Turin. Its goals are thus relevant to TIM's interest in the development of new technologies to eventually improve their services' performance.

The studies hereby presented aim at testing L4S's performance on a cabled network also hosting classic connections when congestion occurs. Comparing delay, jitter and bandwidth occupation will allow us to obtain metrics to describe the performance improvements offered by the architecture, draw some conclusions about its advantages and disadvantages, and try to assess the gains network providers might obtain through the adoption of this new technology.

All of this is done thanks to a set of passive monitoring techniques, whose effectiveness in providing the results is equally of great interest: Explicit Flow Measurement (EFM) techniques are relatively recent and were developed with consistent contributions by the TIM's working group for research and innovation: their successful application is hence crucial in substantiating their value and advocating for their adoption across diverse scenarios.

1.3 Thesis structure

This section contains a brief overview of each of the following chapters and their content. Namely:

- Chapter 2 provides an overview of the L4S architecture, its characteristics and its core elements, and illustrates its relevance in today's network landscape.
- Chapter 3 illustrates the thought process behind the definition of the experiment, its requirements and provides a logical overview of the main elements required in the test plant.
- Chapter 4 provides an in-depth dive of passive performance monitoring, explicit flow measurement as a whole as well as the specific EFM techniques employed inside the the test plant.
- Chapter 5 describes actual configuration process of the test plant.
- Chapter 6 gives a detailed explanation of the implementation of the EFM techniques on the physical hosts.
- Chapter 7 describes the tests aimed at comparing Spin Bit and Delay Bit, provides and discusses the obtained results.
- Chapter 8 describes the use cases for L4S's analysis and comparison with classic TCP, provides and discusses the obtained results.
- Chapter 9 draws the conclusions of the experimentation and discusses future work.

Chapter 2

The L4S Architecture

2.1 Overview

The Low-Latency, Low-Loss, Scalable Throughput (L4S) suite represents a groundbreaking standard for managing congestion control on the internet. Initially conceived by Bob Briscoe, Greg White (CableLabs), and Koen De Schepper (Nokia), L4S has been formally standardized in 2023 as a new component of the internet protocol suite (TCP/IP) following its approval by the IETF committee[1]

The L4S network architecture is dedicated to internet applications that require low latency to operate, like web applications, conversational video, online gaming, remote desktop, and many cloud-based services. The main challenge that these technologies have to face in order to ensure a seamless user interaction consists as a matter of fact in the delays that might occur in any network connection: these are not the base time that a packet takes to reach its destination (i.e., the RTT or half-RTT), yet the additional intermittent latency mainly caused by the queuing process that packets incur into as they have to be forwarded by the network devices when a congestion occurs (which means the number of incoming packets is higher than the capacity of the link on which they have to be forwarded to). Delays like these add up and create spikes in the time required by the packets to reach their destination.

The goal of the L4S architecture is therefore that of providing a connection with as little transmission delays as possible, thus eliminating the jitter in the RTT and ensuring a more seamless interaction between the endpoints.

L4S relies on two complementary technologies, namely:

- a TCP congestion control algorithm able to quickly adjust its flow in order to avoid queuing - and therefore delays - both fast and precisely, in order to fully exploit the channel's capacity at any given time;
- a differentiated queue on the network hosts prone to congestion, in order to handle separately, but fairly, the L4S flow from the normal flow, and also to inform the L4S endpoints if a congestion is likely to occur.

These two elements must be able to interact together through an Explicit Congestion Notification (ECN) algorithm.

2.2 The congestion control algorithm

L4S requires a scalable congestion control algorithm, which is defined as "*one where the average time from one congestion signal to the next (the recovery time) remains invariant as flow rate scales*"[2]

This is critical for L4S technologies, due to the fact that in order to efficiently exploit the available bandwidth at any given time, the flow rate must be able to reach full capacity fast[3]. Classic CC algorithms like Reno take many round trips to reach the channel's full capacity: in fact TCP Reno increases its CWND linearly, and while more efficient alternatives like CUBIC are faster, the constant trend of network communications is moving towards links of greater and greater capacity to accomodate for the ever increasing needs of the users, which hinders classic protocols from being able to quickly fill up the channel.

Scalable congestion control algorithms are therefore necessary to ensure a good starting point for the L4S architecture, and in the last years many options were developed: DCTCP is a relatively affirmed solution, although it was developed to work in data centers' networks; a more recent solution is TCP Prague, which extends the benefits of DCTCP for WANs.

2.3 The Accurate ECN mechanism

CC algorithms can be divided into three categories[4]:

- black box algorithms operating with no knowledge of the state of the network other than the binary packet drop feedback upon congestion (e.g., Reno, Cubic);

- grey box algorithms operating with no direct knowledge of the network state, but relying on measurements taken by the endpoints which provide some degree of information about the situation of the link (e.g., Vegas);
- green box algorithms relying on signalling methods implemented by the routers in order to obtain a direct knowledge of the network (e.g., BBR, DCTCP, Prague),

While a scalable CC is the foundation of the L4S framework, the keystone of the architecture is the accurate ECN mechanism, a green box solution that allow for a timely flow reduction when needed.

ECN means explicit congestion notification, and it describes[5] algorithms relying on the routers' ability (usually through AQM) to mark packets when a congestion is likely, instead of dropping them, thus signalling the endpoints ahead of the actual congestion event: this allows them to reduce the flow and possibly avoid the congestion from happening. ECN is a mechanism therefore able to differentiate between what could be called "hard congestion signals", i.e., the actual packet losses due to the congestion, from "soft congestion signals", namely the marking made by the routers to inform the endpoints of an incoming congestion. In the latter case, if the situation on the network isn't servery compromised yet, the endpoints can adjust the flow rate through a less severe reduction in order to avoid congestion from actually happening, thus preserving the bandwidth.

Accurate ECN is a more sophisticated version of the standard ECN[6], that provides more than one feedback signal per RTT in the TCP header, therefore allowing for an even more precise control over the flow.

2.4 The dedicated queue

As mentioned above, the AccECN algorithm requires a signalling mechanism in place on the routers in order to inform the endpoints of the approaching of a congestion event. The aim of this section is therefore that of describing all requirements that need to be taken in account on the router to successfully implement the L4S framework.

L4S as a technology isn't meant to be deployed alone in the entire network: as a matter of fact, it's unreasonable to think that a network technology that requires both dedicated endpoints and network devices might become ubiquitous on the internet: the L4S architecture was hence designed to work in mixed environment with many flows and even unaware network devices. In fact, most routers on a

WAN can potentially be totally unaware of the L4S solution, that needs to be deployed only on the nodes where a congestion is likely.

First of all, capable routers must handle the L4S traffic in a dedicated way, and in order to do that they require a queue dedicated to the L4S packets. This is due to the fact that the delay the solution aims at eliminating originates in the queues on the routers: while a buffer is really necessary to avoid too many packet losses, it also introduces a space where packets can be queued up, considerably increasing their forwarding time. Obviously a shallow buffer is not the solution, because it makes the device less capable of handling a temporary burst of rate higher than the router is able to handle, and results in an unreasonably (and unnecessarily) high number of packet drops.

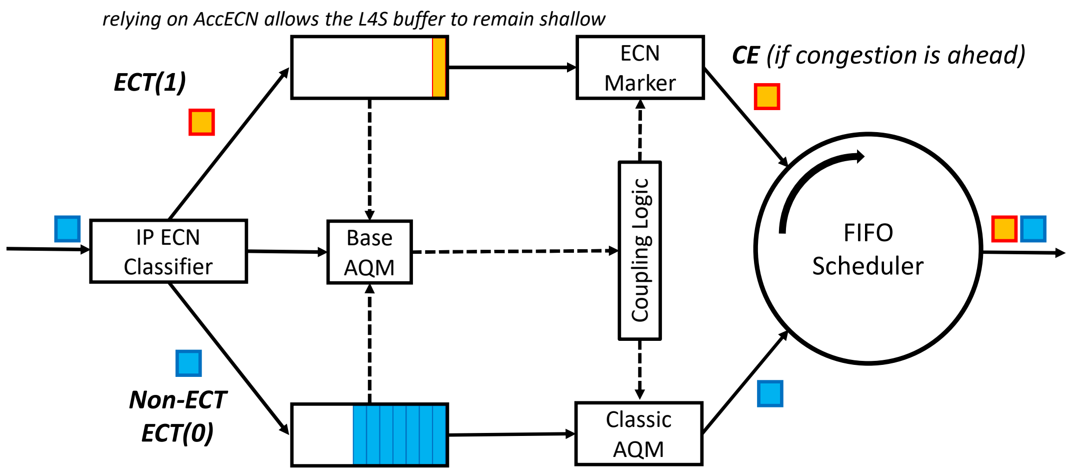


Figure 2.1: Schema of the dual queue structure with flows separation

The balance to strike here is having a queue which has a good capacity to deal with temporary bursts, while keeping it mostly empty. Therefore the L4S solution relies on a dedicated queue for the L4S traffic, which must therefore be flagged in order to be told apart from the classic traffic. The dedicated queue is kept shallow in order to decrease each packet's waiting time before retransmission, and this is done by informing the endpoints of an incoming congestion.

The only problem is that this solution must take into account that with two parallel queues, the flows are basically handled in two different ways, in an approach that might look like diffserv. L4S's objective is though not that of creating a privileged traffic service: to the contrary, it aims to enhance its flow performance without hampering the other connections. Therefore, the bandwidth should be shared as fairly as possible, in order not to compromise the network usage by non-L4S applications.

2.4.1 How to tell L4S packets apart from classic traffic

As previously mentioned the Dual Queue works by inserting L4S packets in a different queue from the one used by the remaining traffic. This is possible as L4S packet are required to have a specific marking inserted in the two-bit ECN field inside the IP header. In fact, while traditionally the ECN field was used as follows:

- 00 $\rightarrow$ ECN not supported (Not-ECT),
- 01 or 10 $\rightarrow$ ECN Capable Transport (ECT(1) or ECT(0)),
- 11 $\rightarrow$ congestion experienced (CE);

the L4S standard redefined the codification to make space for the L4S traffic:

- 00 $\rightarrow$ ECN not supported (Not-ECT),
- 01 $\rightarrow$ traffic eligible for L4S treatment (ECT(1)),
- 10 $\rightarrow$ ECN Capable Transport (ECT(0)),
- 11 $\rightarrow$ congestion experienced (CE).

This allows the Dual Queue to differentiate between the two traffic classes and avoid latency to build up in the buffer used by L4S due to less responsive CC protocols.

2.4.2 How the endpoints are informed by the router

Lastly, the ECN field is also used, in true ECN fashion, by the queue itself to inform the endpoints if a congestion is likely to occur: the CE value is set if the L4S queue is growing, which means that a congestion is at least plausible. Outgoing packets marked with CE then reach the endpoints, which can therefore adjust the throughput trying to prevent packet loss.

2.5 The relevance of the L4S architecture

L4S as a technology is highly important for any internet provider: in fact web technologies require more and more speed and performance, and while upgrading the network infrastructure can accommodate for that, it's obviously much more expensive than simply deploying a new technology that can increase the efficiency of highly demanding applications. This acquires even more relevance by putting into account the downward trend of consumer prices worldwide[7][8][9] which makes the topic even more relevant.

On the other hand, many of today's more crucial and promising technologies rely heavily on real-time remote interaction, like cloud-based technologies, online video-chat applications, internet video streaming and gaming, and might benefit heavily from a differentiated, reliable connection over the internet. An equally important perk L4S provides though relies in all the web applications and services which do not yet exist, due to the lack of possibility to establish a high-bandwidth, low-latency, reliable connection over the internet. Thus, implementing L4S also offers a unique opportunity for the development of tomorrow's innovations.

Chapter 3

Testing Ground Architecture

3.1 Preliminary considerations

The best way to thoroughly examine the L4S architecture is analysing a congestion scenario on a network hosting both L4S and classic traffic, in order to compare, through EFM techniques, the flows' behaviour when resources are limited and must be shared. This scenario is indeed most interesting because it provides a direct comparison aming different traffic flows with peculiar characteristics, and also allows showcasing L4S's behaviour in its most representative use case.

The test plant must therefore be a reflection of this objective: multiple hosts need to be able to exchange traffic flows among one another, across a single congestion point.

3.1.1 Active and passive monitoring

Equal attention must still be given to the measurement process to use in order to obtain the data. Network measurement techniques for performance evaluation can be broadly categorized in two main groups[10][11]:

- active monitoring, which require additional traffic to be injected in order to obtain measurements;
- passive monitoring, which does not require additional traffic, thus representing the less intrusive alternative.

Passive monitoring in most cases just requires the endpoints to implement a packet marking technique, relying on the reserved bits inside the TCP header (some techniques can also do without any marking), and offers therefore two great advantages: once implemented it can be applied on any kind of existing traffic, and is therefore more suited to provide realistic metrics; furthermore, it makes it easier for an observer placed on the network to gather information, because it doesn't need to perform any kind of deep inspection.

Still, passive monitoring requires a kernel-level patch at the endpoints, which is not straightforward and will be discussed in depth in chapter 6.

3.2 Testbed architecture

In order to create a congestion between a classic and a L4S flow, four hosts are needed as endpoints. The congestion scenario then implies the presence of a router, which means that senders and receiver have to to be deployed on two separate LANs.

A device must also be able to perform traffic monitoring according to the requirements defined above. This device, which from now on will be called observer, must be able to access the entirety of the traffic flowing through the test plant's network, and either store it or directly process it and provide with the required metrics.

3.2.1 Endpoints characteristics

Two endpoints (one for each subnet) need to support a L4S-compatible TCP protocol with an AccECN congestion control algorithm, while the other pair of hosts just needs to be able to generate traffic through a classic TCP flavour. All of them must also implement the required packet marking algorithms, and must therefore allow for easy network stack modifications to be introduced.

We can define for each pair two distinct roles: server and client: the former accepts incoming connections and generates the traffic, while the latter establishes the connection while only acting as the receiving end of the transmission, by acknowledging the incoming packets.

Regarding the TCP CC flavours the choice fell on Prague as the L4S-compatible protocol, thanks to its design optimized to operate on the wide Internet network; regarding the classic TCP CC protocol Cubic was chosen for its wide spread use and good performance at moderate bandwidths.

3.2.2 Router characteristics

The router needs just to have its NICs configured with the dualpi2[12], which is a dual-queue AQM protocol built for L4S, thus also implementing the dual queue (as described in 2.4) on the interface buffers. As the network's single congestion point, the router must also be deployed so that the congestion can actually be triggered, meaning more traffic must be allowed to come in than the downwards link's capacity actually supports.

3.2.3 Network characteristics

The network must be divided into two LANs to have the traffic flowing across the router, with both the clients' pair and the servers' pair each connected one network interface respectivelt. Thus, servers generate traffic towards the same router port (granted its bitrate is lower then the NIC's capacity), which is then forwarded to the NIC connected to the clients: if the latter's link does not support the incoming bitrate, a congestion occurs, and the AQM system is going to be able to ensure the Prague flow's correct behaviour.

Last but not least, a way to introduce a base delay on the network must be considered: in fact Prague, being an AccECN congestion protocols rests on the employment of multiple control messages over a single RTT. Thus, a test plant striving to resemble a real-world scenario, should not stray from employing some system to simulate delays (and possibly also additional losses) on its simulated network.

Drawing from all requirements discussed in this chapter, the image below shows a logical rendition of the test plant architecture:

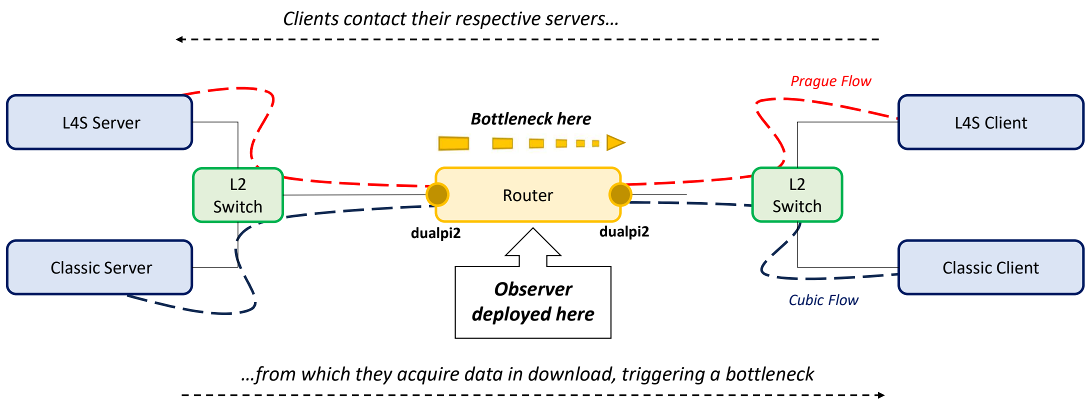


Figure 3.1: Logical rendition of the test plant

Chapter 4

Passive monitoring in depth

4.1 Packet marking for passive measurements

In section 3.1.1 the two main categories of network measurements techniques were already presented, and a passive solution was deemed more suited for most real-life scenarios. Still the provided definition remains broad and needs to be detailed to understand the relevance of the the strategy here employed. Passive measurements in fact may be also applied to the traffic as-is, for example by trying to extract the latency metrics through the timestamp sometimes carried by the timestamp option inside the TCP header, or by measuring the time elapsed between a packet transmission and its acknowledgement. These strategies work but also carry some problems: for this reason a new family of solutions was envisioned, relying on packet marking techniques, i.e., algorithms that carry information that can be decoded by an observer in charge of extracting the data and compiling the metrics.

4.1.1 Introduction to Explicit Flow Measurements

Passive monitoring solutions relying on packet marking were called Explicit Host-to-Network Flow Measurement techniques[13]. They were in fact first developed in close cooperation with the QUIC community in order to allow monitoring a protocol designed for security, and thus hiding by nature all information which might be exploited to perform monitoring activities. EFM techniques hence owe their name to the fact that they carry explicit information in a context where concealment is the keyword.

TCP nonetheless proves to be a suitable platform for passive monitoring implementation, thanks to the reserved bits inside its header, which can be easily repurposed to carry the marking bits all techniques require.

4.1.2 EFM techniques

Currently there are several EFM techniques available for implementation, but most of them just provide the same two metrics (i.e., delay and losses) in different ways. Both dimensions are obviously critical for L4S' performance evaluation, as the name itself suggests that its characteristics are low latency and low loss. Throughput on the other hand, while equally important, does not require any tailored technique due to the straightforwardness of its measurement.

The original target of the research consisted in developing both latency and loss techniques: unfortunately, the chosen loss technique, i.e., the Q Bit, wasn't implemented due to time constraints which didn't allow to properly address the technique's integration in the context of TCP (the Q Bit is in fact already in use in other network protocols). Losses metrics were therefore obtained from the endpoints, while the Spin Bit and the Delay Bit were used to extract latency metrics. The reason for this choice lies in the fact that the Spin Bit is the simplest and most widespread latency EFM technique, while the Delay Bit is still a simple yet more accurate alternative recently developed at TIM: assessing how the Delay Bit fares in comparison to the more well-known Spin Bit has been in fact a key target of the research, in order to further substantiate its benefits and advocate for its adoption.

4.2 The Spin Bit

The Spin Bit[14] was the first packet marking technique to be conceived, and it's aimed at performing passive latency measurements on a network. Currently, it's optionally implemented in the QUIC protocol.

The Spin Bit idea is to exploit a bit inside the packet's header to create a square wave signal on the data flow, whose length is equal to the RTT. By doing so, an observer in the middle can therefore measure the end-to-end RTT only by watching the Spin Bit's transition, granted it's able to perform packet inspection and knows where the information is located.

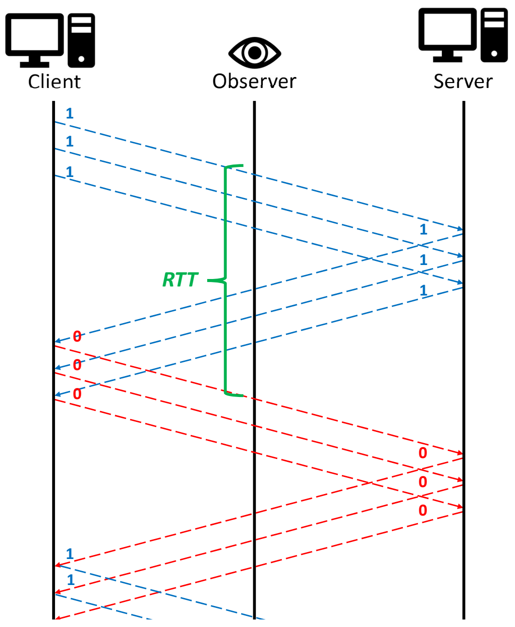


Figure 4.1: Spin Bit algorithm

The Spin Bit technique works as follows:

- *when the client receives a packet with the packet number larger than any number seen so far, it sets the connection spin value to the opposite value contained in the received packet;*[14]
- *when the server receives a packet with the packet number larger than any number seen so far, it sets the connection spin value to the same value contained in the received packet.*[14]

This technique has its own limitations, mainly due to reordering which might blur the edges of the square wave signal generated, which is the main reason why the delay bit represents an improvement over this solution.

4.3 The delay Bit

While from a theoretical standpoint the Spin Bit is a perfect technique to obtain latency measurements, in a concrete scenario packet reordering may blur the edges of the square wave resulting in decreased accuracy.

The Delay Bit on the other hand sets the marking bit just once per round trip, resulting in a single packet (called delay sample) bouncing back between the endpoints throughout the connection. This allows the observer to measure both the right and the left half trip delays, and thus the whole RTT.

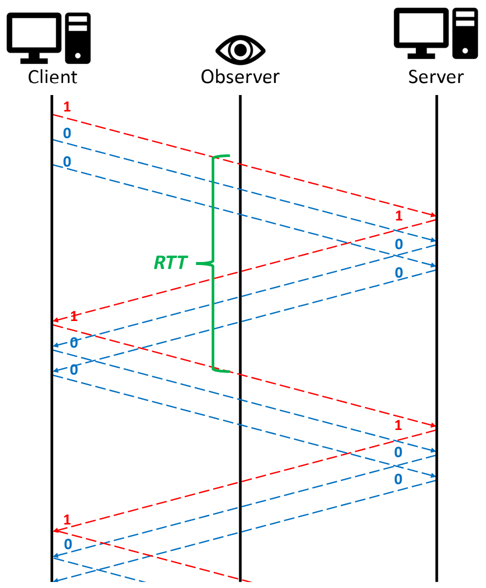


Figure 4.2: Delay Bit algorithm

The algorithm's behaviour is comprised of two phases: generation and reflection.

4.3.1 The generation phase

Generation of the delay sample is only entitled to the client, which is required to maintain a variable ds_{time} referencing the timestamp of the sample transmission.

The variable must be updated at every subsequent re-transmission and it's of paramount importance for the following reasons:

- the delay sample might be lost or discarded.
- the server might have reduced its transmission rate, withholding the delay sample.

In both situation the accuracy of the measurement is going to be compromised, and therefore the timestamp comes in handy: when a packet is generated, the time elapsed by the last delay sample transmission is checked, and if it's greater than a predefined variable T_{max}, a new time sample is created and the variable reset.

Obviously the T_{max} must be defined to be greater than the maximum RTT on the network, plus a fixed amount, in order to avoid triggering unnecessary retransmissions but also keep the mechanism idle for too long. The T_{max} can be either defined in advance, if the network topology and the traffic behaviour are known, or dynamically computed during the transmission. As far as the scope of this research goes, a fixed value is sufficient.

4.3.2 The reflection phase

The reflection phase is almost identical for client and server: both endpoints, upon delay sample reception, mark the delay bit on the first packet produced in the opposite direction, thus effectively "reflecting" the sample back to their counterpart. The client then, as previously mentioned, is also bound to update its own ds_{time}.

4.4 The Q Bit

The sQuare Bit (Q Bit) produces, similarly to the Spin Bit, a square wave between client and server. In order to allow measuring the losses though, the client marks N packets and then inverts its marking, thus generating a signal with period equal to $2 * N$. This simple method makes it possible for an external observer to count the packets in one direction and check the upstream losses, between the server and the observer. Thus, to evaluate the amount of packet lost throughout the connection, each endpoint produces its square wave independently and the observer separately analyzes each flow direction. The downstream losses can then be estimated as equal (on average) to thee upstream losses in the opposite direction.

In order to do that though the observer needs to learn the number N making up the Q block length, which is chosen by each endpoint and must be inferred by the observer. N must be a power of 2 and at least equal to 64. The power of 2 ensures that the observer is able to guess the Q block size; furthermore, being reordering a common event that can cause false measurements, the observer must count the packets belonging to a Q block up until X packet after the transition occurred (with X necessarily lower than N/2): this implies that a small number makes the measurement less resistant to blurred edges, and in case of a spike in the losses might also make harder to guess the right value for N.

Chapter 5

Test Plant Setup

5.1 Premise

According to the considerations made in chapter 3 it's now possible to actually build the environment where the experiments will take place. There are basically two possible alternative solutions which are now going to be discussed:

- creating a virtual environment which is capable to host the entire network
- deploying the network as a simple physical environment with real computers and network devices

As both solutions have their own strengths and weaknesses, the only sensible decision was to implement both the virtual and the physical test plants, and they are in fact discussed in the following sections.

5.2 Virtual scenario

The virtual solution offers many obvious advantages, mainly:

- independence from peculiar behaviours and characteristics of the single machines, that might impact the performance measurements of one specific traffic flow;
- higher simplicity, portability, reproducibility.

There are many different options to create virtual networks, basically either relying on full virtualization or para-virtualization. Still, in the Linux world, both are based upon virtualization primitives, i.e., Cgroups and Namespaces. The easiest and most efficient solution is therefore that of using Network Namespaces to isolate network functions inside a machine and thus mock the various elements of the test plant.

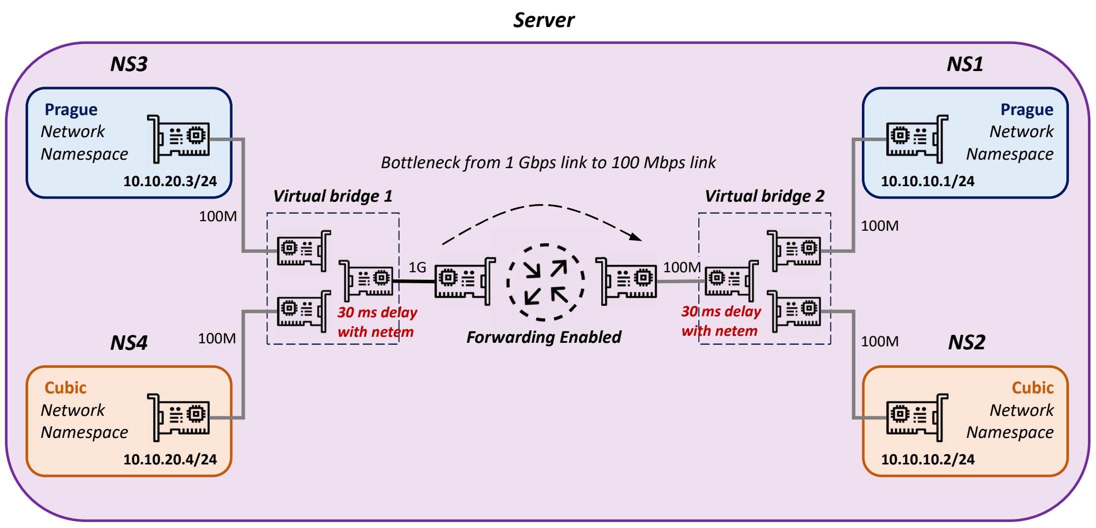


Figure 5.1: Virtual test plant representation

5.2.1 Kernel patch for L4S support

The L4S architecture relies on two technologies: a scalable congestion control protocol on the endpoints and an additional dedicated queue on the nodes prone to congestion. The L4STeam open source project on GitHub provides a patch for the Linux 5.15 kernel version with both Prague and Dualpi2[15]. By just following the guides provided inside the repository with a few a precautions, it it possible to deploy the required functions on any Linux machine with an OS version supporting the Linux 5.15 kernel. In order to install the kernel it's sufficient to follow the guide inside the README.md in the repository mentioned above. Any information regarding the setup of a Linux host and deployment of the L4S architecture is available at appendix A.

5.2.2 Deploying the Namespaces

At this point the host can utilize all the protocols defining the L4S architecture. The next step is then that of creating, using virtual network interfaces and bridges, a network topology mirroring the architecture described in 3.2. Network namespaces are then used to isolate the virtual NICs in charge of generating and receiving the traffic from one another as well as the router, thus creating completely independent network environments that can be used as hosts in the experiment.

All information regarding how the configuration was achieved as well as a bash script to reproduce this exact configuration are available at appendix B.

5.2.3 Defining the bottleneck

Lastly, the network must be designed to host a reduction of the available bandwidth in the path from the client to the server. It's possible to achieve this by creating $100Mbps$ links connected to all hosts, and then to set up a $1Gpbs$ link connecting the servers to the router: this allows all $200Mbps$ of traffic coming from the servers to reach the NICs behaving as router. Still, by limiting the channel connecting the remaining NIC to the clients to $100Mbps$, a bottleneck on the router is created, showcasing the behaviour of dualpi2 in the dealing with congesting flows.

5.2.4 Generating the flows

Both flows are generated via iperf3, a tool for network performance measurement and tuning. Iperf3 offer many features for customizing flows, and most importantly, it provides end-to-end retransmissions: since the Q bit couldn't be implemented, end-to-end techniques are the only way to measure the losses, which are essential to evaluate all use cases hereafter examined.

The result (fig.5.1) is a fully-virtualized test plant deploying all the elements needed to test L4S and compare it to a TCP Cubic flow. Each host can be directly controlled by command line by just preceeding the instruction to execute with the namespace command, as reported in the example below:

```
    sudo ip netns exec [namespace name] [actual command]
```

5.3 Physical deployment

On the other hand, while the physical rendition of the test plant might give back less "universally valid" results, it is in fact maybe more interesting because it allows to notice the actual constraints of a real network. Now then the physical implementation of the test plant is discussed, both according to the considerations made in chapter 3 and the machines which were available at TIM laboratories.

5.3.1 General setup for the physical machines

The TIM Lab had just four servers freely available, which are too few to have all endpoints, the router and possibly an impairment each on a separate machine. For this reason the endpoints were grouped with both traffic generators and both traffic receivers on just one machine respectively. More details on that later.

The available servers came without any OS yet, hence Ubuntu 22.04 was installed in its server version on all machines, saved the router which required the desktop version. The router is in fact the main observation point for the experiment and Ubuntu Desktop allows an easier use of desktop-oriented applications like Wireshark.

The OS choice fell upon Ubuntu 22.04, thanks to its compatibility with the Linux Kernel 5.15 version, which was used by L4STeam to create the patch for Prague and dualpi2. The installation of the new Kernel on all machines followed the exact procedure detailed in section A.2.

After the installation, the servers were configured with static IP addresses and connected to a management network, in order to enable their control via SSH (for the endpoints) and RDP (for the router server).

5.3.2 Configuring the endpoints

As anticipated above, just four machines were available, while the experiment architecture as it was designed requires at least five if not six entities (four endpoints, a router, and possibly an impairment simulating delays on the network). The solution presented here relies on the network namespaces techonlogy (ref. section 5.2.2) in order to have just two machines hosting respectively both servers and both clients. This is possible as the servers have many physicals network interfaces that can be assigned to isolated network environments, effectively replicating the behaviour, of multiple hosts on the network. By doing that, only four machines are needed, and there are also advantages in the synchronization of the traffic generation process across multiple hosts.

5.3.3 Configuring the router

The router is the heart of the system, as it needs to deploy dualpi2 on its interfaces, but also for it is the main observation point for test plant validation and data collection, as all traffic flows across it. For this reason it required the installation of Wireshark, a packet inspection tool, to check the correctness in the protocols' behaviour, and Spindump, a command-line utility to at acquire in-network metrics.

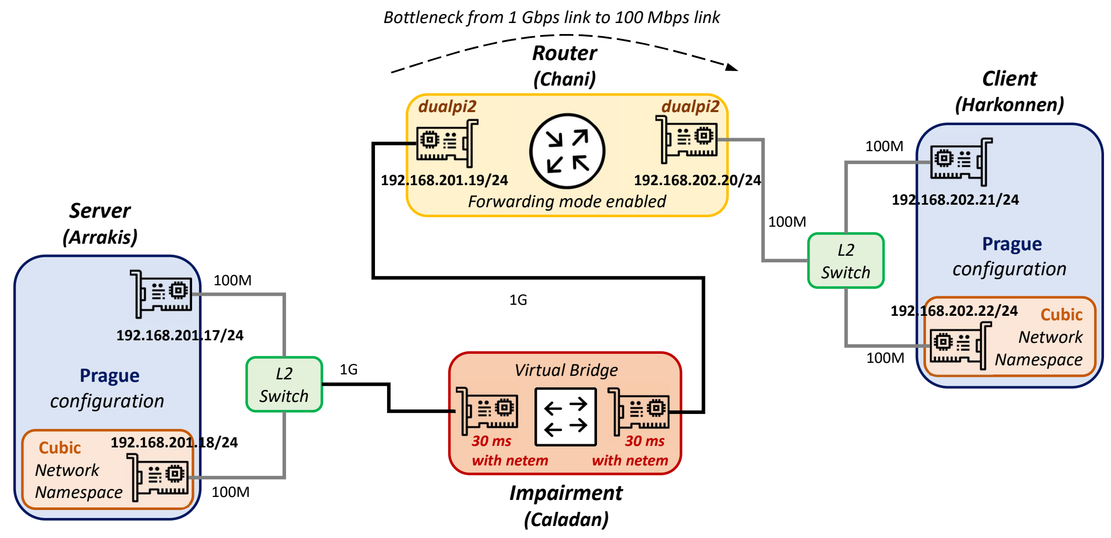


Figure 5.2: Physical test plant representation

5.3.4 Setting up the experimental network

Once the hosts and the router were ready, the network for the experiment was set up using Ethernet cables and L2 switches. According to the requirements discussed previously two separate LANs were configured through assignation of the addresses and the definition of the routing tables. Then, the channel's capacity was set to create a congestion by mirroring the configuration laid out in 5.2.3. Finally, iperf3 was used to generate the flows, as detailed in 5.2.4

A more in-depth guide of the configuration process is available at appendix C. The achieved configuration is shown in fig. 5.2

5.4 Additional requirements for dualpi2 compatibility

As better explained in appendix D, dualpi2 could not work properly when deployed as-is: in fact, even though the Prague endpoints were operating correctly and were thus able to keep low delay and zero retransmissions on the link, the share of traffic was much lower than Cubic's when a congestion was triggered.

After many researches and trials at least one solution arose: adding the traffic control tbf discipline to all dualpi2 interfaces solved the issue and allowed to obtain a fairer behaviour, when coupled with a reasonable latency parameter (at least a few milliseconds). This expedient was therefore employed throughout all tests as an indispensable part of the L4S architecture.

```
    sudo tc qdisc add dev [network interface] root handle 1: tbf rate
    1000mbps latency 10ms burst 1540
    sudo tc qdisc add dev [network interface] parent 1:0 dualpi2
```

Chapter 6

Implementation of Passive Monitoring Techniques

6.1 Introduction

The test plant is now completed, but it still lacks the traffic monitoring capabilities, which have to be implemented both on the endpoints, in order to perform the marking according to the selected algorithms, and on the router, which is the only device that is able to act as the observer for the entire network.

The endpoints need thus to be able to mark the packets before transmission, and to do that the Linux kernel can be modified according to the Spin Bit, Delay Bit and Q Bit techniques. The observer can instead either save all transmitted packets for further processing, or analyse them as they flow, returning only the required metrics. The latter option is definitely more viable and can be implemented without much struggle relying on Spindump[16], which is a software piece designed to inspect the traffic flowing through the host or a specific interface, and provide some valuable metrics like throughput and RTT.

Spindump is in fact especially viable due to its EFM support (including Spin Bit, Delay Bit and Q Bit) for the QUIC traffic, and it can be patched to perform the same analysis over a TCP connection.

6.2 Kernel patching

Kernel patching consists in modifying the TCP/IP stack in Linux, in order to force TCP connections to employ one EFM technique. The solution proposed hereafter uses the Kernel patch for L4S developed by L4STeam[15] as the base version for Spin Bit and Delay Bit, which does not mean that these are only available when using a L4S connection, and are in fact fully integrated for any TCP CC protocol. Appendixes E and F contain a detailed guide of the changes made to the Prague kernel source code, which is publicly available on Github at github.com/MatteoGuarna/linux_l4s_mod_for_passive_measurements.

Editing the kernel wasn't easy, and in fact, while integrating the Spin Bit didn't bring along many concerns, the Delay Bit raised a very important issue with EFM techniques which has been until now overlooked, and deserves to be discussed.

6.2.1 EFM and the topic of fragmentation

EFM techniques rely in fact on packet marking, which is performed in the transport layer by TCP, as it requires connection-level information. Still, fragmentation most of the times occurs at lower levels in the network stack, and might also happen along the network (for example due to MTU changes at data link layer). It's clear then that packets containing the TCP segments are broken down into smaller sizes, and crucially, their header is rewritten in each generated fragment. Depending on how the code is written, at least three things may happen:

- each header is built anew, losing the marking on the reserved bits
- the original header reserved bits are kept in the first fragment, and cleared in every other header
- the reserved bits are copied and kept in each fragment

The first issue is that according to the behaviour of the host performing the fragmentation different algorithms may behave differently (e.g., the Spin Bit requires the marking to be kept in each fragment, while the Delay Bit works only if the field is cleared in all fragments saved one - better if the the first; the Q bit may fail if fragmentation occurs anywhere on the network).

The silver lining is that most likely fragmentation happens on the endpoints, as the MTU on border link is most definitely lower than along the network: being it that a patch is already required to employ the chosen technique, the endpoints can therefore be edited to support whatever technique they are using where fragmentation occurs.

Another challenge lies in the point inside the network stack where fragmentation occurs: most likely packets are split between layer 3 and layer 2 (where the MTU is known), which is well below the layer 4, and that means that connection-layer data structures are unavailable, in accordance with the OSI model which states that each layer should be isolated and unaware of the other layers. Algorithms like the Q bit isn't as trivial to implement even just on the endpoints, as layer 4 should be able to control the number of consecutive marked packets, i.e., information should be passed vertically across multiple layers. Unfortunately due to this very issue more time would have been needed to adapt the Q bit to the Linux kernel architecture: in fact only the Spin Bit and Delay Bit were eventually deployed, while loss metrics were recovered on the endpoints via iperf3.

6.2.2 Fragmentation in L4S

As a side note, it's worth adding that dualpi2, as it was implemented in the Linux L4S Kernel, chooses to perform fragmentation by default when it receives packet with size greater than $1500 Bytes$, as "*Optionally, dualpi2 will split GSO skbs into independent skbs and enqueue each of those individually. This yields the following benefits, at the expense of CPU usage: Finer-grained AQM actions as the sub-packets of a burst no longer share the same fate (e.g., the random mark/drop probability is applied individually); Improved precision of the starvation protection/WRR scheduler at dequeue, as the size of the dequeued packets will be smaller.*"[17].

This creates an issue with EFM techniques if there are frames with size exceeding $1500 Bytes$ travelling through the network, as the queue surely is a point where fragmentation is bound to happen. The patch obviously ensures that fragmentation performed by any queue is compatible with the supported EFM technique.

6.3 Spindump patching

Spindump was chosen as tool for acquiring the metrics as it is already compatible with multiple EFM, including Spin Bit and Delay Bit. Still, the support was available only for UDP QUIC datagrams and a patch was needed to extend the support to TCP. It was basically sufficient to add some code to parse the TCP header and then provide the data to the functions already used for QUIC. The process was much easier and straight forward, but is still thoroughly detailed in appendix G. The patch is public and available on Github at github.com/MatteoGuarna/spindump_passive_measurements.

6.3.1 Aquiring data with Spindump

Spindump now is able to acquire throughput and latency measurements, which by default are shown in real time on the Linux shell. It's also possible though to retrieve a json file containing the data for each packet, and process it independently to extract connection-wide information: by doing that it was possible to evaluate not only the global RTT average, but also its standard deviation, as well as second-by-second average values, which are more useful to create reader-friendly plots. Appendix H contains the scripts that were used to parse the captures and perform the calculations.

Chapter 7

Testing the EFM techniques

7.1 Disclaimer

All experiments described from now on (where not specified differently) were run on both the physical and the logical test plant. As expected, all tests produced very similar results and while the numbers didn't match perfectly (as it can be expected due to the underlying hardware differences) no relevant difference was found. Thus, all data here reported were retrieved from the logical test plant, as they are easily replicable.

The experiments are divided in two sections: the current chapter aims at assessing the Delay Bit's validity and verify whether its advantages to the Spin Bit are in fact real and relevant; chapter 8 aims at analysing Prague's performance and adherence to the standard. Both sections rely on simultaneous $60s$ Prague and Cubic flows transmissions through the test plant as described in section 5 i.e., with a base RTT of 60ms, a bottleneck of $200Mbps$ to $100Mbps$, a $1500Bytes$ network MTU, and neither jitter nor reordering. This scenario is from now on called "base scenario", and all test stem from it changing one parameter at the time to analyse any change in the system's behaviour.

7.2 Comparing Spin Bit and Delay Bit

Spin Bit and Delay Bit are now both implemented and fully integrated, and are necessary to obtain reliable latency measurements on the test plant. Still, both techniques are identical in their objective: the first test aims thus at comparing the results obtained with both techniques to assess which one should be employed to test all all further scenarios.

7.2.1 Pros and cons of Spin Bit and Delay Bit

The Spin Bit was designed to obtain latency measurements, by sniffing traffic flowing through a specific point on the network. As a technique it is quite effective and there's plenty of academic evidence of its successful employment. Still, a well known major flaw is its vulnerability to ordering events, which may cause additional false wave fronts to be detected.

The Delay Bit was designed specifically to avoid this issue without introducing additional major flaws. The only weakness the Delay Bit may have against the Spin Bit come from the risk of losing the marked packet. Still, all our scenarios are designed to work with heavy congestion, where losses are not only plausible, yet voluntarily sought. The target of the next section is than that of assessing if the Delay Bit can hold up against the Spin Bit, and, if this is the case, if it can really outperform the Spin Bit when reordering is introduced.

7.3 The experiment

To attest the performance of both methods a simple test was run twice, retrieving the results once with the Delay Bit and once with the Spin Bit.

7.3.1 Base scenario

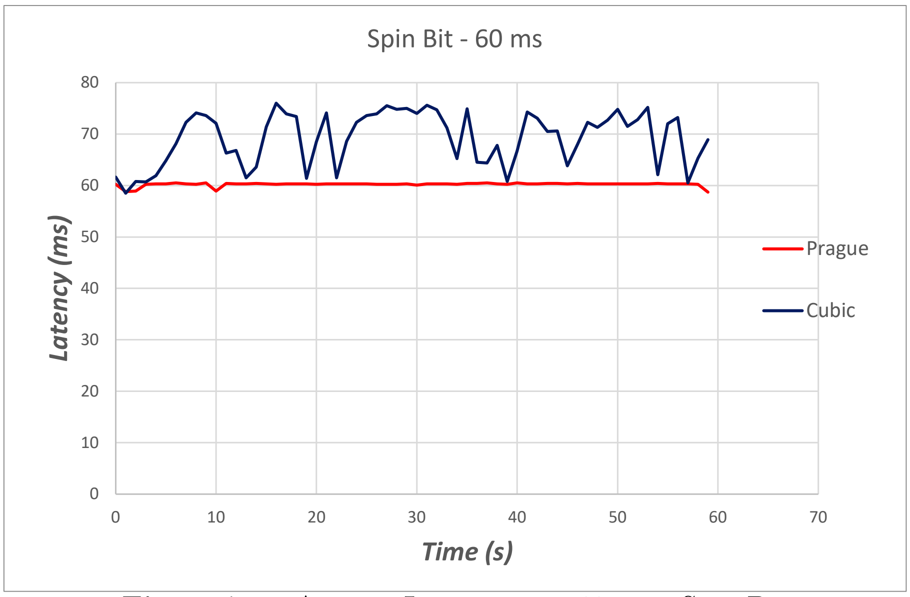


Figure 7.1: Average Latency over 60 ms - Spin Bit

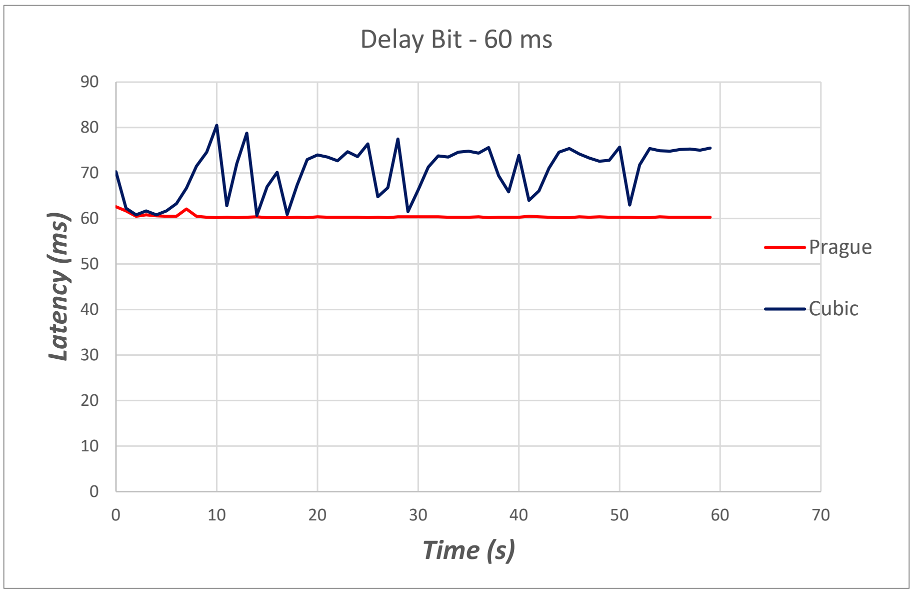


Figure 7.2: Average Latency over 60 ms - Delay Bit

Spin Bit and Delay Bit cannot be applied both simultaneously to the same flow, and therefore the only way to compare the two solutions is to run the experiment twice. For this reason the reader should not expect the same data and deem the difference as flaws of either methods. In fact, by comparing the global metrics acquired with either method (tab. 7.1) the throughput and losses look glaringly similar.

It is therefore meaningful to try and compare the latency values obtained via Spin and Delay Bit, by looking at both the global latency metrics and the trends throughout the capture (fig. 7.1 and 7.2): not only the results live up to the expectations derived from the system, but are also once again very comparable. It is thus safe to conclude that in the base scenario Spin and Delay Bit are synonymous: the losses are in fact very low even for Cubic and the Delay Bit measurements look virtually unaffected.

Base Scenario - Average results				
	Spin Bit		***Delay Bit***	
reordering	***Prague***	***Cubic***	***Prague***	***Cubic***
Throughput	*391 MB*	*317 MB*	*375 MB*	*331 MB*
Latency(avg.)	*60.3 ms*	***68.9 ms***	*60.4 ms*	***70.1 ms***
Latency(std.dev.)	*2.4 ms*	*7.2 ms*	*2.1 ms*	*9.0 ms*
Retransmissions	*0*	*11*	*0*	*15*

Table 7.1: Spin Bit and Delay Bit average results

7.3.2 Reordering impact over Spin Bit and Delay Bit

The next step consists in introducing reordering on the network. In most scenarios reordering on the network happens (besides when losses occur) due to latency variations and alternative routing. In order to replicate this phenomenon on a simple network, it's possible to virtually introduce reordering by running the netem command variant

```
    sudo tc qdisc add dev [network_interface] root netem delay [
   delay_value] reorder [reorder_percentage]
```

on the impairment. Reordering percentages introduced are very low as in real-world scenario reordering is never too pronounced, and in fact if even 1% reordering is introduced, most TCP protocol variant stop being able to occupy the available bandwidth and need huge retransmissions.

Average LATENCY (RTT 60 ms)						
	Socket Stats		***Spin Bit***		***Delay Bit***	
reordering	***Prague***	***Cubic***	***Prague***	***Cubic***	***Prague***	***Cubic***
0	*60.4 ms*	*70.4 ms*	*60.3 ms*	*68.9 ms*	*60.4 ms*	*70.1 ms*
0.001%	*60.5 ms*	*70.7 ms*	*37.6 ms*	*49.5 ms*	*60.4 ms*	*70.5 ms*
0.01%	*60.6 ms*	*65.3 ms*	*26.4 ms*	*44.4 ms*	*60.6 ms*	*65.2 ms*
0.1%	*61.1 ms*	*62.2 ms*	*20.2 ms*	*7.5 ms*	*62.3 ms*	*63.0 ms*

Table 7.2: Heatmap comparing the average RTT according to SB, DB, and SS

LATENCY standard deviation (RTT 60 ms)						
	Socket Stats		***Spin Bit***		***Delay Bit***	
reordering	***Prague***	***Cubic***	***Prague***	***Cubic***	***Prague***	***Cubic***
0	*0.0 ms*	*5.8 ms*	*2.4 ms*	*7.2 ms*	*2.1 ms*	*9.0 ms*
0.001%	*0.3 ms*	*5.8ms*	*23.3 ms*	*30.1 ms*	*2.4 ms*	*9.9 ms*
0.01%	*0.8 ms*	*6.1 ms*	*22.4 ms*	*26.4 ms*	*3.4 ms*	*10.1 ms*
0.1%	*1.0 ms*	*4.0 ms*	*21.0 ms*	*7.7 ms*	*5.6 ms*	*10.9 ms*

Table 7.3: Heatmap comparing the RTT std. dev. according to SB, DB, and SS

After repeating the experiment with several reordering rates (0.001%, 0.01%, 0.1%), it's sufficient to take a look at the global measurements 7.2 to see that the results provided by the Spin Bit are not realistic: as the theory suggests, it's fair to assume that the Spin Bit is rendered useless by even the lowest reordering rate. Concerning the Delay Bit instead, its results are plausible, but need to be compared to those obtained through an already established method to be validated. Therefore, the same tests was performed relying on socket statistics, a CLI command that returns on-hosts measurements, and is thus unaffected by reordering. It's easy to see that the average results are very close, and the standard deviation (tab. 7.3) is acceptable, although higher. These results hence deem the Delay Bit a perfectly viable instrument to obtain in-network end-to-end latency.

7.3.3 Delay Bit against losses

MTU - Average results		
Retransmissions	***Prague***	***Cubic***
0	*0*	*15*
0.001%	*0*	*14*
0.01%	*276*	*17*
0.1%	*511*	*436*

Table 7.4: Prague and Cubic losses

It is now clear that the Delay Bit is immune to reordering: as a matter of fact, it was designed with this precise purpose. Still, having only one marked packet travelling through the network at any given moment, makes this technique theoretically vulnerable to packet loss. The Delay Bit's saving grace is that, at least for TCP, losses occur on the network very sparsely, and a stable, properly functioning connection has losses in a still very small percentage compared to the actually delivered packets. As a matter of fact, the losses the flows experienced in the previous experiment are very high (tab. 7.4), but did not significantly affect the results, further validating the Delay Bit's viability.

Chapter 8

Testing L4S

8.1 Evaluating L4S' performance

The aim of this chapter is that of analyzing L4S' behaviour in the envisioned testing ground. The following experiments try to examine L4S' characteristics by varying one of the test plant's parameters. All measurements are obtained through the Delay Bit.

8.1.1 Base scenario

It's best to start from the base scenario, which was specifically designed in order to showcase L4S' essence when deployed in a congested network.

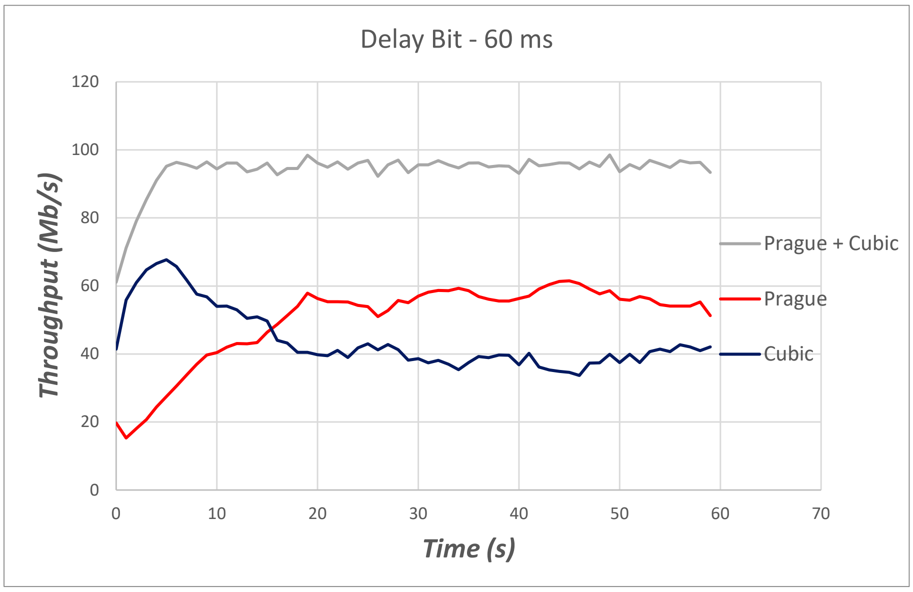


Figure 8.1: Bandwidth share - Prague and Cubic

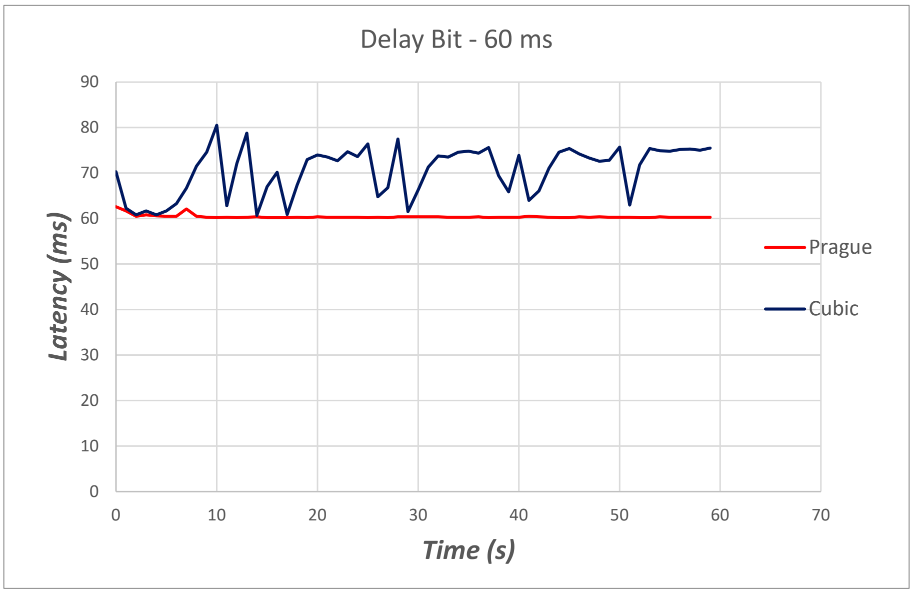


Figure 8.2: Latency comparison - Prague and Cubic

Figure 8.1 provides a connection-wide representation of the bandwidth share via throughput measurements: Prague's flow grows much slower than Cubic, and surpasses it almost exactly 15 seconds in during the test, then the average values remain more or less stable with a 60-40 ratio. Figure 8.2 presents an overview of the RTT variation throughout the experiments, and clearly shows Prague as consistently adherent to the 60 seconds base network latency. Cubic on the other hand, as soon as the overall throughput reaches the channel's capacity (around 8 seconds in) starts rising and oscillating widely.

The average results' (tab. 8.1) reflect these observations, showing a much lower standard deviation in Prague's latency. Furthermore, as L4S' name itself declares, no losses occur for Prague, while the same cannot be said for Cubic.

Base Scenario - Average results		
	Prague	***Cubic***
Throughput	*375 MB*	*331 MB*
Latency (avg.)	*60.4 ms*	*70.1 ms*
Latency (std. dev.)	*2.1 ms*	*9.0 ms*
Retransmissions	*0*	*15*

Table 8.1: Base Scenario average results

All gathered information are in line with L4S' objectives and show that in a controlled situation the architecture behaves exactly as expected. The next steps consist in observing the changes occurring when some test plant parameters are altered.

8.2 Introducing reordering

As already partially detailed in 7.3, reordering can be added to the test plant to see how L4S behaves. Likewise, the introduced reordering ratio remains low to simulate a plausible network phenomenon.

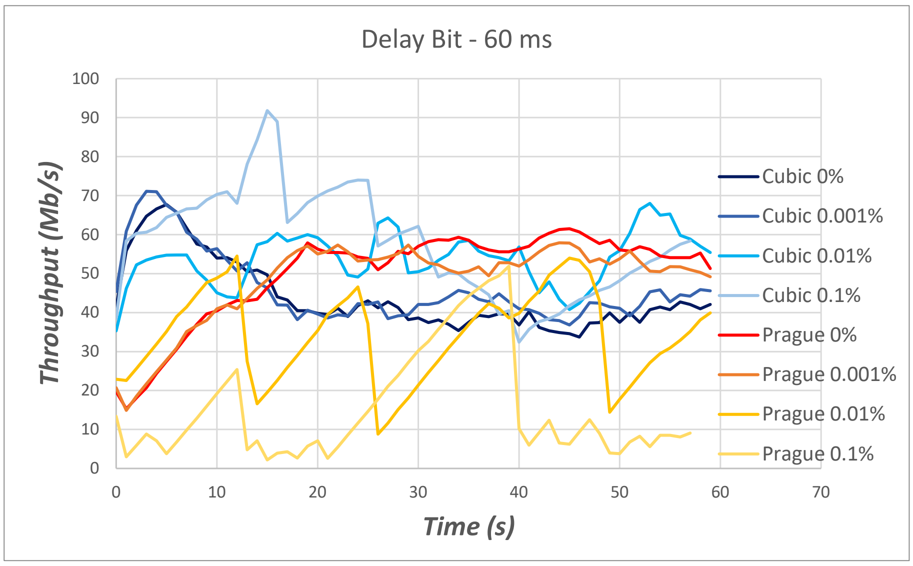


Figure 8.3: Bandwidth share with multiple reordering ratios

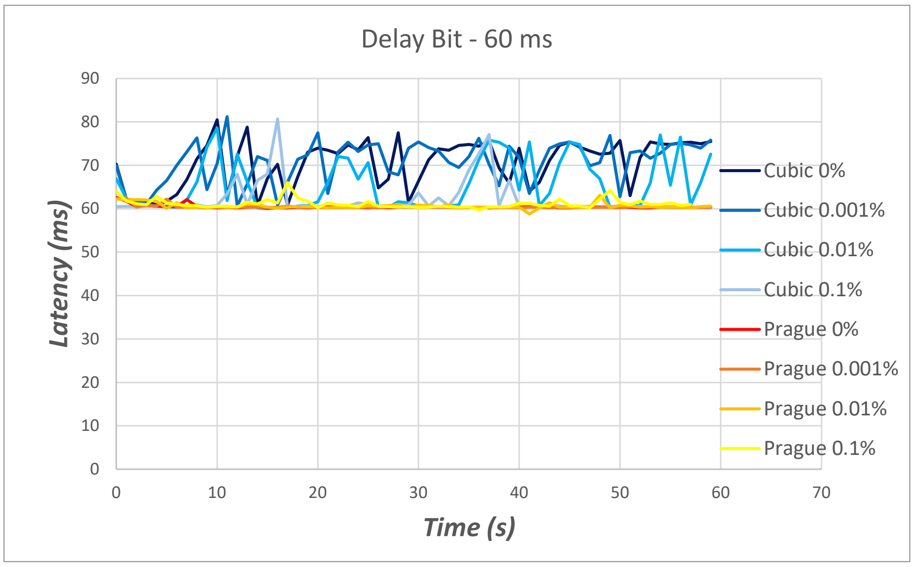


Figure 8.4: Latency comparison with multiple reordering ratios

It becomes evident that when high reordering is introduced (0.01% or more)

Prague stops working properly: while in fact the delay remains low (more on that later) the bandwidth share drops significantly. The reason for the latter can be seen in tab 8.2, where very high losses are shown for Prague.

Reordering - Average results						
	Throughput		Latency		Retransmissions	
Reordering	***Prague***	***Cubic***	***Prague***	***Cubic***	***Prague***	***Cubic***
0	*375 MB*	*331 MB*	*60.4 ms*	*70.1 ms*	*0*	*15*
0.001%	*361 MB*	*347 MB*	*60.4 ms*	*70.5 ms*	*0*	*14*
0.01%	*256 MB*	*405 MB*	*60.6 ms*	*65.2 ms*	*276*	*17*
0.1%	*111 MB*	*419 MB*	*62.3 ms*	*63.0 ms*	*511*	*436*

Table 8.2: Reordering scenario average results

8.2.1 Reduced Prague throughput scalability

As Prague does not expect losses to normally occur, as they arise they are interpreted as a "hard congestion event", as opposed to the "soft congestion event" consisting in a simple CE marking notification performed by the dual queue. Therefore, if losses occur, Prague's rate drops to zero, as it can be clearly seen in the graphs. Cubic, on the other hand, doesn't drastically reduce its CWND when a packet is lost, and is therefore able to maintain a higher throughput, even when losses are comparably high (i..e., with 0.1% reordering rate).

A second clearly-visible issue consists in Prague's throughput increase: it becomes in fact evident that the growth is linear, while a scalable TCP CC algorithm should be able to promptly increase its CWND and swiftly occupy the available bandwidth. This can be also observed at the beginning of the connection, and even in the base scenario (graph. 8.1). The reasons for this peculiar behaviour are contained in the latest ITEF draft for Prague, where the following is stated:

"*So, in the Linux implementation of Prague, the Reduced RTT-Dependence algorithm only comes into effect after D rounds, where D is configurable (current default* 500*). Continuing the previous example, if actual rtt* $= 5ms$ *and rtt_virt* $= 25ms$*, Prague would use the regular RTT-dependent algorithm for the first* $500 * 5ms = 2.5s$*. Then it would start to converge to more equal rates using its Reduced RTT-Dependence algorithm. If the actual RTT were higher (e.g.* $20ms$*), it would stay in the regular RTT-dependent mode for longer* ($500rounds = 10s$)*, but this would be mitigated by the actual RTT it uses at the start being closer to the virtual RTT that it eventually uses* ($20ms$ *and* $25ms$ *resp.).*"[18]

This means that when a connection starts, in order to ensure that Prague remains Reno-friendly, a AIMD algorithm is used for the first 500 round-trips ($30ms$ in our experiment). This means that the rise is linear, as in fact it can be observed. Likewise, the same behaviour was also kept when losses occur."[19]

8.2.2 Poor Prague performance with manually introduced reordering

Still, there are two more observations that can be raised. The former is that too many retransmissions occur (tab. 8.2), and that cannot be explained by a supposed high latency addition caused by the reordering, as if that were the case Cubic would have been affected at least equally if not more. The latter is that the Cubic's RTT decreases together with Prague's flow drop, as the congestion softens with the decreased rate, almost converging with Prague's RTT when 0.1% reordering is introduced, with the latter even rising slightly. Both these behaviours can be partially ascribed to the fact that Prague itself should limit the reordering naturally occurring on the network, and isn't therefore as wired as Cubic to handle a high rate of artificially introduced reordering. Furthermore, L4S' implementation is still experimental in nature, and isn't reported ready for commercial use, which may help explaining some issues with its performance.

8.3 Introducing multiple base latencies

In this use case the base network latency is changed, in order to find how Prague's and Cubic's behaviours are influenced by the RTT.

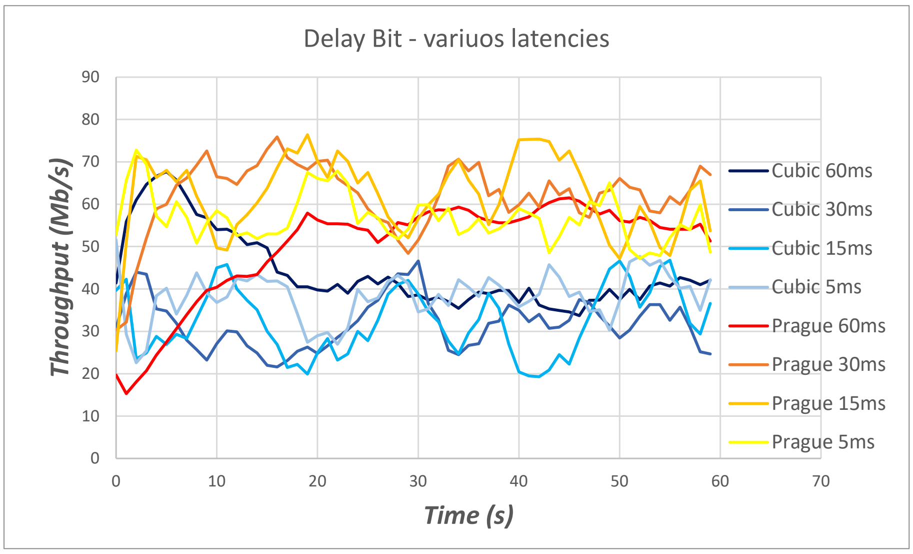


Figure 8.5: Bandwidth share with multiple base latencies

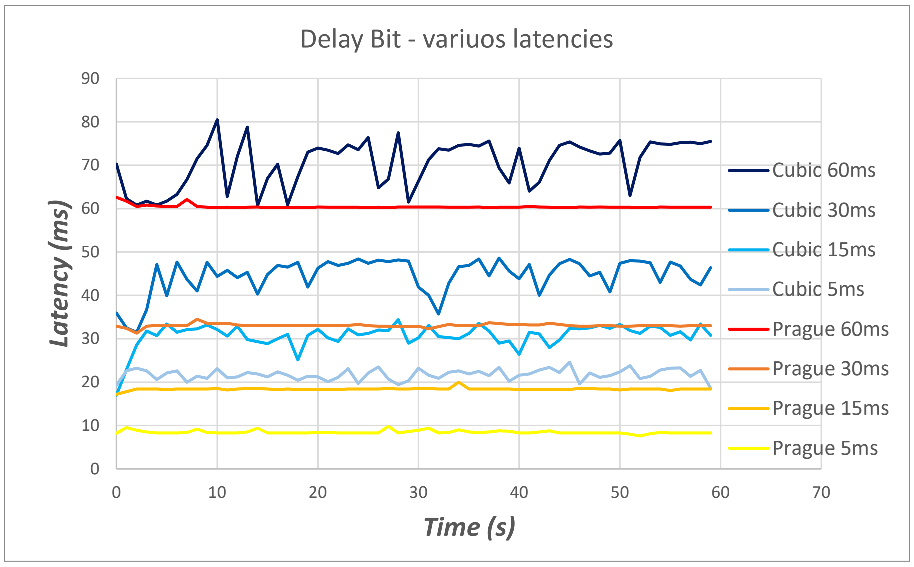


Figure 8.6: Latency comparison with multiple base latencies

8.3.1 Base latency adherence

Starting by the latencies graph, its easy to see that for low base latencies Cubic's RTT is much further above than the minimum RTT. Throughput's share too is reportedly lower, as it can be clearly seen in 8.3. Prague on the other hand shows an average RTT which is very close to the network base latency in every test, which highlight Prague's capability to perform well independently from the network's characteristics.

Multiple latencies - Average results						
	Throughput		Latency		Retransmissions	
Reordering	***Prague***	***Cubic***	***Prague***	***Cubic***	***Prague***	***Cubic***
0	*375 MB*	*331 MB*	*60.4 ms*	*70.1 ms*	*0*	*15*
0.001%	*466 MB*	*238 MB*	*31.0 ms*	*42.4 ms*	*0*	*29*
0.01%	*466 MB*	*243 MB*	*16.4 ms*	*28.4 ms*	*0*	*35*
0.1%	*426 MB*	*287 MB*	*6.4 ms*	*19.7 ms*	*0*	*59*

Table 8.3: Latencies scenario average results

8.3.2 Prague's additive increase

The main observation inherited from the previous experiment concerns L4S' reduced scalability. The current scenario provides a more in-depth look on this behaviour, as the additive increase algorithm is RTT-dependant, which means that for a smaller RTT the CWND (and thus the throughput) growth should be faster.

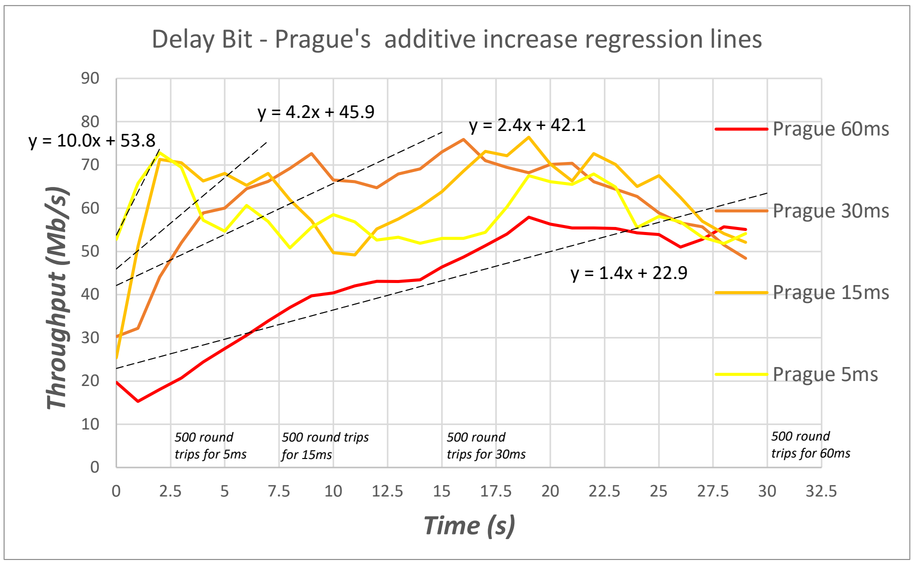


Figure 8.7: Regression line over the first 500 round trips

Indeed, by evaluating the linear regression over the first 500 round trips for each trial (fig. 8.7), it's fairly easy to spot a relation between between the base latency and the slope of the line. The obtained relationship (fig. 8.8) is an almost perfect inverted correlation (as the theory suggests) despite the presence of another flow competing for the available bandwidth. It is also equally evident that as soon as the first 500 round trips are over the scalable algorithm kicks in, which improves Prague's aim to an equal bandwidth share between the flows.

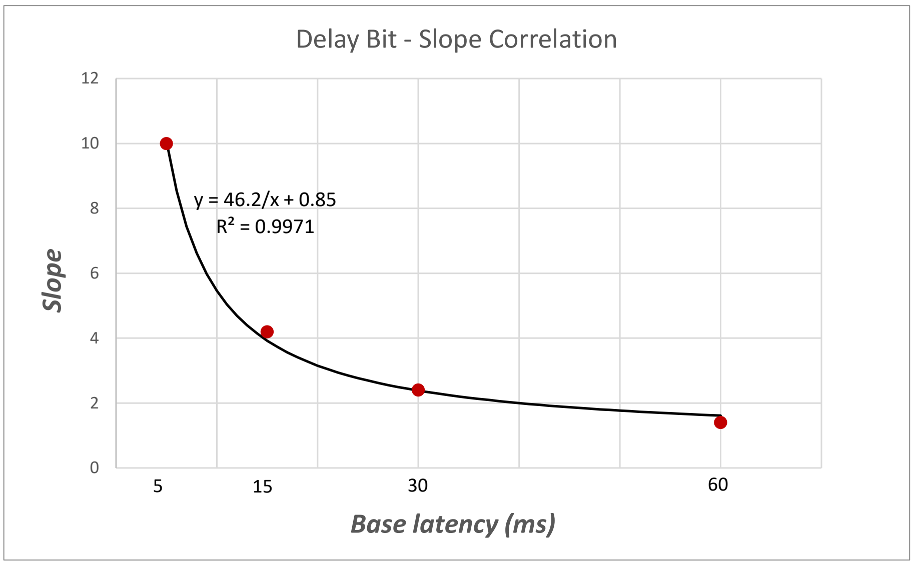


Figure 8.8: Base latency - slope correlation

All retrieved data are thus in full accord with the theoretical description of Prague, and highlight its robustness compared to the classic TCP protocols.

8.4 Applying some jitter

Testing Prague for various base latencies is certainly useful, but it's very common to have a variable latency, due to many factors bound to the networks' own complexity. The idea is therefore that of introducing jitter virtually on the router through netem:

```
 sudo tc qdisc add dev [network_interface] root netem rate 1000
mbit delay [delay_value] [jitter_value]
```

The experiment was conducted starting from the base scenario and introducing progressively more jitter, slowing the flowing packets of an amount varying from zero up until the maximum jitter value. The commands are defined in order to avoid introducing any reordering, as the base netem jitter command (without the rate parameter) works by applying a different delay to each packet, which may often result in later packets being delayed for less then their predecessor. The rate parameter is therefore necessary in order to avoid that from happening[20][21], thus introducing a sort of "spring effect" over the segments, which are randomly delayed but keep their original sequencing.

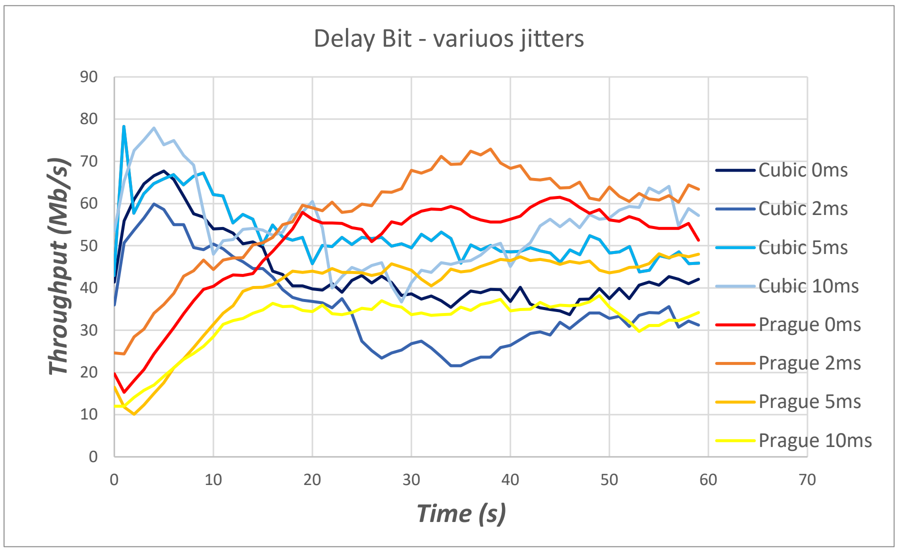


Figure 8.9: Bandwidth share with multiple jitter values

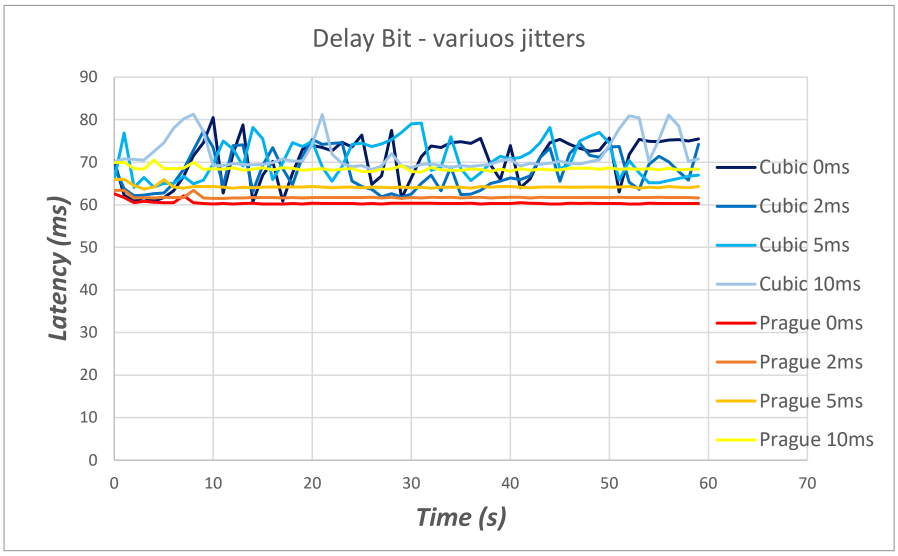


Figure 8.10: Latency comparison with multiple jitter values

Starting with figure 8.10, it's worth noting that all latency graph are not traced using the single values measured for each individual packet, but the averages for each seconds of transmission, in order to be able to discern the trend more easily. Then, despite the jitter, the average RTT values for each second for Prague remain stable, while Cubic, as always, varies considerably. Obviously, Prague's values are higher the more jitter is introduced, as the average by each second is evaluated over packets delayed by various amounts within the current maximum value.

Jitter - Average results						
	Throughput		Latency		Retransmissions	
Jitter	***Prague***	***Cubic***	***Prague***	***Cubic***	***Prague***	***Cubic***
0 ms	*375 MB*	*331 MB*	*60.4 ms*	*70.1 ms*	*0*	*15*
2 ms	*430 MB*	*269 MB*	*31.0 ms*	*67.8 ms*	*0*	*12*
5 ms	*298 MB*	*396 MB*	*16.4 ms*	*70.4 ms*	*0*	*15*
10 ms	*239 MB*	*410 MB*	*6.4 ms*	*71.6 ms*	*0*	*8*

Table 8.4: Jitter scenario average results

Still, it's easy to see that while the Cubic latency values never fall below Prague, the global averages come closer for high jitter values (tab. 8.4): this can be explained by looking at the throughput (figure 8.9): in fact, in order to keep its latency low and stable, Prague's throughput is reduced when high jitter is introduced, which lessen the severity of the congestion, which allows Cubic's average RTT to come closer to Prague As a further proof for this phenomenon, Cubic's retransmissions also decrease highlighting that the congestion is lessened. Still, this should not be attributed to a supposed Cubic's ingrained resistance to network jitter, but solely to Prague's conservative rate management.

8.5 Changing the MTU

Lastly, it may be interesting to check how the network behaves with different MTUs. Iperf3 allows to specify the application layer MTU size, ranging from a minimum of $88 Bytes$ to a maximum of $1460 Bytes$ (actually higher values are possible but not useful in the current test plant as offloading is disabled and network interfaces fragment packets over $1500 Bytes$). The application layer size isn't comprehensive of the TCP and IP header, which add $20 Bytes$ each, pushing the highest MTU to the maximum Ethernet payload (i.e., $1500 Bytes$).

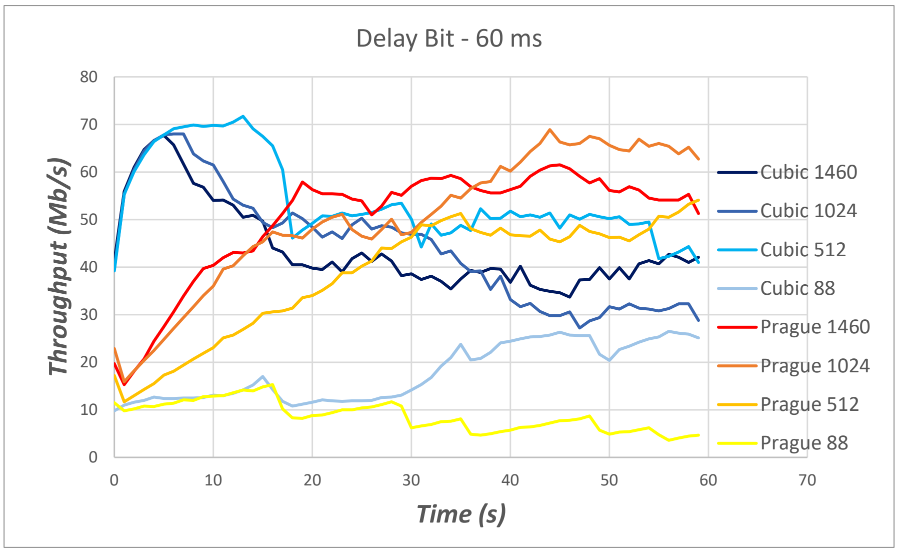


Figure 8.11: Bandwidth share with multiple MTU values

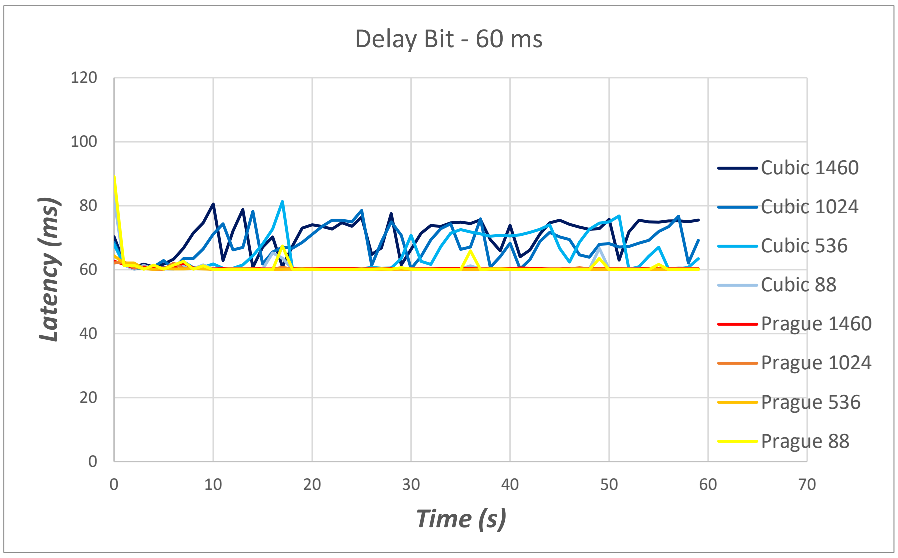


Figure 8.12: Latency comparison with multiple MTU values

For the sake of clarity, it's worth reiterating that all results will be referenced in terms of the application layer MTU.

MTU - Average results						
	Throughput		Latency		Retransmissions	
L7 MTU	***Prague***	***Cubic***	***Prague***	***Cubic***	***Prague***	***Cubic***
1460 Bytes	*375 MB*	*331 MB*	*60.4 ms*	*70.1 ms*	*0*	*15*
1024 Bytes	*377 MB*	*333 MB*	*60.3 ms*	*67.4 ms*	*0*	*28*
512 Bytes	*288 MB*	*405 MB*	*60.2 ms*	*64.8 ms*	*8*	*29*
88 Bytes	*65 MB*	*134 MB*	*60.8 ms*	*60.7 ms*	*1595*	*430*

Table 8.5: MTU scenario average results

The results show that MTU size can affect greatly both Prague's and Cubic's performance. While 1024*Bytes* of MTU have very little impact on the protocols other than decreasing Cubic's latency, as its delay is mainly due to the queues on the bottleneck, and smaller packets create smaller queues, the situation changes for smaller frames: with 512*Bytes* of MTU the throughput share varies drastically, with a sharp decline of Prague in favour of Cubic. Still, with 88*Bytes* MTU Cubic's throughput too drops, even if remains double that of Prague.

8.5.1 MTU-throughput connection

The throughput variation isn't linked to Prague's or Cubic's behaviours per se, and rather by the CWND algorithm: let's in fact consider that the CWND, and therefore the throughput, is increased with each Round Trip. Still, if the MTU is very low, it will take much more time to fill the available bandwidth, as the same number of packets sent on the network for any round trips (which is controlled by the CWND) will carry much less data than they would with a higher MTU (fig. 8.13).

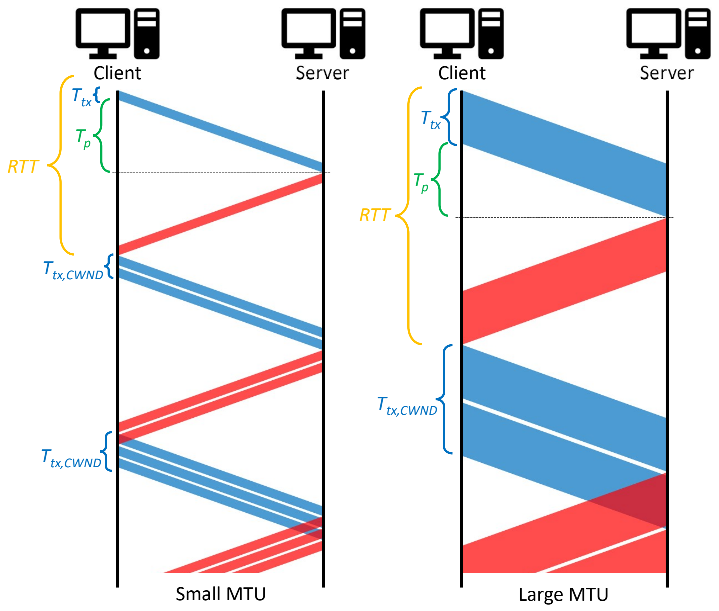


Figure 8.13: Small and Large MTU throughput comparison

In order to reach a stable situation where the MTU does not influence the throughput any longer the total transmission time for the window must be equal to the RTT, i.e., double the sum of the transmission and network propagation times (eq. 8.1): when that happens, the transmission has reached the interface's maximum speed.

$$RTT = 2(T_{tx} + T_{tx,propagation}) = 2 \cdot v_{transmission} \cdot MTU + 2 \cdot T_{tx,propagation} \tag{8.1}$$

The window's transmission time is obviously proportional to the window size in Bytes (eq. 8.2), which depends though on the MTU: this translates for smaller MTUs in both a slower growth and a later reach (if any) of the channel's speed.

$$T_{tx,CWND} = CWND \cdot T_{tx} = CWND \cdot v_{transmission} \cdot MTU \tag{8.2}$$

On a side note, obviously the RTT is dependent on the MTU too, but on a much smaller level than the CWND's transmission time: this is why for larger MTUs the channel's speed is reached earlier in the latter case.

The phenomenon can seen clearly for Prague in figure 8.14, as the protocol has a linear growth (AIMD) which allows the reader to better appreciate the slower rise the smaller the MTU.

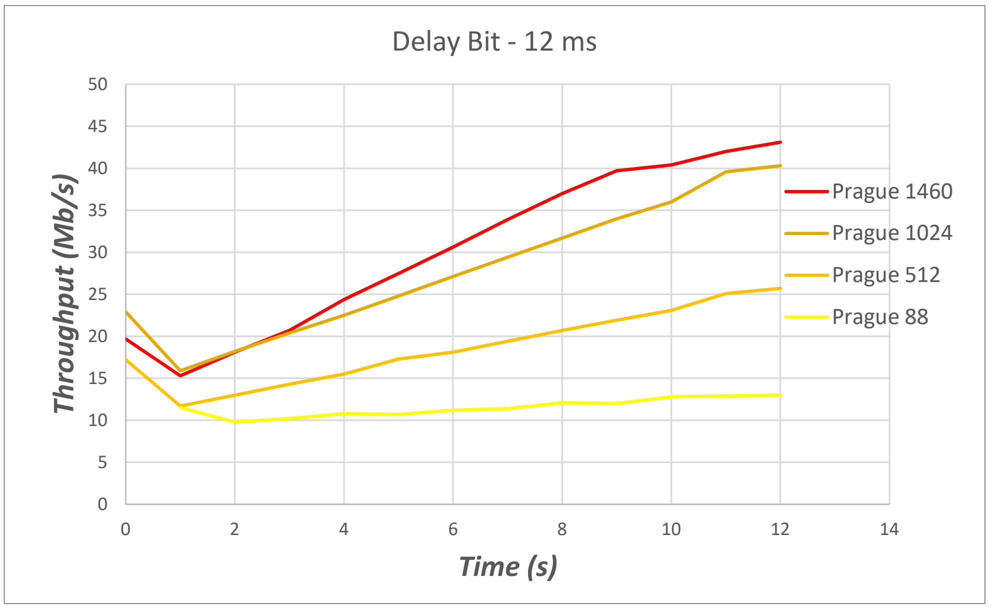


Figure 8.14: Prague throughput increase for different MTUs

Cubic, as the name suggests, takes much less to grow to the channel's capacity, and therefore the behaviour cannot be appreciated for plausible MTUs, and in fact, as soon as Prague slows its rise Cubic is able to obtain a larger share of the channel.

Still, it is evident that for 88*Bytes* of MTU Cubic too struggles to increase its RTT). It's hard to point out a reason for this behaviour, but it's worth noting that MTUs that small are not only unrealistic, but in fact practically non-existent in real applications; furthermore, as the averages heatmap shows (tab. 8.5) the retransmissions are really high for both Prague and Cubic, which is a contributing factor in the protocols' poor performance. The test suggests that both protocols were not created to work in such conditions, which are again, very unrealistic; furthermore, even Prague's abysmal performance can be overlooked if even Cubic performs similarly poorly.

Chapter 9

Conclusions

9.1 Research's Goals

The main goal of this research, i.e., deploying a working implementation of some Host-to-Network EFM techniques, and assess their viability, can be deemed reached. What's especially relevant is the improvement obtained with the Delay Bit compared to the much more popular Spin Bit: hopefully this work will be instrumental in contributing to a more wide spread adoption of this powerful solution.

The EFM techniques' implementation developed as a part of this work is furthermore very useful as it is the first time they were employed over TCP/IP: this achievement showcases the solutions' validity in a wider array of scenarios than only within QUIC, as it was originally conceived. As the TCP/IP stack is in fact the most commonly used network solution by far, a working implementation on the kernel linux, as well as a linux-based application capable of extracting the metrics (i.e., the Spindump patch herein developed) make up an accessible solution for testing and employing both the Spin Bit and the Delay Bit. Hopefully, future works will help spreading and improve over this work.

Another relevant result consists in the deployment of a fully-operating L4S environment, which can be (at least in its virtual form) easily replicated as a demo of this new and promising technology. The results obtained with the EMF techniques are for the most part exactly in line with the protocol's description, and lead to the expected improvements: this research will thus hopefully help the L4S project gaining more momentum.

9.2 Future works

Future developments over this research consist surely in the implementation of the Spin Bit and Delay Bit on a vanilla version of the Linux kernel, and possibly a contribution to the Linux foundation.

Another topic worth investigating is the implementation of the Q Bit, and loss measurement EFM techniques more broadly, which did not find space in this work due to time constraints, which did not allow for a more through analysis of the fragmentation process in TCP/IP and its relation to the techniques here reported.

Lastly, testing both EFM-based solutions and the L4S architecture over more complex and realistic network environments is needed to evaluate more thoroughly how these technologies (both together and on their own) behave over the internet. EFM techniques especially beg for this study, as their compatibility with devices which are not designed to support them is due.

Appendix A

L4s Deployment

Deploying a functioning Linux kernel supporting L4S is actually fairly easy, provided you already have all the pieces of the puzzle. This appendix is a step-by-step guide to ensure you can at least deploy a virtual environment where you can replicate the experiments which were run in this thesis.

A.1 Creating a dual boot Linux partition from Windows 11

The L4S patch I used is only available in Linux. This section gives you some tips regarding how to create a working Linux environment inside a Windows 11 computer if you don't already have a machine equipped with Linux. If you do, feel free to skip to the next section.

There are many online guides explaining how to create a working Linux partition. Personally I chose this guide[22] for it's easy to understand and guides you with most of the process.

One big issue you might find when trying to install the ISO form USB is that no memory partition is recognized. This happens because many Windows-native PC have a BitLocker function that encrypts the main memory to avoid data to be copied and read by anyone besides the user. This prevents the installer to properly recognize the created partition and thus to write in it. To solve this issue the only solution is to disable the BitLocker encryption from the system settings BEFORE creating the partition.

A.2 Installing the L4S Kernel

Once you have a Linux machine running, you can rely on the Installation (Debian derivatives)" section of the README.md inside the linux L4STeam repository[15]. You might need to disable memtest86 in order to complete the installation process successfully.

```
sudo apt purge memtest86+
sudo update-grub
```

A.3 Switching to a new Kernel

You can easily check which kernel version is currently running by executing from terminal:

```
uname -r
```

If you are using a dual-booted system, usually on boot the GRUB menu is loaded, which allows you to choose among the different kernels you have installed. If the following error messages appear:

```
Bad shim signature
you need to load the kernel first
```

you just need to disable the UEFI secure boot to be able to load the OS with a different kernel.

If you cannot or do not want to use the GRUB menu you can follow the procedure belows[23][24], or alternatively use grub-customizer[25]

- Find the correct kernel index inside the OS by using this command from terminal:

```
cat /boot/grub/grub.cfg | grep -iE "menuentry 'Ubuntu, with Linux
" | awk '{print i++ " : "$1, $2, $3, $4, $5, $6, $7}'
```

- Edit the GRUB setup file (found at this source) with a text editor...

```
sudo nano /etc/default/grub
```

- ...by changing the line GRUB_DEFAULT=[something, probably 0] to:

```
GRUB_DEFAULT="1>[the actual kernel index]"
```

- then you can update the grub and reboot the machine

```
sudo update-grub
sudo reboot
```

Appendix B

Virtual Test Plant Configuration

The whole configuration relies on virtual network elements like virtual bridges and the virtual links. The routing function at the heart of the configuration relies on the forwarding capability between network interfaces inside the same namespace, which can be enabled as a system-wide setting through:

```
    sudo sysctl -w net.ipv4.ip_forward=1
```

B.1 Adding the L4S protocols

Most importantly, the dual queue AQM must be enabled on both the router's interfaces

```
    sudo tc qdisc replace dev [network_interface] root dualpi2
```

The network namespaces are then set up creating totally separate environments where network settings can be independently defined. This allows us to create multiple local network with their own routing without triggering conflicts. The Prague endpoints are then configured with Prague as CC protocol with the following command:

```
    sudo sysctl -w net.ipv4.tcp_congestion_control=prague
```

B.2 Further specifications

At this point the basic protocols of L4S are in place, but still some attention must be reserved for some settings which are required by the protocols or the setup to work properly.

B.2.1 Network settings

First of all, being all links virtual, the speed must be reduced in order to create a bottleneck from a well-known throughput to a smaller one. The basic tool for that is usually ethtool, but due to it not working properly with virtual interface the best solution is inserting a queuing discipline with traffic control:

```
 sudo tc qdisc add dev [network_interface] root tbf rate 100mbit
latency 1ms burst 1540
```

This raises the issue of allowing more disciplines to coexists on the same network interface. The solution to this issue is to rely on hooks and can be seen in the final script.

Furthermore, as in specified 3.2.3 some delay needs to be introduced to mirror L4S' working condition, due to the virtual network obviously having really low RTT (less than $2ms$). Delays were therefore introduced via traffic control on the bridges' NICs through the command:

```
 sudo tc qdisc add dev [network_interface] root netem delay 30ms
```

We settled on $30ms$) on each LAN because $60ms$) is a very plausible delay for most real internet applications.

B.2.2 Additional settings to allow L4S to work properly

Lastly, we need to take into account L4S' specifications. First of all, the GitHub README.md file specifies that their Prague implementation is designed to work with fair queue, which then needs to be enabled also on the Cubic endpoints to allow the comparison to be as fair as possible:

```
 sudo tc qdisc add dev [network_interface] root fq limit 20480
flow_limit 10240
```

Equally, offloading capabilities are disabled everywhere to mirror the "standard behaviour" of the network, with L2 frames not exceeding the channel's MTU. Section 6.2.1 describes more in detail how MTU variations and fragmentation may impact on the EFM techniques; still, the kernel used is designed to perform the fragmentation correctly and therefore is also compatible with the offloading enabled, as detailed in sec. 6.2.2.

```
 sudo ethtool -K [network_interface] tso off gso off gro off lro
off
```

B.2.3 Virtual Configuration Script

the following script contains all settings hereby discussed and can be launched on any Linux machine patched with the Prague-dualpi2 kernel provided by L4Steam.

Please consider that these are only the default settings for the experiment and some tests require modifications to be done to the scripts: all changes are thoroughly reported alongside the tests definitions.

```
#!/bin/bash
#This script aims at creating a testbed for testing L4S traffic

function cleanup {
    set +e

    # Delete all the veth created in the script
    sudo ip link del veth1br
    sudo ip link del veth2br
    sudo ip link del veth3br
    sudo ip link del veth4br
    sudo ip link del veth5rt
    sudo ip link del veth6rt
    sudo ip link del br10
    sudo ip link del br20

    # Delete all the namespaces you create in the script
    sudo ip netns del ns1
    sudo ip netns del ns2
    sudo ip netns del ns3
    sudo ip netns del ns4

     # Restore the root ns configuration
    sudo sysctl net.ipv4.tcp_congestion_control=cubic
    sudo sysctl net.ipv4.tcp_ecn=2
}
trap cleanup EXIT

set -e

```

```
echo "------------------------------------------------------------------"
echo "                    execution started                             "
echo "------------------------------------------------------------------"

#ROOT NS
#enable ip forwarding to turn the ns into a router
sudo sysctl -w net.ipv4.ip_forward=1
forward=$(sudo sysctl net.ipv4.ip_forward | cut -f 3 -d ' ')
if [ $forward -ne 1 ]; then
    echo "unable to execute 'sudo sysctl -w net.ipv4.ip_forward=1'"
    exit
fi
echo "status of ipv4 forwarding parameter successfully enabled"

#add a link (veth5) towards br1 ns
sudo ip link add veth5rt type veth peer name veth5br
#turn the veth5rt up
sudo ip link set dev veth5rt up
#add ip address 10.10.10.254/24 to the veth5rt
sudo ip addr add 10.10.10.254/24 dev veth5rt

#add a link (veth6) towards br2 ns
sudo ip link add veth6rt type veth peer name veth6br
#turn the veth6rt up
sudo ip link set dev veth6rt up
#add ip address 10.10.20.254/24 to the veth6rt
sudo ip addr add 10.10.20.254/24 dev veth6rt


# NAMESPACE NS1
sudo ip netns add ns1
echo "namespace ns1 created"
#add a link (veth1) towards br1 ns
sudo ip link add veth1br type veth peer name veth1ns
#move the veth veth1ns to the ns1
sudo ip link set veth1ns netns ns1
#turn the veth1ns up
sudo ip netns exec ns1 ip link set dev veth1ns up
#add ip address 10.10.10.1/24 to the veth1ns + route towards bridge
sudo ip netns exec ns1 ip addr add 10.10.10.1/24 dev veth1ns
sudo ip netns exec ns1 ip route add default via 10.10.10.254 dev
    veth1ns
echo "ip address 10.10.10.1/24 assigned to ns1"


# NAMESPACE NS2
sudo ip netns add ns2
echo "namespace ns2 created"
#add a link (veth2) towards br1 ns
```

```
sudo ip link add veth2br type veth peer name veth2ns
#move the veth veth2ns to the ns2
sudo ip link set veth2ns netns ns2
#turn the veth2ns up
sudo ip netns exec ns2 ip link set dev veth2ns up
#add ip address 10.10.10.2/24 to the veth2ns + route towards bridge
sudo ip netns exec ns2 ip addr add 10.10.10.2/24 dev veth2ns
sudo ip netns exec ns2 ip route add default via 10.10.10.254 dev
    veth2ns
echo "ip address 10.10.10.2/24 assigned to ns2"

# NAMESPACE NS3
sudo ip netns add ns3
echo "namespace ns3 created"
#add a link (veth3) towards br2 ns
sudo ip link add veth3br type veth peer name veth3ns
#move the veth veth3ns to the ns3
sudo ip link set veth3ns netns ns3
#turn the veth3ns up
sudo ip netns exec ns3 ip link set dev veth3ns up
#add ip address 10.10.20.3/24 to the veth3ns + route towards root ns
sudo ip netns exec ns3 ip addr add 10.10.20.3/24 dev veth3ns
sudo ip netns exec ns3 ip route add default via 10.10.20.254 dev
    veth3ns
echo "ip address 10.10.20.3/24 assigned to ns3"

# NAMESPACE NS4
sudo ip netns add ns4
echo "namespace ns4 created"
#add a link (veth4) towards br2 ns
sudo ip link add veth4br type veth peer name veth4ns
#move the veth veth4ns to the ns4
sudo ip link set veth4ns netns ns4
#turn the veth4ns up
sudo ip netns exec ns4 ip link set dev veth4ns up
#add ip address 10.10.20.4/24 to the veth4ns + route towards root ns
sudo ip netns exec ns4 ip addr add 10.10.20.4/24 dev veth4ns
sudo ip netns exec ns4 ip route add default via 10.10.20.254 dev
    veth4ns
echo "ip address 10.10.20.4/24 assigned to ns4"


# BRIDGE 10
sudo ip link add br10 type bridge
sudo ip link set dev br10 up

sudo ip link set dev veth1br up
sudo ip link set dev veth2br up
sudo ip link set dev veth5br up

sudo ip link set veth1br master br10
sudo ip link set veth2br master br10
sudo ip link set veth5br master br10


# BRIDGE 20
sudo ip link add br20 type bridge
sudo ip link set dev br20 up

sudo ip link set dev veth3br up
sudo ip link set dev veth4br up
sudo ip link set dev veth6br up

sudo ip link set veth3br master br20
sudo ip link set veth4br master br20
sudo ip link set veth6br master br20



#Configure rootNS with prague and dualpi2 (to execute after all nss
    are created otherwise properties are inherited)
sudo tc qdisc add dev veth5rt root handle 1: tbf rate 100mbit latency
     1ms burst 1540
sudo tc qdisc add dev veth6rt root handle 1: tbf rate 1000mbit
    latency 1ms burst 1540
echo "bottleneck configured"
sudo tc qdisc add dev veth5rt parent 1: dualpi2
sudo tc qdisc add dev veth6rt parent 1: dualpi2
echo "dualpi2 configured"

#add delay on the bridges
sudo tc qdisc add dev veth5br root handle 1: netem delay 30ms
sudo tc qdisc add dev veth6br root handle 1: netem delay 30ms
echo "netem configured"
sudo tc qdisc add dev veth5br parent 1: pfifo limit 1000
sudo tc qdisc add dev veth6br parent 1: pfifo limit 1000
echo "pfifo configured"

#Configure ns1 with prague and dualpi2
sudo ip netns exec ns1 sudo sysctl net.ipv4.tcp_congestion_control=
    prague
sudo ip netns exec ns1 sudo sysctl -w net.ipv4.tcp_ecn=3

#Configure ns3 with prague and dualpi2
sudo ip netns exec ns3 sudo sysctl net.ipv4.tcp_congestion_control=
    prague
sudo ip netns exec ns3 sudo sysctl -w net.ipv4.tcp_ecn=3

```

```
#Ensure correct ns2 config is kept
sudo ip netns exec ns2 sudo sysctl net.ipv4.tcp_congestion_control=
    cubic
sudo ip netns exec ns2 sudo sysctl net.ipv4.tcp_ecn=2

#Ensure correct ns4 config is kept
sudo ip netns exec ns4 sudo sysctl net.ipv4.tcp_congestion_control=
    cubic
sudo ip netns exec ns4 sudo sysctl net.ipv4.tcp_ecn=2

#Disable offloading capabilities everywhere on the network (most
    importantly on the router's interfaces)
sudo ip netns exec ns1 sudo ethtool -K veth1ns tso off gso off gro
    off lro off
sudo ip netns exec ns2 sudo ethtool -K veth2ns tso off gso off gro
    off lro off
sudo ip netns exec ns3 sudo ethtool -K veth3ns tso off gso off gro
    off lro off
sudo ip netns exec ns4 sudo ethtool -K veth4ns tso off gso off gro
    off lro off
sudo ethtool -K veth1br tso off gso off gro off lro off
sudo ethtool -K veth2br tso off gso off gro off lro off
sudo ethtool -K veth3br tso off gso off gro off lro off
sudo ethtool -K veth4br tso off gso off gro off lro off
sudo ethtool -K veth5br tso off gso off gro off lro off
sudo ethtool -K veth6br tso off gso off gro off lro off
sudo ethtool -K veth5rt tso off gso off gro off lro off
sudo ethtool -K veth6rt tso off gso off gro off lro off

#Configure the fq, limit bandwith on all interfaces generating
    traffic
sudo ip netns exec ns1 sudo tc qdisc add dev veth1ns root handle 1:
    tbf rate 100mbit latency 50ms burst 1540
sudo ip netns exec ns2 sudo tc qdisc add dev veth2ns root handle 1:
    tbf rate 100mbit latency 50ms burst 1540
sudo ip netns exec ns3 sudo tc qdisc add dev veth3ns root handle 1:
    tbf rate 100mbit latency 50ms burst 1540
sudo ip netns exec ns4 sudo tc qdisc add dev veth4ns root handle 1:
    tbf rate 100mbit latency 50ms burst 1540
sudo ip netns exec ns1 sudo tc qdisc add dev veth1ns parent 1:0 fq
    limit 20480 flow_limit 10240
sudo ip netns exec ns2 sudo tc qdisc add dev veth2ns parent 1:0 fq
    limit 20480 flow_limit 10240
sudo ip netns exec ns3 sudo tc qdisc add dev veth3ns parent 1:0 fq
    limit 20480 flow_limit 10240
sudo ip netns exec ns4 sudo tc qdisc add dev veth4ns parent 1:0 fq
    limit 20480 flow_limit 10240

echo "-----------------------------------------------------------------"
```

```

read -p "Press ENTER to delete current configuration..."
```

Appendix C

Physical Test Plant Configuration

The configuration of the physical test plant follows a very similar trail to that taken in the virtual scenario discussed in appendix B. The followings paragraph are dedicated to add some details to how the configuration described in section 5.3.

C.1 The endpoints

The lack of a sufficient number of machines at my disposal pushed me to rely solely on two server to implement all endpoints (Arrakis and Harkonnen). This was possible thanks to the fact that both servers had many physical network interfaces which could be used to establish several simultaneous flows from different sources on each machine. Still, Linux doesn't allows to configure the TCP congestion control protocol only as a global function, while each interface should support a different protocol (Prague or Cubic). For this reason, the network namespaces were once again employed to create a distinct network environment from the main one, so that both configurations could exist. This time though the actual physical NICs were brought inside the namespaces in order to keep the configuration as close to the real scenario as possible, by eliminating virtual network elements.

```
    sudo ip netns add [namespace name]
    sudo ip link set [network interface] netns [namespace name]
```

I decided to keep Prague inside the default namespace and deploy Cubic in the newly create environment due to fears that the still experimental implementation of Prague might misbehave. Still, once the test plant was configured I repeated my tests with Prague inside the namespaces and noticed no difference at all. I can therefore affirm with confidence that namespaces do not impact the network performance of either protocol.

I created a script to set up the endpoints, which can be applied on both endpoints. The general requirements for the interface setup are explained in appendix B:

```
#!/bin/bash
#This script allows to move the $NIC2 inside a Linux namespace, in
    order to have it working with the TLS CUBIC cubic flavour instead
    of PRAGUE, which is the defaut on this machine.
#Furthermore, it disables the explicit marking (ECN) of the traffic
    sent through the NIC, and sets the speed of both $NIC1 and $NIC2
    to 100M.

NIC1=nic1_name
NIC2=nic2_name

function cleanup {
        set +e

    #kill iperf3 in both namespaces
    sudo pkill iperf3

    #remove netem config
        sudo tc qdisc del dev $NIC1 root

    # Delete all the namespaces created in the script
        sudo ip netns del ns1

        #Restore the default speed inside the root environment
        sudo ethtool -s $NIC1 speed 1000 duplex full autoneg on
    #Restore offloading
    sudo ethtool -K $NIC1 tso on gso on gro on lro on

        #Keep Prague as default CC protocol
        sudo sysctl net.ipv4.tcp_congestion_control=prague

    echo "default configuration restored"
    echo ""

}


trap cleanup EXIT

set -e

```

```
clear

echo "-------------------------------------------------------------"
echo "                    execution started                        "
echo "-------------------------------------------------------------"

#Set Prague as CC protocol
echo "Default namespace:"
sudo modprobe sch_cake
sudo sysctl net.ipv4.tcp_congestion_control=prague
sudo sysctl net.ipv4.tcp_ecn=3
echo ""

#change the advertised link speed inside the root environment to 100M
sudo ethtool -s $NIC1 speed 100 duplex full autoneg on
#disable all offloading capabilities on the l4s nic
sudo ethtool -K $NIC1 tso off gso off gro off lro off


#create a new network namespace
sudo ip netns add ns1
#move enp129s0f3 inside the namespace
sudo ip link set $NIC2 netns ns1
#turn the nic up
sudo ip netns exec ns1 ip link set dev $NIC2 up
#give back the network address to the interface
sudo ip netns exec ns1 ip addr add 192.168.201.18/24 dev $NIC2
#configure the route towards the bridge
sudo ip netns exec ns1 ip route add default via 192.168.201.19 dev $NIC2

#set the TCP flavour and the ECN
echo "Namespace NS1:"
sudo ip netns exec ns1 sudo sysctl net.ipv4.tcp_congestion_control=cubic
sudo ip netns exec ns1 sudo sysctl net.ipv4.tcp_ecn=2
echo ""

#change the advertised link speed to 100M
sudo ip netns exec ns1 ethtool -s $NIC2 speed 100 duplex full autoneg on
#disable offloading on cubic too for equity
sudo ip netns exec ns1 sudo ethtool -K $NIC2 tso off gso off gro off lro off


#introduce the delay
#sudo tc qdisc add dev $NIC1 root netem delay 15ms
#sudo ip netns exec ns1 tc qdisc add dev $NIC2 root netem delay 15ms
```

```


#set the fq queue on both NICs
sudo tc qdisc add dev $NIC1 root handle 1: tbf rate 100mbit latency
    50ms burst 1540
sudo ip netns exec ns1 sudo tc qdisc add dev $NIC2 root handle 1: tbf
     rate 100mbit latency 50ms burst 1540
sudo tc qdisc replace dev $NIC1 parent 1: fq limit 20480 flow_limit
    10240
sudo ip netns exec ns1 tc qdisc replace dev $NIC2 parent 1: fq limit
    20480 flow_limit 10240


echo "——————————————————————————————————————————————————"

echo ""

read -p "Press ENTER to delete current configuration..."
```

C.2 The router

The router is where most of the trouble came from. In theory it would have been sufficient to enable the Linux forwarding mode to allow the machine to behave as a router, and then set up dualpi2 on both interfaces. Still, when trying to test the achieved environment I observed that the Prague traffic was being starved by the Cubic traffic. I spent a whole month trying to isolate the issue but all I could make out was that dualpi2 wasn't behaving as expected, and I even managed to discuss it with some people behind design and implementation of the L4S architecture[26], which were very kind and willing to help, and allowed me to have a clearer picture of the issue itself. Still, any actual solution seemed unfathomable.

As a matter of fact the idea of deploying a virtual scenario came as one of many tests to try and get a grasp on the issue. In fact it was thanks to the virtual scenario that I found the solution: due to the fact that it's not possible to set the network speed via ethtool on virtual interfaces I fell back on tbf as additional queuing discipline to restrict the traffic (which otherwise would have been unconstrained and wouldn't have allowed me to create the bottleneck). I then could attest that the virtual scenario was in fact working, and being tbf the only difference in the router configuration I migrated this modification on the real scenario (though without constraining the NICs' throughput). Doing that I could finally make the physical scenario work.

My conclusion is than that in order to correctly configure dualpi2, tbf has to also be added on the same interface. It is possible to use handles to deploy multiple disciplines on the same interface, provided they fill in different roles (e.g., its not possible to use two queues like dualpi2 and fq on the same time):

```
    sudo tc qdisc add dev [network interface] root handle 1: tbf rate
     1000mbps latency 10ms burst 1540
    sudo tc qdisc add dev [network interface] parent 1:0 dualpi2
```

The script with all router's settings is included below:

```
#!/bin/bash
#This script is useful in order to properly configure the NICs queue
    to dualpi2

NIC1=nic1_name
NIC2=nic2_name

function cleanup {
    set +e

    pkill spindump

     # Restore the root ns configuration
    sudo tc qdisc delete dev $NIC1 root
    sudo tc qdisc delete dev $NIC2 root
    sudo ip addr del 192.168.201.19/24 dev $NIC1
    sudo ip addr del 192.168.202.20/24 dev $NIC2

    sudo ethtool -s $NIC2 autoneg on speed 1000 duplex full
    sudo ethtool -s $NIC1 autoneg on speed 1000 duplex full
}
trap cleanup EXIT

set -e

echo "--------------------------------------------------------------------"
echo "                    execution started                               "
echo "--------------------------------------------------------------------"

sudo clear

sudo ip addr add 192.168.201.19/24 dev $NIC1
echo "$NIC1 configured with ip address 192.168.201.19"
sudo ip addr add 192.168.202.20/24 dev $NIC2
echo "$NIC2 configured with ip address 192.168.202.20"
sudo sysctl net.ipv4.ip_forward=1
```

```


#sudo tc qdisc add dev $NIC1 root dualpi2
#sudo tc qdisc add dev $NIC2 root dualpi2
sudo tc qdisc add dev $NIC1 root handle 1: tbf rate 1000mbit latency
    100ms burst 1540 #latency 10ms burst 1540
sudo tc qdisc add dev $NIC2 root handle 1: tbf rate 1000mbit latency
    100ms burst 1540 #latency 10ms burst 1540
echo "tbf added"
sudo tc qdisc add dev $NIC1 parent 1: dualpi2
sudo tc qdisc add dev $NIC2 parent 1: dualpi2
echo "dualpi2 configured"
sudo ethtool -s $NIC2 autoneg on speed 100 duplex full
sudo ethtool -s $NIC1 autoneg on speed 1000 duplex full
echo "bottleneck 1000>100 configured"
sudo ethtool -K $NIC1 tso off gro off gso off lro off
sudo ethtool -K $NIC2 tso off gro off gso off lro off
echo "offloading disabled"

spindump --interface $NIC2 2>/dev/null &

read "anything"
```

C.3 The network

The network setup relies on Ethernet links: in order to have multiple flows converging on the same router, I decided to use a L2 switch. Obviously, this requirement is valid on both the server and the client side, and therefore two switches are needed. Having just one switch at my disposal in the lab, I used L2 VLANs to separate the flows on either side of the router.

The endpoint interfaces are set to $100Mbps$ and while the server side interface supports $1Gbps$ of traffic to accommodate for all the incoming data and avoid a bottleneck on the switch, the client side interface is set to $100Mbps$, thus creating a bottleneck.

Latency (and jitter too in some experiments) need to be introduced to simulate a real network: Caladan was deployed to mimic a simple L2 switch in order to be transparent to the rest of the network, saved for the delay introduced with netem on each interface. The script used for its configuration is here reported:

```
#!/bin/bash
```

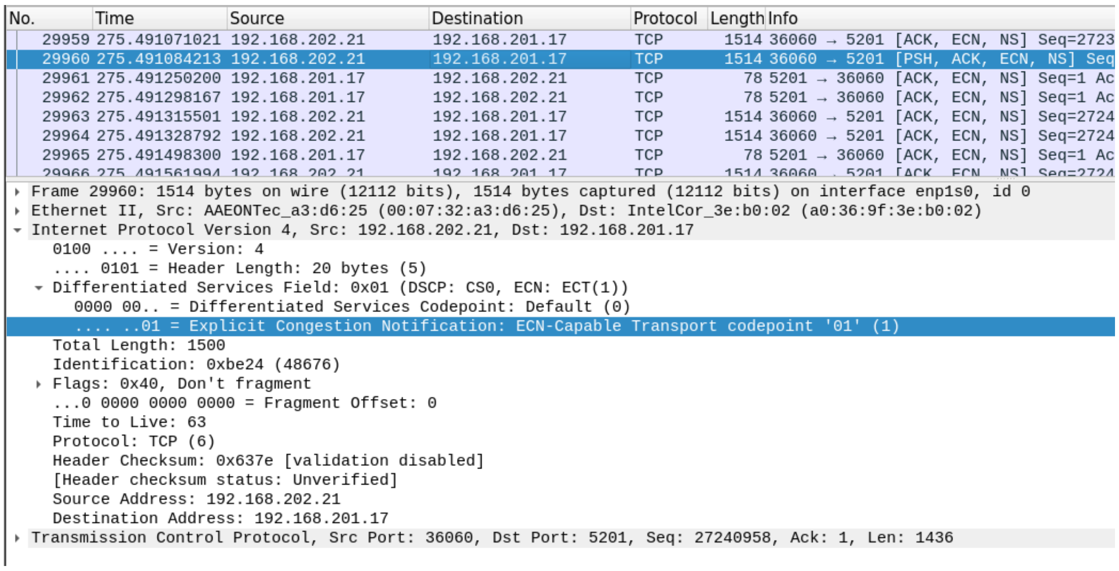
```
#This script aims at configuring the impairment to emulate real-life
    network conditions, for testing L4S traffic

NIC1=nic1_name
NIC2=nic2_name

function cleanup {

    sudo ip link del vbridge
    sudo tc qdisc delete dev $NIC1 root
    sudo tc qdisc delete dev $NIC2 root
}
trap cleanup EXIT

set -e

sudo clear

echo "----------------------------------------------------------------"
echo "                    execution started                           "
echo "----------------------------------------------------------------"

sudo ip link add vbridge type bridge
sudo ip link set dev vbridge up
sudo ip link set $NIC1 master vbridge
sudo ip link set $NIC2 master vbridge
sudo tc qdisc add dev $NIC1 root handle 1: netem delay 30ms
sudo tc qdisc add dev $NIC2 root handle 1: netem delay 30ms
sudo tc qdisc add dev $NIC1 parent 1: pfifo limit 1000
sudo tc qdisc add dev $NIC2 parent 1: pfifo limit 1000


echo ""
echo "bridge functions enabled"
echo "added 30ms delay on each direction"
echo ""

echo "----------------------------------------------------------------"

read -p "Press ENTER to delete current configuration..."
```

Appendix D

Validating the Test Plants

D.1 Exploring the packets

In this section the correctness of the test plant is discussed. The first thing to check was the correct marking of the IP header, which allows the Qual Queue algorithm to treat the flows accordingly (see section 2.4.1). To do that the packet sniffing tool Wireshark was used, and it was also very extensively used later for further testing. Data was generated via iperf3, which is useful as it also shows the throughput for each connection.



```
No.   Time             Source           Destination      Protocol Length Info
29959 275.491071021 192.168.202.21 192.168.201.17   TCP    1514 36060 → 5201 [ACK, ECN, NS] Seq=2723
29960 275.491084213 192.168.202.21 192.168.201.17   TCP    1514 36060 → 5201 [PSH, ACK, ECN, NS] Seq
29961 275.491250200 192.168.201.17 192.168.202.21   TCP      78 5201 → 36060 [ACK, ECN, NS] Seq=1 Ac
29962 275.491298167 192.168.201.17 192.168.202.21   TCP      78 5201 → 36060 [ACK, ECN, NS] Seq=1 Ac
29963 275.491315501 192.168.202.21 192.168.201.17   TCP    1514 36060 → 5201 [ACK, ECN, NS] Seq=2724
29964 275.491328792 192.168.202.21 192.168.201.17   TCP    1514 36060 → 5201 [ACK, ECN, NS] Seq=2724
29965 275.491498300 192.168.201.17 192.168.202.21   TCP      78 5201 → 36060 [ACK, ECN, NS] Seq=1 Ac
29966 275.491561994 192.168.202.21 192.168.201.17   TCP    1514 36060 → 5201 [ACK, ECN, NS] Seq=2724

▸ Frame 29960: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits) on interface enp1s0, id 0
▸ Ethernet II, Src: AAEONTec_a3:d6:25 (00:07:32:a3:d6:25), Dst: IntelCor_3e:b0:02 (a0:36:9f:3e:b0:02)
▾ Internet Protocol Version 4, Src: 192.168.202.21, Dst: 192.168.201.17
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▾ Differentiated Services Field: 0x01 (DSCP: CS0, ECN: ECT(1))
      0000 00.. = Differentiated Services Codepoint: Default (0)
      .... ..01 = Explicit Congestion Notification: ECN-Capable Transport codepoint '01' (1)
    Total Length: 1500
    Identification: 0xbe24 (48676)
  ▸ Flags: 0x40, Don't fragment
    ...0 0000 0000 0000 = Fragment Offset: 0
    Time to Live: 63
    Protocol: TCP (6)
    Header Checksum: 0x637e [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 192.168.202.21
    Destination Address: 192.168.201.17
▸ Transmission Control Protocol, Src Port: 36060, Dst Port: 5201, Seq: 27240958, Ack: 1, Len: 1436
```

Figure D.1: ECN marking is ECT(1), in line with Prague's specification

The capture shown in figure D.1 was performed on a single Prague flow and clearly shows that packets carry the ECT(1) value (01) inside the ECN field, and means that the queue is able to recognize and handle the flow.

By simultaneously enabling both a Prague and a Cubic flow a congestion is created, and in fact the iperf3 bitrate measurement show that the sum of the flows is slightly less than $100Mbps$ (as it's to be expected due to the header overhead). By looking at new Wireshark captures a number of CE-marked Prague packets is found (only after the router and towards the receiver), which means that the Queue is notifying the endpoints of the congestion.

As far as the information that a visual examination can provide, all involved protocols are working correctly. The next step is checking whether L4S' behaviour in congestion reflects its declared characteristics.

D.2 Taking a look at the connections metrics

Iperf3 offers end-to-end loss metrics and allows to easily attest that in fact Prague flows have consistently 0 losses during 60 seconds parallel flow transmissions, while several occur with Cubic. Furthermore we can observe the end-to-end RTT via this command (to execute on each server before the transmission begins):

```
(while true; do date; ss -tenmoi; sleep 1; done) > /root/ss.txt
```

The results show lower and more consistent RTT values in the Prague connection compared to Cubic, which is once again in line with all expectations.

D.2.1 The issue with Dualpi2

The real issue is though probably in the most obvious and most accessible measurement: Cubic's throughput is consistently four times higher than Prague's, so much that we might consider Cubic to be by all means "starving" Prague. The source of the problem is evidently dualpi2, as delay, jitter, and losses are still lower for Prague, which is a sign of the endpooints' good behaviour; furthermore, if the Dual Queue is replaced by a normal queue, delay, jitter and losses are comparable with Cubic, but also the throughput: all these clues seem to suggest that dualpi2 is favouring the classic flow over L4S.

Trying to solve this issue was long and complex, and is more thoroughly discussed in C.2. Still, it looks like daulpi2's implementation as made available by the L4STeam works only properly when paired with tbf (any reasonable parameter should work):

```
sudo tc qdisc add dev [network interface] root handle 1: tbf rate
1000mbps latency 10ms burst 1540
sudo tc qdisc add dev [network interface] parent 1:0 dualpi2
```

This expedient allows to have a much more even ratio between the flows: all experiments in chapters 7 and 7 rely in fact on tbf and provide the reader with sufficient data to attest its viability.

Appendix E

Kernel Modifications for Spin Bit support

E.1 Downloading, compiling and installing a new kernel

As anticipated in chapter 6, the Spin Bit and Delay Bit implementation were both developed upon L4STeam's Kernel patch for L4S[15]. The github project's README.md contains a guide on how to download and compile the source code. I am here providing a more detailed step-by-step walk-through of the whole procedure.

E.1.1 Downloading the source code

The first step consists in downloading the source from Github: this can be done either through the website in .zip format, or via command line through the git command, which allowed me (–depth parameter) to retain only the last version of the repository, without the project's history:

```
    git clone --depth 1 https://github.com/L4STeam/linux.git
```

I then created my own repository (which is now too publicly available on Github[27]) by following this guide[28]

E.1.2 Compiling and installing the new kernel

In order to recompile the kernel after any modification the first had to inherit the kernel configuration I had previously obtained by installing the L4S kernel on the machines, by copying it in the .config file in the same folder containing the source code (this is quite important, do not make a new config as suggested as an alternative in the README for it is a very stripped down configuration over which the built kernel doesn't work). Then I compiled according to that configuration:

```
    cp "/boot/config-$(uname -r)" .config
    sudo make olddefconfig
```

In order to complete the compilation process with success the following modules are required:

```
    sudo apt install build-essential     # basic tool to compile c/c
   ++ software
    sudo apt install flex
    sudo apt install bison
    sudo apt install libelf-dev
    sudo apt-get install libssl-dev
```

Then the actual compilation can begin (it might take more than an hour the first time, while following modifications take a variable time according to the edited files' dependencies.

```
    sudo make -j$(nproc) LOCALVERSION=-new-kernel-name
    sudo make modules_install
    sudo make install
    sudo update-grub
```

After the installation is completed the instructions detailed in section A.3 are to follow in order to test the new kernel version.

E.1.3 Clearing old kernel versions

To remove old kernels it's sufficient to remove the old image and modules:

```
    sudo cd /boot
    sudo rm *-new-kernel-name*
    sudo cd /lib/modules
    sudo rm  -r *-new-kernel-name*
```

Alternatively, this guide[29] can be also used.

E.2 where the marking takes place

EFM rely on packet marking to convey information and make available of an in-network observer, and need therefore a dedicated field inside the packet, possibly on TCP header as these techniques operate in the connection layer.

TCP works very well for that, as there are three reserved bits inside the header which do not carry any information. Since our target is that of implementing multiple marking techniques, it is best to define a way to advertise which method is going to be used at the beginning of the connection, to allow the in-network observer to tell the algorithm in action (otherwise it has no way to understand, let's say, if the Spin Bit or the Delay Bit is being used).

As the original objective of the research was that of implementing Spin Bit, Delay Bit and Q Bit, the choice I made was the following:

- the 5th bit of the 13th Byte of the TCP header was renamed "loss bit"
- the 56h bit of the 13th Byte of the TCP header was renamed "time bit"

The idea was that of using the time bit to carry either the Spin Bit or the Delay Bit marking, and using the loss bit to carry the Q bit. The algorithms were codified inside the SYN and SYNACK packets to allow the observer to know which technique would be used. Considering both the 5th and 6th bit together, in the handshake the segments were marked as follows:

- 00 $\rightarrow$ no EFM
- 01 $\rightarrow$ Delay Bit
- 10 $\rightarrow$ Spin Bit
- 11 $\rightarrow$ Delay Bit + Q Bit

E.3 Editing the source code

The workflow for the Spin Bit implementation is the following:

- Identifying the structures hosting the connection-level TCP parameters, and adding to them the flags necessary to check the current state of the algorithm on each end of the connection
- Finding a way to define the roles of each endpoint (the Spin Bit requires the two hosts to adopt two different behaviours, and thus there must be a way to assign a different role to each of them)
- Finding the functions responsible for the parsing of the TCP header, allowing them to extract the time and loss fields, and save the value in the connection-level variables
- Finding the function responsible for the construction of the TCP header of the outgoing segment, and marking it according to both the role and the current state of the algorithm

E.3.1 Identifying the connection-level structures

The TCP header flags are stored inside the `struct sock_common`, defined at line 164 of `include/net/sock.h` as part of the `struct sock`. This is a good place to define the variables to host the information required by the algorithm to operate as it is unique for each connection, and must always be attainable. In fact, since these varaibles are to be updated according to the states of TCP, it is reasonable to suppose that the `struct sock_common` will be always available in the functions where said variables will be read or written.

Let's therefore define the two emuns we'll need to save the internal states of the algorithm and place them in the file. Basically the only information we need to store are the role (as each of the endpoints must behave either by reflecting the received value, or by inverting it), and the internal value according to which the segments must be marked. Therefore, I added at line 164 (above `struct sock_common`) the following enums:

```
/*SPIN BIT impl: required to define the role for the algorithm*/
enum spin_role {
    SPIN_ROLE_CLIENT,         //sets the spin_value to the opposite of
    the
//last received value
    SPIN_ROLE_SERVER,         //sets the spin_value to the same value
    as the
};                      //last received one

/*SPIN BIT impl: required to carry the value of the algorithm*/
enum spin_value {
    SPIN_BIT_DOWN,
    SPIN_BIT_UP,
};
```

Then, the actual variables need to be defined inside the struct (added at line 198):

```
/*SPIN BIT impl: required to define the role for the algorithm*/
enum spin_role         __skc_spin_role;
/*SPIN BIT impl: required to carry the value of the algorithm*/
enum spin_value        __skc_spin_value;
```

Lastly, in order to gain access more easily to the variables from the parent struct (`struct sock`), it's useful to define some macros, as it's done for all variables contained in `struct sock_common` the aliases (line 411);

```
/*SPIN BIT impl: defining macro to access values defined inside the
    struct sock_common*/
#define sk_spin_value         __sk_common.__skc_spin_value
```

```
#define sk_spin_role          __sk_common.__skc_spin_role
```

E.3.2 Defining the roles

In order to define the roles the first idea that came to my mind was that of looking for a connection-level variables that might hold whether or not the TCP connection was started by that host or by its counterpart. Still, unfortunately, no field inside the `struct sock` turned out to be useful, hence I had to come up with a work-around.

There is a function inside `net/ipv4/tcp.c` called `tcp_set_state` that manages the state changes of the TCP connection: the idea consists of using the state transitions towards "connection established" and looking at the previous state: if the previous state was "syn sent" then the host was the connection initiator, and we can assign to it the role of the client, otherwise, if the old state was "syn received", the host will fill in as server (line 2576):

```
/*SPIN BIT impl: assign role for a conversation*/
if (oldstate == TCP_SYN_SENT && state == TCP_ESTABLISHED) {
    /*this is the server of the connection*/
    sk->sk_spin_role = SPIN_ROLE_CLIENT;
    sk->sk_spin_value = SPIN_BIT_DOWN;
}
else if (oldstate == TCP_SYN_RECV && state == TCP_ESTABLISHED) {
    /*this is the server of the connection*/
    sk->sk_spin_role = SPIN_ROLE_SERVER;
    sk->sk_spin_value = SPIN_BIT_DOWN;
}
```

E.3.3 Parsing the incoming packets

This is the part that required more code analysis: fist of all there's a structure called `struct tcphdr` inside `include/uapi/linux` (line 25) which is used inside various TCP functions to cast a pointer to the buffer containing the whole frame, with the aim of extracting the TCP header fields. There I edited the "reserved" field and added the space for the time and loss bits:

```
struct tcphdr {
    __be16  source;
    __be16  dest;
    __be32  seq;
    __be32  ack_seq;
#if defined(__LITTLE_ENDIAN_BITFIELD)
```

```
    __u16   ae:1,
        /*SPIN BIT impl*/
        time: 1,
        loss: 1,
        res1:1,
        doff:4,
        fin:1,
        syn:1,
        rst:1,
        psh:1,
        ack:1,
        urg:1,
        ece:1,
        cwr:1;
#elif defined(__BIG_ENDIAN_BITFIELD)
    __u16   doff:4,
        res1:1,
        /*SPIN BIT impl*/
        loss:1,
        time:1,
        ae:1,
        cwr:1,
        ece:1,
        urg:1,
        ack:1,
        psh:1,
        rst:1,
        syn:1,
        fin:1;
#else
#error  "Adjust your <asm/byteorder.h> defines"
#endif
    __be16  window;
    __sum16 check;
    __be16  urg_ptr;
};
```

Then, I found the function in charge of processing incoming TCP packets, which is `tcp_rcv_established` at line 6263 of `tcp_input.c`. As its name suggests, the function only processes the packets received after the three-way handshake, since during the handshake the bits are used to advertise the algorithm in use. There I called a new custom function in charge of updating the variable for the value to match the value read in the header:

```
void tcp_rcv_established(struct sock *sk, struct sk_buff *skb)
{
    const struct tcphdr *th = (const struct tcphdr *)skb->data;
    struct tcp_sock *tp = tcp_sk(sk);
    unsigned int len = skb->len;

    /*SPIN BIT impl: call custom function to save the spin value*/
    tcp_set_spin_value(sk,th);
```

The new function was implemented at line 6455 inside the same file:

```
/*SPIN BIT impl: save value of the last received packet*/
void tcp_set_spin_value(struct sock *sk, const struct tcphdr *th) {
    if (th->time) sk->sk_spin_value = SPIN_BIT_UP;
    else sk->sk_spin_value = SPIN_BIT_DOWN;
}
```

Obviously, i also needed to insert the function header inside the associated .h file (include/net/tcp.h at line 353):

```
/*SPIN BIT impl: define header for tcp_set_spin_value inside net/ipv4
    /tcp_input.c*/
void tcp_set_spin_value(struct sock *sk, const struct tcphdr *th);
```

All these lines work together to update the internal state of the algorithm.

E.3.4 Implementing the Spin Bit logic

The actual Spin Bit logic is still missing, as it was implemented in the function managing the actual TCP header generation: among the many function enrolled in the output process, the one responsible for building the header is `__tcp_transmit_skb` at line 1446 of `net/ipv4/tcp_output.c`. There let's define the local variable which will be pushed inside the header, and initialize it to the value to write during the handshake phase:

```
/*SPIN BIT impl: here is defined the value to write inside the
    reserved bit*/
u8 spin_value = 0b10; //set to 10 in order to recognize the spin
    algorithm during the handshake (by an external observer)
```

Now the heart of the algorithm can finally be developed: the Spin Bit is very simple as it is sufficient to update the local variable either the last received value if the host is enacting the server's role, or its opposite if the host is filling the role of the client (line 1534):

```
/*SPIN BIT impl: assign value to the variable*/
if (sk->sk_state == TCP_ESTABLISHED && /*!sk->sk_state_change &&*/
    !((tcb->tcp_flags) & TCPHDR_SYN)) {  /*otherwise role and value
    are undefined and might lead to undefined behaviours*/
    if (sk->sk_spin_value == SPIN_BIT_DOWN) spin_value = 0;
    else spin_value = 0b1;
    if (sk->sk_spin_role == SPIN_ROLE_CLIENT) spin_value ^= 0b1; /*
    invert the value if the role is that of the client*/
}
```

And then obviously, the value must be written alongside the rest of the header by shifting it to the right place (line 1541):

```
/* Build TCP header and checksum it. */
th = (struct tcphdr *)skb->data;
th->source        = inet->inet_sport;
th->dest          = inet->inet_dport;
th->seq           = htonl(tcb->seq);
th->ack_seq       = htonl(rcv_nxt);
*(((__be16 *)th) + 6)    = htons(((tcp_header_size >> 2) << 12) |
                  (tcb->tcp_flags & TCPHDR_FLAGS_MASK) |
                  (spin_value << 9)); /*SPIN BIT impl: write value
    inside the header*/
```

On a final note, the logic written above only checks if the segment is a SYN, leaving out the SYNACK case: that's because SYNACK packets are created inside a function of their own, i.e., `tcp_make_synack` at line 3759. There I just had to add inserted the following line (line 3848):

```
/*SPIN BIT impl: mark the header of the SYNACK packet*/
th->loss = 0b1;
```

E.4 Final attentions

These are all the modifications applied to the kernel, and can be used as-is without any configuration to deploy the Spin Bit on any network, provided the conditions for the fragmentation match the requirements defined in section 6.2.1 and both endpoints are patched. To extract the information from an active connection a capable observer is needed, like my Spindump patch, which is discussed thoroughly in appendix G. enums

Appendix F

Kernel Modifications for Delay Bit support

F.1 Inheriting from the Spin Bit experience

The Delay Bit implementation process is very similar to that of the Spin Bit, aside from the one major issue with the fragmentation which I'm going to discuss in the next section. Obviously the process for the editing, compilation and installation of the Delay Bit Kernel is identical (cf. E.1) aside from the fact that the Delay Bit was added as a new branch of the already existing Github repository. The section regarding the position of the marking algorithm advertisement (cf. E.2) applies also to the Delay Bit, and the modifications performed to the kernel too are for the most part completely identical other than from the variables' names. Here I am therefore reporting in brief just the pieces of the code concerning the Delay Bit implementation, with minimal explanations, since it is already provided in section E.3.

F.1.1 The code

Starting from the connection-wide structures in `include/net/sock.h`, only the names are updated (line 164):

```
/*DELAY BIT impl: required to define the role for the algorithm*/
enum delay_role {
    DELAY_ROLE_CLIENT, //sets the spin_value to the opposite of the
    //last received value
    DELAY_ROLE_SERVER, //sets the spin_value to the same value as the
};  //last received one

/*DELAY BIT impl: required to carry the value of the algorithm*/
```

```
enum delay_sample {
	DELAY_BIT_DOWN,
	DELAY_BIT_UP,
};
```

A an additional variable is added to the `struct sock_common` to keep track of the timestamp of the last transmitted delay value, in order to allow the client to regenerate the marking if the segment carrying it is lost (line 198):

```
enum delay_role __skc_delay_role;
/*DELAY BIT impl: required to forward the delay sample*/
enum delay_sample __skc_delay_sample;
ktime_t __skc_ds_time;
```

And of course, including its macro is also necessary (line 413):

```
/*DELAY BIT impl: defining macro to access values defined inside the
    struct sock_common*/
#define sk_delay_role __sk_common.__skc_delay_role
#define sk_delay_sample __sk_common.__skc_delay_sample
#define sk_delay_ds_time __sk_common.__skc_ds_time
```

The role definition in `net/ipv4/tcp.c` remains unvaried, as the Delay Bit too requires the endpoints to adopt different behaviours (line 2576):

```
/*DELAY BIT impl: assign role for a conversation*/
if (oldstate == TCP_SYN_SENT && state == TCP_ESTABLISHED) {
	/*this is the client of the connection*/
	sk->sk_delay_role = DELAY_ROLE_CLIENT;
}
else if (oldstate == TCP_SYN_RECV && state == TCP_ESTABLISHED) {
	/*this is the server of the connection*/
	sk->sk_delay_role = DELAY_ROLE_SERVER;
}
```

The definition of the `struct tcphdr` inside `include/uapi/linux/tcp.h` (line 25) remains the same:

```
struct tcphdr {
	__be16	source;
	__be16	dest;
	__be32	seq;
	__be32	ack_seq;
#if defined(__LITTLE_ENDIAN_BITFIELD)
	__u16	ae:1,
		/*DELAY BIT impl*/
		time: 1,
		loss: 1,
		res1:1,
        doff:4,
        fin:1,
        syn:1,
        rst:1,
        psh:1,
        ack:1,
        urg:1,
        ece:1,
        cwr:1;
#elif defined(__BIG_ENDIAN_BITFIELD)
    __u16   doff:4,
        res1:1,
        /*DELAY impl*/
        loss:1,
        time:1,
        ae:1,
        cwr:1,
        ece:1,
        urg:1,
        ack:1,
        psh:1,
        rst:1,
        syn:1,
        fin:1;
#else
#error  "Adjust your <asm/byteorder.h> defines"
#endif
    __be16  window;
    __sum16 check;
    __be16  urg_ptr;
};
```

Also no changes are applied to the handling of the incoming packets. So the custom function invocation inside `tcp_rcv_established` stays the same (line 6269 of `net/ipv4/tcp_input.c`)

```
/*DELAY BIT impl: define header for tcp_set_spin_value inside net/
   ipv4/tcp_input.c*/
tcp_set_delay_sample(sk,th);
```

as well as its implementation (line 6456)

```
/*DELAY BIT impl: save value of the last received packet*/
void tcp_set_delay_sample(struct sock *sk, const struct tcphdr *th) {
    if (th->time) sk->sk_delay_sample = DELAY_BIT_UP;
}
```

and its definition in `include/net/tcp.h`

```c
/*DELAY BIT impl: define header for tcp_set_delay_sample inside net/ipv4/tcp_input.c*/
void tcp_set_delay_sample(struct sock *sk, const struct tcphdr *th);
```

Lastly, the Delay Bit logic is applied to `__tcp_transmit_skb`, starting with the local variable at line 1462:

```c
/*DELAY BIT impl: here is defined the value to write inside the reserved bit*/
u8 delay_sample = 0; //set to 1 in order to recognize the delay algorithm during the handshake (by an external observer)
```

Its then time to apply the marking to the local variable: as for the Delay Bit the SYN segment must be left alone, then segments with the "selective acknowledgement" option are also barred from being marked as they take a different path inside the network stack and might cause some issues with packet duplication in Cubic. At this point the packet is marked if the algorithm state variable, meaning the Delay Bit has been received, and the delay flag is cleared; then in that case, if the role is that of the client, its associated timestamp is updated. This is necessary as when the flag is not set, the client is required to check how much time has passed since the transmission of the last marked segment: if the difference exceeds a maximum value (here set to 100ms according to the developers' suggested specifics) which must be well above the RTT on the network, a new delay sample is generated. Starting at line 1536:

```c
    /*DELAY BIT impl: assign value to the variable*/
    if (tcb->tcp_flags & TCPHDR_SYN) {
        delay_sample = 0b1;
    }
    else if (opts.num_sack_blocks) { //avoid SACK marking to avoid duplicate packets
        delay_sample = 0;
    }
    else if (sk->sk_delay_sample == DELAY_BIT_UP) {
        sk->sk_delay_sample = DELAY_BIT_DOWN;
        delay_sample = 0b1;
        if (sk->sk_delay_role == DELAY_ROLE_CLIENT) {
            sk->sk_delay_ds_time = ktime_get_coarse(); //reset the timestamp if the role is that of the client
        }
    }
    else if (sk->sk_delay_role == DELAY_ROLE_CLIENT && ktime_to_ms(ktime_sub(ktime_get_coarse(), sk->sk_delay_ds_time)) > 100 ){
        delay_sample = 0b1;
        sk->sk_delay_ds_time = ktime_get_coarse();
    }
```

The result is then push to the newly formed header (line 1555):

```
    /* Build TCP header and checksum it. */
    th = (struct tcphdr *)skb->data;
    th->source      = inet->inet_sport;
    th->dest        = inet->inet_dport;
    th->seq         = htonl(tcb->seq);
    th->ack_seq     = htonl(rcv_nxt);
    *(((__be16 *)th) + 6)   = htons(((tcp_header_size >> 2) << 12) |
                    (tcb->tcp_flags & TCPHDR_FLAGS_MASK) |
                    (delay_sample << 9)); /*DELAY BIT impl: write
    value inside the header*/
```

And lastly, as per the Spin Bit, the Delay Bit handshake codification is added to the TCP header of the SYNACK packet (line 3862):

```
    /*DELAY BIT impl: mark the header of the SYNACK packet*/
    th->time |= 0b1;
```

F.2 Making the Delay Bit compatible with fragmentation

The code as it now stands still doesn't work, since a TCP segment is created without the information regarding the network MTU, and what happens is than as the segment descends the network stack, is tuned into packet and is going to be encapsulated inside a L2 frame, a fragmentation may likely occur. As detailed in section 6.2.1, fragmentation is a real issue for EFM, since as the TCP headers created for each fragment, the reserved bight be copied or not: the delay bit specifically requires the delay sample to be mantained on the first packet and cleared on all others in order to keep working as expected. Fragmentation is handled in dozens of places inside the Linux kernel, but after more than a month of struggle I was able to identify the main function responsible for the actual copy of the TCP header, i.e., `__skb_gso_segment`, inside `net/core/dev.c`. I then changed the code as follows.

In file `include/net/tcp.h` at line 886 I created a mask for the time and loss bits:

```
/*DELAY BIT impl: define mask */
#define TCPHDR_TIME 0x200
#define TCPHDR_LOSS 0x400
```

Then, coming back to `net/core/dev.c` I imported the modified file (line 156)

```
/*DELAY BIT impl: import libraries to allow TCP parse*/
#include <net/tcp.h>
```

Hence I modified `__skb_gso_segment`, starting by adding a struct for the header parsing, and two pointers to the `struct sk_buff`, a pointer to the buffer containing the packet during its processing by the network stack (line 3383):

```
/*DELAY BIT impl: necessary to clear the delay bit*/
struct tcphdr _tcphdr;
struct sk_buff *nskb;
struct sk_buff *next;
```

Then I intervened right after the packet fragmentation by `skb_mac_gso_segment` adding a while loop where I checked if the fragments were TCP segment, and in that case, I cleared the marked bits from the header on each segment saved the first one (line 3419 and onward):

```
segs = skb_mac_gso_segment(skb, features);

/*DELAY BIT impl: try to fix the fragmentation issue for the delay
    bit*/
nskb = segs;
while (nskb) {
    next = nskb->next;
    nskb = next; //first row does not get its bit cleared

    /*DELAY BIT impl: clear the delay bit*/
    if (nskb && nskb->protocol == htons(ETH_P_IP)) {
        //IP confirmed
        struct tcphdr *tcp_header = skb_header_pointer(nskb,
    skb_transport_offset(nskb), sizeof(_tcphdr), &_tcphdr);
        if(tcp_header){
            //TCP confirmed
            *(((__be16 *)tcp_header) + 6) &= ~htons(TCPHDR_TIME);
        }
    }
}
```

Now the Delay Bit works too. The same requirements detailed in section E.4 apply here to ensure the algorithm is correctly deployed.

Appendix G

Spindump patch to extend EFM to TCP

G.1 How does Spindump work

Spindump is an in-network latency measurement tool aimed at providing various connection metrics, like data exchanged and end-to-end latency. In this chapter I'm going to discuss how I patched Spindump in order to allow it to extract the RTT from TCP connections employing either the Spin Bit or the Delay Bit.

Crucially, spindump already supports Spin Bit and Delay Bit for QUIC, which means that it was sufficient to add a control on the functions processing TCP, and to invoke the same functions used in QUIC, being only wary of potential adjustments with the required arguments.

The workflow for implementing the support of both the Spin Bit and the Delay Bit is quite simple: actually I tracked down the flow used in QUIC and mirrored it for TCP, coming down to these five basic steps:

- defining and initializing the data structures that need to maintain the information regarding the protocol for the TCP connection
- defining the function able to extract the information form the TCP header (which method is used and which value is being carried)
- parsing the packet in order to obtain the carried information
- extracting the time values according to the algorithm in place
- invoking the function responsible for the actual RTT evaluations

G.1.1 Managing the project

The project was managed through Github very similarly to what I already explained in section E.1. In order to compile the source code it is sufficient to download the source code from the online repository[16] and then (for Ubuntu linux) running the following command to install the compiler and the essential tools:

```
sudo apt-get install pkg-config cmake make gcc libpcap-dev
    libncurses5-dev libcurl4-openssl-dev libmicrohttpd-dev
```

Then, from the project folder, compiling and installing with gcc:

```
cmake .
make
sudo make install
```

After a new version of the source code is ready, it is sufficient to uninstall the old application before compiling and installing again:

```
sudo make uninstall
```

G.2 Developing the code

Let's now take a look at the modifications I managed to introduce to the application according to the road-map traced above.

G.2.1 The connection-level structures

All connection-level structures in Spindump are saved in a single file named **`spindump_connections_structs.h`**, and are used to keep track of all information that needs to persist during the lifetime of each connection in order to provide the user with the results. There I added a new enum definition needed to save the current EFM technique in TCP (line 54):

```
//ADDED TO ENABLE EFM SUPPORT FOR TCP
enum spindump_tcp_EFM_technique {
  spindump_tcp_no_EFM,
  spindump_tcp_EFM_spin,
  spindump_tcp_EFM_delay,
  spindump_tcp_EFM_delay_plus_q
};
```

Then, in the same file, I updated the `tcp` struct inside the self-explaining `struct spindump_connection` with the new enum and the structs defined for Spin and Delay Bit (line 156):

```c
//ADDED TO ENABLE EFM SUPPORT FOR TCP
enum spindump_tcp_EFM_technique EFM_technique; //necessary to tell
    the EFM techniques apart
struct spindump_spintracker spinFromPeer1to2; // tracking spin bit
    flips from side 1 to 2
struct spindump_spintracker spinFromPeer2to1; // tracking spin bit
    flips from side 2 to 1
struct spindump_delaybittracker delaybitFromPeer1to2; // tracking
    delay bit from side 1 to 2
struct spindump_delaybittracker delaybitFromPeer2to1; // tracking
    delay bit from side 2 to 1
```

The new structures are then to be added to the wider initialization of `struct spindump_connection`: function `spindump_connections_newconnection_aux` inside `spindump_connections_new.c` takes care of that, thus inside it, at line 123 I invoked the creation function and initilized the EFM enum:

```c
//ADDED TO ENABLE EFM SUPPORT FOR TCP
spindump_spintracker_initialize(&connection->u.tcp.spinFromPeer1to2);
spindump_spintracker_initialize(&connection->u.tcp.spinFromPeer2to1);
spindump_delaybittracker_initialize(&connection->u.tcp.
    delaybitFromPeer1to2);
spindump_delaybittracker_initialize(&connection->u.tcp.
    delaybitFromPeer2to1);
connection->u.tcp.EFM_technique = spindump_tcp_no_EFM; /*may not be
    necessary*/
```

G.2.2 Enabling the header's parsing

The next step consists in creating a dedicated function which is able to parse the header of a TCP segment and return the extracted value. To do that I created a new file (which I then had to include inside the make file to ensure its compilation and linking to the rest of the project) called `spindump_analyze_tcp_parser.c` there I defined a function dedicated to recognizing the used algorithm, calling it `spindump_analyze_tcp_parser_check_EFM`, returning the enum which points at the EFM in use (`spindump_tcp_EFM_technique`) initialized accordingly. Then I defined a function to extract the time bit, which would contain the spin or delay value. Both function are very simple and just shift the reference to the TCP header received as only argument to the right amount, and check the value through an AND operation. Here I'm providing the whole file, stripped down of unnecessary comments:

```c
#include <stdlib.h>
#include <string.h>
```

```
#include <limits.h>
#include "spindump_util.h"
#include "spindump_analyze.h"
#include "spindump_analyze_tcp.h"
#include "spindump_analyze_tcp_parser.h"

enum spindump_tcp_EFM_technique
spindump_analyze_tcp_parser_check_EFM(const unsigned char* header) {

  //
  // Sanity checks
  //

  spindump_assert(header != 0);

  uint8_t off_rsvd;

  //spindump_decodebyte(off_rsvd,header,pos);          // ff_rsvd
    gets its memory allocated
  off_rsvd = header[12];

 if ((off_rsvd & 0x06) == 0x04){  //SPIN BIT
     fprintf(stderr, "SPIN IS ACTIVE\n");
     return spindump_tcp_EFM_spin;
  }
  else if ((off_rsvd & 0x06) == 0x02){  //DELAY BIT
     fprintf(stderr, "DELAY IS ACTIVE\n");
     return spindump_tcp_EFM_delay;
  }
  else if ((off_rsvd & 0x06) == 0x06){  //DELAY BIT + Q BIT
     fprintf(stderr, "DELAY*Q ARE ACTIVE\n");
     return spindump_tcp_EFM_delay_plus_q;
  }
  fprintf(stderr, "NO MARKING IS ACTIVE\n");
  return spindump_tcp_no_EFM;
}

int
spindump_analyze_tcp_parser_gettimebit(const unsigned char* header) {
  uint8_t off_rsvd;

  //
  // Sanity check
  //
  spindump_assert(header != 0);

  off_rsvd = header[12];

  //if ((*off_rsvd && 0b00000010) == 0b00000010){  //SPIN OR DELAY
    BIT
  if ((off_rsvd & 0x02) == 0x02){  //SPIN OR DELAY BIT
    return 1;
  }
  else return 0;
}
```

The creation of a new .h file (`spindump_analyze_tcp_parser.h`) was necessary to export the functions:

```
//ADDED TO ENABLE EFM SUPPORT FOR TCP
#include "spindump_connections.h"

enum spindump_tcp_EFM_technique spindump_analyze_tcp_parser_check_EFM
    (const unsigned char* header);
int spindump_analyze_tcp_parser_gettimebit(const unsigned char*
    header);
```

Lastly, `spindump_analyze_tcp_parser.h` was included (for dependency reasons) in `spindump_connections_new.c` at line 44:

```
//ADDED TO ENABLE EFM SUPPORT FOR TCP
#include "spindump_analyze_tcp_parser.h"
```

G.2.3 Parsing the header and invoking the algorithm

Let's head now inside `spindump_analyze_tcp.c`, which contains the function `spindump_analyze_process_tcp`, that, as its name suggest, is responsible for the processing of a TCP segment. Let's first include `spindump_analyze_tcp_parser.h` here too (line 28):

```
//ADDED TO ENABLE EFM SUPPORT FOR TCP
#include "spindump_analyze_tcp_parser.h"
```

`spindump_analyze_process_tcp` manages all possible states of the TCP connection: our first target is that of recognizing the algorithm in use, so I extracted the algorithm inside the if branch managing the SYN packets (line 402):

```
    //ADDED TO ENABLE EFM SUPPORT FOR TCP
    //
    //Check if EFM techniques are used
    //
    connection->u.tcp.EFM_technique =
   spindump_analyze_tcp_parser_check_EFM(packet->contents +
   tcpHeaderPosition);
```

Then it's useful to double check that both endpoints are actually using the same algorithm to avoid providing false measurements, so in the SYNACK branch I added (line 462):

```
//sanity check
enum spindump_tcp_EFM_technique efm =
    spindump_analyze_tcp_parser_check_EFM(packet->contents +
    tcpHeaderPosition);
if (connection->u.tcp.EFM_technique != efm) {
connection->u.tcp.EFM_technique = spindump_tcp_no_EFM; //check for
    correctness: both SYN and SYNACK must carrry the same marking,
                                                                  //
    otherwise, do not apply the efm algorythm
}
```

Finally, if a connection has been established the header parsing can take place. Once the value is retrieved, it is sufficient to call a customized function for either the Spin Bit or Delay Bit RTT evaluation. Both are basically wrappers which extract timestamps from the associated structures I initialized above and then call the real function used to compute the RTT, i.e., `spindump_connections_newrttmeasurement`. So here I just invoked the functions, which are further analyzed in the following sections. It is sufficient to highlight that both receive a parameter which contains the marking, which is the value we just parsed, and a timestamp of the packet being processed. From line 636 onward:

```
//ADDED TO ENABLE EFM SUPPORT FOR TCP
//Normal case of connection established, check if efm is active
if (
     connection->state == spindump_connection_state_established &&
     connection->u.tcp.EFM_technique == spindump_tcp_EFM_spin
) {
     //fprintf(stderr,"spin evaluation is ongoing...");
     //if spin bit is used, retrieve the spin value:
     int spin = spindump_analyze_tcp_parser_gettimebit(packet->
    contents + tcpHeaderPosition);
     //call function for RTT measurement
     int isFlip = 0;
     spindump_spintracker_observespinandcalculatertt(state,packet,
    connection,(struct timeval*)timestamp,spin,fromResponder,
    ipPacketLength,&isFlip);
}
else if (
     connection->state == spindump_connection_state_established && (
        connection->u.tcp.EFM_technique == spindump_tcp_EFM_delay ||
        connection->u.tcp.EFM_technique ==
    spindump_tcp_EFM_delay_plus_q
)) {
     //if delay bit is used, retrieve the delay value:
     int delay = spindump_analyze_tcp_parser_gettimebit(packet->contents + tcpHeaderPosition);
     //call function for RTT measurement
     spindump_delaybittracker_observeandcalculatertt(state,
                                                     packet,
                                                     connection,
                                                     (struct timeval*) timestamp,
                                                     fromResponder, ipPacketLength,
                                                     delay /*spindump_extrameas_delaybit*/);//not actually useful, just meant for retrocompatibility with QUIC

     if (connection->u.tcp.EFM_technique == spindump_tcp_EFM_delay_plus_q){
        //implement q bit logic

     }
}
```

Lastly, let us intervene on `spindump_analyze_process_tcp_markackreceived`, which is the function processing the ACK segments: there we can see that once again `spindump_connections_newrttmeasurement` is invoked, this time though according to the default spindump specifications, i.e., associating the timestamp to which the ACK was received to that of its counterpart and evaluating the RTT according to these values: as for Spin and Delay Bit the function is already invoked, it must not be called twice (aside from the fact that it does not work very well and often does not update its measurements). So it is sufficient to place this condition in front of the two `spindump_connections_newrttmeasurement` invocations (line 138 and line 170):

```
//avoid new rtt measurement if connection is established and any EFM is active
if (connection->u.tcp.EFM_technique == spindump_tcp_no_EFM || connection->state != spindump_connection_state_established)
```

G.2.4 Expanding the Spin Bit for TCP

The `spindump_spintracker_observespinandcalculatertt` is implemented inside `spindump_spin.c` and it's fairly simple: it checks the value and the direction of the spin (from which host it came) to figure out whether a transition (`isFLip`) occurred. If if did occur, it associates the timestamp of the segment with that of the last flip, and then invokes `spindump_connections_newrttmeasurement`. The only modification i did here was replicating the code for TCP to force the functions to use the TCP structures instead. From line 164:

```c
//ADDED TO ENABLE EFM SUPPORT FOR TCP
if(connection->type == spindump_connection_transport_tcp) {
    //mock the original function code but for tcp
    if (fromResponder) {
    tracker = &connection->u.tcp.spinFromPeer2to1;
    otherDirectionTracker = &connection->u.tcp.spinFromPeer1to2;
    } else {
      tracker = &connection->u.tcp.spinFromPeer1to2;
      otherDirectionTracker = &connection->u.tcp.spinFromPeer2to1;
    }

    int spin0to1;
    if (spindump_spintracker_observespin(state,
                                         packet,
                                         connection,
                                         tracker,
                                         ts,
                                         spin,
                                         fromResponder,
                                         ipPacketLength,
                                        &spin0to1)) {

      *isFlip=1;
      //
      // Try to match the spin flip with the most recent matching flip in the other direction.
      // Responder spin flips match with equal flips, initiator flips match with inverse flips.
      //

      struct timeval* otherSpinTime =
        spindump_spintracker_match_bidirectional_spin(otherDirectionTracker,
                                                      1,
                                                      fromResponder ? spin0to1 : !spin0to1);

      if (otherSpinTime) {
        spindump_connections_newrttmeasurement(state,
                                               packet,
                                               connection,
                                               ipPacketLength,
                                               fromResponder, // 0 = left, 1 = right
                                               0, // bidirectional
                                               otherSpinTime,
                                                  ts,
                                                  "SPIN");
      }

      //
      // Match spin with previous in same direction to obtain end to end RTT.
      //

      otherSpinTime = spindump_spintracker_match_unidirectional_spin(tracker, spin0to1);

      if (otherSpinTime) {
        spindump_connections_newrttmeasurement(state,
                                                  packet,
                                                  connection,
                                                  ipPacketLength,
                                                  fromResponder,
                                                  1, // unidirectional
                                                  otherSpinTime,
                                                  ts,
                                                  "SPIN_UNIDIR");
      }
    }

    return;
}
```

G.2.5 Expanding the Delay Bit for TCP

`spindump_delaybittracker_observeandcalculatertt` function can be found implemented inside `spindump_titalia_delaybit.c`. As for the Spin Bit the algorithm is very simple since it just checks whether the delay sample is marked and it associates it to the last sample from the same direction: it retrieves the two timestamps and then it invokes `spindump_connections_newrttmeasurement`. As for the example before, I just doubled the code replacing the structures with those initialized for TCP. From line 42 onward:

```
//ADDED TO ENABLE EFM SUPPORT FOR TCP
if(connection->type == spindump_connection_transport_tcp) {
    if (extrameasbits == 0) return; //carries delay bit info
    //mock the original function code but for tcp
    struct spindump_delaybittracker* tracker;
    struct spindump_delaybittracker* otherTracker;
    if (fromResponder) {
      tracker = &connection->u.tcp.delaybitFromPeer2to1;
      otherTracker = &connection->u.tcp.delaybitFromPeer1to2;
```

```
  } else {
    tracker = &connection->u.tcp.delaybitFromPeer1to2;
    otherTracker = &connection->u.tcp.delaybitFromPeer2to1;
  }

  unsigned long long diff = spindump_timediffinusecs(ts, &tracker->lastDelaySample);
  fprintf(stderr, "DELAY RTT IS BEING CALCULATED: diff = %llu\n", diff);
  if (diff < spindump_delaybit_tmax) {
    spindump_connections_newrttmeasurement(state,
                                           packet,
                                           connection,
                                           ipPacketLength,
                                           fromResponder,
                                           1, // unidirectional
                                          &tracker->lastDelaySample,
                                           ts,
                                           "DELAYBIT_UNIDIR");
  }

  //
  // Try to compute LeftRTT or RightRTT
  //

  diff = spindump_timediffinusecs(ts, &otherTracker->lastDelaySample);
  if (diff < spindump_delaybit_tmax) {
    spindump_connections_newrttmeasurement(state,
                                           packet,
                                           connection,
                                           ipPacketLength,
                                           fromResponder, // 0 = left, 1 = right
                                           0, // bidirectional
                                          &otherTracker->lastDelaySample,
                                           ts,
                                           "DELAYBIT");
  }

  //
  // Save delay sample timestamp
  //

  tracker->lastDelaySample = *ts;

  return;
```

```
}
```

Appendix H

From Spindump captures to connection metrics

H.1 Performing a capture

In order to perform a connection capture with Spindump it is sufficient to use the following command ahead of the beginning of the connection:

```
spindump --interface [interface_name] --textual --format json 2>[
file_name]
```

By doing that, a file is generated containing all data acquired by Spindump for each packet. Having unaggregated metrics allows as to extract the information most suitable to the experiment evaluation. I decided to retrieve second-by-second throughput and latency values, as well as standard deviation for latency, in addition to their connection-wide counterpart. In order to do that i crafted a bash script, which I'm presenting in the next section.

H.2 Parsification and computation

The script below is made up of a main body, which extracts only the JSON objects related to the flow under examination. It is tuned to iperf3 as any iperf3 connection is preceded by another, which is used to exchange between the endpoints the transmission parameter: the script thus only extracts the main connection. Then, Prague and Cubic flows are separated, independently parsed to isolate just the relevant information, and then scanned to evaluate second-by-second and global throughput, latency average, and latency standard deviation. The results by the second are packed in a .csv file, while the global values are kept in a .txt which provides an overview of the whole flow.

```
#!bin/bash

PRAGUE_IP1=
PRAGUE_IP2=
CUBIC_IP1=
CUBIC_IP2=

prague="$PRAGUE_IP1\|PRAGUE_IP2"
cubic="$CUBIC_IP1\|$CUBIC_IP2"

function print_floating(){
    if (( $(echo "$number < 1 && $number != 0" |bc -l) )); then
        number=$(echo "0$number")
    fi
}

function evaluate_avg() {
    avg=$(bc <<< "scale=1; $sum/$count")
}



function evaluate_std_dev() {
    sum_sq=0
    while read line
    do
        rtt_value=$(echo $line | cut -d "," -f 2)
        sum_sq=$(bc <<< "scale=5; $sum_sq+($rtt_value-$avg)*(
    $rtt_value-$avg)")
    done < temp4

    std_dev=$(bc <<< "scale=3; sqrt($sum_sq/$count)")
}



function parse_connection {
    start_time=$(cat temp | head -n 1 | cut -d "," -f 2)
    sum_total=0
    count_total=0

    while read line
    do
        timestamp=$(echo $line | cut -d "," -f 2)
        rtt_value=$(echo $line | cut -d "," -f 3)
        bytes1=$(echo $line | cut -d "," -f 4)
        bytes2=$(echo $line | cut -d "," -f 5)

```

```bash
        timestamp=$(($timestamp-$start_time))
        timestamp=$(($timestamp/1000000))
        rtt_value=$(bc <<< "scale=1; $rtt_value/1000")

        sum_total=$(bc <<< "scale=1; $sum_total+$rtt_value")
        count_total=$(($count_total+1))

        echo "$timestamp,$rtt_value,$((bytes1+$bytes2))"
    done < temp

    global_avg=$(bc <<< "scale=1; $sum_total/$count_total")
}


function evaluate_metrics() {
    sum=0
    count=0
    last_time=-1 #to allow first line computation
    last_bytes=0
    last_sec_bytes=0

    sum_sq_total=0

    while read line
    do
        timestamp=$(echo $line | cut -d "," -f 1)
        rtt_value=$(echo $line | cut -d "," -f 2)
        bytes=$(echo $line | cut -d "," -f 3)

        if [ $timestamp -ne $last_time ]
        then
            if [ $count -ne 0 ] #if no packets were produced avoid calculating metric which could result in errors with arithmetic operations
            then
                evaluate_avg
                cat temp3 | grep "$last_time," > temp4
                evaluate_std_dev
            fi

            if [ $last_time -ne -1 ]
            then
                number=$avg; print_floating; avg=$number
                number=$std_dev; print_floating; std_dev=$number
                number=$(bc <<< "scale=1; ($last_bytes-$last_sec_bytes)/1024/128"); print_floating
                echo "$last_time,$number,$avg,$std_dev"
            fi

```

```
                sum=$rtt_value
                count=1
                last_sec_bytes=$last_bytes
            else
                sum=$(bc <<< "scale=1; $sum+$rtt_value")
                count=$(($count+1))
            fi
            last_time=$timestamp #update last_time
            sum_sq_total=$(bc <<< "scale=5; $sum_sq_total+($rtt_value-
    $global_avg)*($rtt_value-$global_avg)")
            last_bytes=$bytes
        done < temp3

        evaluate_avg
        cat temp3 | grep "$last_time," > temp4
        evaluate_std_dev
        total_bytes=$(($total_bytes-$last_bytes))

        number=$avg; print_floating; avg=$number
        number=$std_dev; print_floating; std_dev=$number
        number=$(bc <<< "scale=1; ($last_bytes-$last_sec_bytes)/1024/128"
    ); print_floating
        echo "$last_time,$number,$avg,$std_dev"

        std_dev_total=$(bc <<< "scale=3; sqrt($sum_sq_total/$count_total)
    ")

        number=$global_avg; print_floating; global_avg=$number
        number=$std_dev_total; print_floating; std_dev_total=$number

        echo "Full connection metrics for $connection" >> $global_metrics
        echo "exchanged data: $((($bytes)/1024/1024)) MBytes" >>
    $global_metrics
        echo "average RTT: $global_avg ms" >> $global_metrics
        echo "RTT standard deviation: $std_dev_total ms" >>
    $global_metrics
        echo "retransmissions: $retr" >> $global_metrics
        echo "" >> $global_metrics
}
function print_floating(){
    if (( $(echo "$number < 1 && $number != 0" |bc -l) )); then
        number=$(echo "0$number")
    fi
}
sudo clear

if [ $# -ne 3 ]
then
    echo "Please the name of the file to parse:"
        read file
        echo "please the number of prague retransmissions"
        read p_retr
        echo "please the number of cubic retransmissions"
        read c_retr
        else
            file=$1
            p_retr=$2
            c_retr=$3
fi

folder="$(echo "$file" | cut -d "." -f 1)_folder"
sudo rm -r "$folder" 2>/dev/null
mkdir $folder 2>/dev/null

file_prague="$folder/$(echo "$file" | cut -d "." -f 1)_prague.csv"
file_cubic="$folder/$(echo "$file" | cut -d "." -f 1)_cubic.csv"
global_metrics="$folder/$(echo "$file" | cut -d "." -f 1)
    _global_metrics.txt"

#get port of first connection initializer
port=$(cat $file | grep TCP | grep "$prague\|$cubic" | head -n 1 |
    cut -d "," -f 5)
#delete first initiator connection
cat $file | grep -v "$port" > temp2

#get port of second connection initializer
port=$(cat temp2 | grep TCP | grep "$prague\|$cubic" | head -n 1 |
    cut -d "," -f 5)
#delete second initiator connection
cat temp2 | grep -v "$port" > temp

#parse useful fields
cat temp | grep TCP | grep Full | grep "$prague\|$cubic" | cut -d ","
        -f 3,6,8,11,12 | cut -d ":" -f 2,3,4,5,6 | cut -d "[" -f 2 | cut
        -d "\"" -f 2,5,7,9,11 > temp2

sed -i 's/"/ /g' temp2
sed -i 's/, //g' temp2
sed -i 's/ : /,/g' temp2

cat temp2 | grep $prague > temp
parse_connection > temp3
connection="PRAGUE"
retr=$p_retr
evaluate_metrics > $file_prague
echo ""
echo "> written into $file_prague:"
cat $file_prague
```

```
echo ""

cat temp2 | grep $cubic > temp
parse_connection > temp3
connection="CUBIC"
retr=$c_retr
evaluate_metrics > $file_cubic
echo "> written into $file_cubic:"
cat $file_cubic
echo ""

echo "> written into $global_metrics:"
cat $global_metrics

cat $prague | sed 's/,/|/g' | sed 's/./,/g' > $prague
cat $cubic | sed 's/,/|/g' | sed 's/./,/g' > $cubic

rm temp
rm temp2
rm temp3
rm temp4

mv $file $folder
```

Bibliography

[1] Briscoe et al. «Low Latency, Low Loss, and Scalable Throughput (L4S) Internet Service: Architecture». In: (Jan. 2023). URL: https://datatracker.ietf.org/doc/rfc9330/ (cit. on p. 3).

[2] Briscoe et al. «Low Latency, Low Loss, and Scalable Throughput (L4S) Internet Service: Architecture». In: (Jan. 2023), 4.3.a. URL: https://datatracker.ietf.org/doc/rfc9330/ (cit. on p. 4).

[3] Briscoe et al. «Prague Congestion Control draft-briscoe-iccrg-prague-congestion-control-02». In: (Mar. 2023), p. 1.4. URL: https://datatracker.ietf.org/doc/draft-briscoe-iccrg-prague-congestion-control/ (cit. on p. 4).

[4] Mamatas et al. «Approaches to Congestion Control in Packet Networks». In: 1 (Jan. 2007), p. 22. URL: https://web.archive.org/web/20140221123729/http://utopia.duth.gr/~emamatas/jie2007.pdf (cit. on p. 4).

[5] pintusaini. «What is ECN(Explicit Congestion Notification)?» In: (Mar. 2022). URL: https://www.geeksforgeeks.org/what-is-ecnexplicit-congestion-notification/ (cit. on p. 5).

[6] Briscoe et al. «More Accurate ECN Feedback in TCP». In: (May 2023). URL: https://datatracker.ietf.org/doc/html/draft-ietf-tcpm-accurate-ecn-22 (cit. on p. 5).

[7] «Broadband Connectivity in the Digital Economy and Society Index». In: (2022). URL: https://digital-strategy.ec.europa.eu/en/policies/desi-connectivity (cit. on p. 8).

[8] Phillip Dampier. «Mobile Data Costs Plummet 88 percent in Five Years, U.S. Consumers Pay 4x More Than Rest of the World». In: (2021). URL: https://stopthecap.com/2021/05/04/mobile-data-costs-plummet-88-in-five-years-u-s-consumers-pay-4x-more-than-rest-of-the-world/ (cit. on p. 8).

[9] William B. Norton. «What are the historical transit pricing trends?» In: (2014). URL: https://drpeering.net/FAQ/What-are-the-historical-transit-pricing-trends.php (cit. on p. 8).

[10] A. Morton. «Active and Passive Metrics and Methods». In: (May 2016). URL: `https://www.rfc-editor.org/rfc/pdfrfc/rfc7799.txt.pdf` (cit. on p. 9).

[11] Cociglio et al. «Multipoint Passive Monitoring in Packet Networks». In: 27 (Dec. 2019). URL: `https://ieeexplore.ieee.org/document/8901157` (cit. on p. 9).

[12] Albisser et al. «DUALPI2 - Low Latency, Low Loss and Scalable Throughput (L4S) AQM». In: (). URL: `https://www.bobbriscoe.net/projects/latency/dualpi2_netdev0x13.pdf` (cit. on p. 11).

[13] Cociglio et al. «Explicit Host-to-Network Flow Measurements Techniques». In: (Oct. 2023). URL: `https://datatracker.ietf.org/doc/rfc9506/` (cit. on p. 12).

[14] Cociglio et al. «Explicit Host-to-Network Flow Measurements Techniques». In: (Oct. 2023), p. 2.1. URL: `https://datatracker.ietf.org/doc/rfc9506/` (cit. on pp. 13, 14).

[15] Torvalds et al. «L4STeam/linux». In: (2023). URL: `https://github.com/L4STeam/linux` (cit. on pp. 18, 24, 48, 67).

[16] Ericsson Research. «spindump». In: (2023). URL: `https://github.com/EricssonResearch/spindump` (cit. on pp. 23, 82).

[17] De Schepper et al. «sch_dualpi2». In: (Sept. 2023). URL: `https://github.com/L4STeam/linux/blob/testing/net/sched/sch_dualpi2.c#L411` (cit. on p. 25).

[18] De Schepper et al. «[L4s-discuss] Configuring a L4S test plant». In: (Oct. 2023), p. 2.4.4. URL: `https://datatracker.ietf.org/doc/draft-briscoe-iccrg-prague-congestion-control/` (cit. on p. 35).

[19] De Schepper et al. «[L4s-discuss] Configuring a L4S test plant». In: (Oct. 2023), p. 2.4.1. URL: `https://datatracker.ietf.org/doc/draft-briscoe-iccrg-prague-congestion-control/` (cit. on p. 36).

[20] Lorem et al. «Introduce delay between each packet». In: (Sept. 2014). URL: `https://stackoverflow.com/questions/25017648/introduce-delay-between-each-packet` (cit. on p. 39).

[21] Ehlinger et al. «[Netem] Emulate jitter without reordering». In: (May 2018). URL: `https://lists.linuxfoundation.org/pipermail/netem/2018-May/001691.html` (cit. on p. 39).

[22] João Carrasqueira. «How to dual-boot Windows 11 and Linux on your PC». In: (2023). URL: `https://www.xda-developers.com/dual-boot-windows-11-linux/` (cit. on p. 47).

[23] «How can I boot with an older kernel version?» In: (2011). URL: https://askubuntu.com/questions/82140/how-can-i-boot-with-an-older-kernel-version (cit. on p. 48).

[24] «What does GRUB DEFAULT=1>2 mean?» In: (2023). URL: https://superuser.com/questions/1358080/what-does-grub-default-12-mean/1358087#1358087 (cit. on p. 48).

[25] Abhishek Prakash. «How to Install Grub Customizer on Ubuntu». In: (2023). URL: https://itsfoss.com/install-grub-customizer-ubuntu/ (cit. on p. 48).

[26] Guarna et al. «[L4s-discuss] Configuring a L4S test plant». In: (2023). URL: https://mailarchive.ietf.org/arch/msg/l4s-discuss/glblrUbYI_CaSuNlSgoZ3deb_xg/ (cit. on p. 60).

[27] Matteo Guarna. «linux_l4s_mod_for_passive_measurements». In: (2023). URL: https://github.com/MatteoGuarna/linux_l4s_mod_for_passive_measurements (cit. on p. 67).

[28] «How to push a shallow clone to a new repo». In: (June 2018). URL: https://stackoverflow.com/questions/50992188/how-to-push-a-shallow-clone-to-a-new-repo/50996201#50996201 (cit. on p. 67).

[29] Vivek Gite. «Debian / Ubuntu Linux Delete Old Kernel Images Command». In: (2023). URL: https://www.cyberciti.biz/faq/debian-ubuntu-linux-delete-old-kernel-images-command/ (cit. on p. 68).